LECTURE 2 (August 29th)

Fast Fourier Transform

Last time we looked at Karatsuba's integer multiplication algorithm which multiplied two n-digit numbers in $O(n^{1.6\ldots})$ time

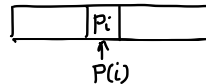
Fast Fourier Transform is one tool which is used to get an $O(n\log n)$ time algorithm

It has many other applications as well and today we will talk about operations on polynomials where Fast Fourier Transform is very useful

Recall that a polynomial is any function of the form

$$p(x) = \sum_{i=0}^{n} p_i x^i$$

Let us represent the polynomial p with an array that holds the values of the coeffecients



There are several operations one could do to polynomials

- Evaluate p on an input x

```
EVALUATE(P[0..n], x):
    X ← 1    ⟨⟨X = x^j⟩⟩
    y ← 0
    for j ← 0 to n
        y ← y + P[j]·X
        X ← X·x
    return y
```

Time = $O(n)$

- Add polynomials p and q

```
ADD(P[0..n], Q[0..n]):
    for j ← 0 to n
        R[j] ← P[j] + Q[j]
    return R[0..n]
```

Time = $O(n)$

- Multiply polynomials p and q

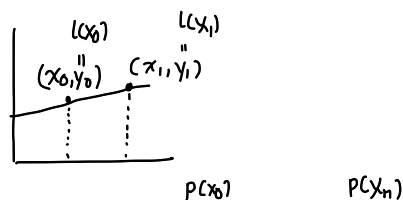
```
MULTIPLY(P[0..n], Q[0..m]):
    for j ← 0 to n + m
        R[j] ← 0
    for j ← 0 to n
        for k ← 0 to m
            R[j + k] ← R[j + k] + P[j]·Q[k]
    return R[0..n + m]
```

Time = $O(n^2)$ → By the end of the lecture we will improve this to $O(n\cdot\log n)$

However, there are other ways of representing polynomials that are also useful

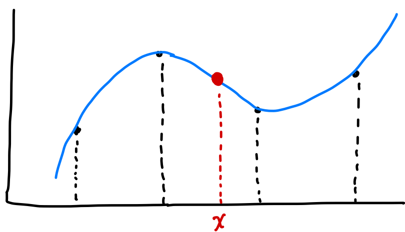
1. Sample points

We know that given a line $l(x) = ax+b$
two points determine the line uniquely



Similarly, for a degree-n polynomial having (n+1)-sample values $(x_0, y_0), \ldots, (x_n, y_n)$ (where $y_0 = p(x_0)$, $y_n = p(x_n)$)
uniquely determines the polynomial

we can choose whatever is convenient



There is exactly one degree 3 polynomial that passes through these 4 pts

In this representation, Addition takes $O(n)$ time
Multiplication takes $O(n)$ time   Needs $2n+1$ sample points
Evaluation — how do we evaluate polynomials in this representation on a new value x?

There is an exact formula by Lagrange from the 1700's to compute the unique polynomial passing through given sample points

$$p(x) = \sum_{j=0}^{n-1} \left(\frac{y_j}{\prod_{k \neq j}(x_j - x_k)} \prod_{k \neq j} (x - x_k) \right)$$

To evaluate this expression, we need $O(n^2)$ time now

We have made multiplication faster but evaluation slower

2. Root representation

Fundamental theorem of algebra says that every polynomial can be written as

$$p(x) = c(x-r_1) \cdots (x-r_n)$$

Evaluation $O(n)$ time
Multiplication $O(n)$ time
Addition ∞ ← Can not compute exactly since roots of $p+q$ might not a nice expression in terms of roots of p and roots of q

To summarize, we are left with two useful representations

representation	evaluate	add	multiply	
coefficients	$O(n)$	$O(n)$	$\mathbf{O(n^2)}$	← Fast Evaluation
~~roots + scale~~	~~$O(n)$~~	~~∞~~	~~$O(n)$~~	
samples	$\mathbf{O(n^2)}$	$O(n)$	$O(n)$	← Fast Multiplication

If we can convert from coefficient to sample representation and back quickly we can do all operations faster — just convert to the representation where that operation is fast, do the operation in that representation and convert back to the original representation

Let's think about what the conversion is. Suppose

$p(x) = \sum_{i=0}^{n-1} a_i x^i$ and sample points are $(x_0, \ldots, x_{n-1})$

$(y_0, \ldots, y_{n-1}) = (p(x_0), \ldots, p(x_{n-1}))$

Then, we have the following relationship

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{n-1} \end{bmatrix}$$

Thus, converting coefficients to sample values is a linear transformation

The space of all degree-n polynomials is a vector space where you use coefficients to specify the polynomial in one basis and sample values in another basis

So, when you are doing this matrix-vector multiplication, what you are doing is converting from one basis to the other

The matrix above is called the Vandermonde matrix. It is guaranteed to not be singular and its determinant is non-zero.

So, we can convert from coefficients to sample values by evaluating this matrix-vector product. How long does it take? $O(n^2)$

But there is a degree of freedom here we haven't exploited — the sample positions. Can we choose them in a clever way to compute this matrix-vector product quickly?

We will describe what properties we want from our sample values first and then we will describe how to satisfy them

So, think in terms of divide-and-conquer algorithms

Recall, Karatsuba's algorithm where we wanted to multiply two integers x and y,

x =	a	b
y =	c	d

and we split them each in parts and multiplied the parts in some way.

We are going to do something that looks strange but it works out better

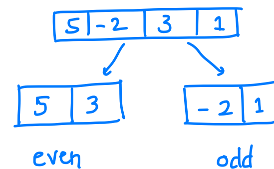
We will write $p(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \dots + a_{n-1} x^{n-1}$

$$= x \cdot P_{odd}(x^2) + P_{even}(x^2)$$

Example, $p(x) = 5 - 2x + 3x^2 + x^3$

then, $P_{odd}(x) = -2 + x$

$P_{even}(x) = 5 + 3x$



Using the above, we want to reduce evaluation of a degree-n polynomial in x to evaluating two degree n/2 polynomials in x^2.

So, we will start with n sample values and when we recurse we only need to evaluate n/2-distinct sample values.

To ensure this, we call a set of sample values X <u>collapsible</u> if

- $|X| = 1$

  or

- $X^2 = \{x^2 \mid x \in X\}$ is collapsible and has $\frac{|X|}{2}$ elements

So, to recurse we will just evaluate the smaller $\frac{n}{2}$-degree polynomials on the set X^2

We have reduced the problem of evaluating a degree-n polynomial on n points to evaluating two degree $\frac{n}{2}$ polynomials on $\frac{n}{2}$ points

So, if T(n) = time to evaluate a polynomial of degree-n on a collapsible set of n sample positions

then, we have the recurrence

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

$$= O(n \cdot \log n)$$

Example : $X = \{1\}$ is collapsible

$X = \{1, 2, 3, 4\}$ is not collapsible

$X = \{\pm 1\}$ is collapsible $\Rightarrow X^2 = \{1\}$

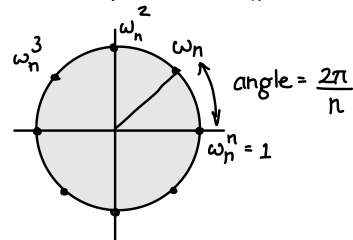
In fact, taking square roots gives us a recipe to construct collapsible sets

$$\{1\} \longrightarrow \{-1, 1\} \longrightarrow \{i, -i, -1, 1\} \longrightarrow \left\{\pm \frac{1}{\sqrt{2}} \pm \frac{1}{\sqrt{2}} i, \pm i, \pm 1\right\}$$

So, we end up with 2^k-th complex roots of unity

Generally, we can use floating point arithmetic when doing signal processing since there is noise in the signal anyway, otherwise we do modular arithmetic modulo some large prime. We will not discuss this here and for the purposes of this class, we can assume that we can compute these square roots exactly.

In general, the nth-root of unity $\omega_n = \cos\left(\frac{2\pi}{n}\right) + i \cdot \sin\left(\frac{2\pi}{n}\right)$



We are going to use the 2^k-roots of unity as our collapsible set and use the divide and conquer strategy

This strategy is called the Fast Fourier Transform

The value of the polynomials on these roots of unity is called the discrete fourier transform (DFT)

DFT of p $\longrightarrow$ $\boxed{P^*[j] := p(\omega_n^j) = \sum_{k=0}^{n-1} P[k] \cdot \omega_n^{jk}}$

If we want to evaluate this, it looks like

```
RADIX2FFT(P[0..n−1]):
  if n = 1
      return P

  for j ← 0 to n/2−1
      U[j] ← P[2j]          ← even      O(n/2) time
      V[j] ← P[2j+1]        ← odd       O(n/2) time

  U* ← RADIX2FFT(U[0..n/2−1])
  V* ← RADIX2FFT(V[0..n/2−1])

  ω_n ← cos(2π/n) + i sin(2π/n)
  ω ← 1

  for j ← 0 to n/2−1
      P*[j]       ← U*[j] + ω · V*[j]
      P*[j+n/2]   ← U*[j] − ω · V*[j]
      ω ← ω · ω_n

  return P*[0..n−1]
```

Everything else takes $O(n)$ time

Overall, $O(n \log n)$ time

Exercise Take a polynomial of degree 4 and compute this by hand

This is known as the Cooley-Tukey algorithm. It was developed by two defense researchers named Cooley and Tukey, building on works by others. They wanted to detect Soviet Nuclear tests from seismic data! The above assumes degree n is a power of two, if not we can just pad it by zeros, but Cooley and Tukey also gave an algorithm for any composite n

But in fact, this algorithm was discovered by Gauss in its full generality in the 1800s who was trying to predict the locations of asteroids.

But he abandoned it because 16th century observations of asteroids were noisy and FFT doesn't work well when there is a lot of noise.

So, Gauss said that because of Kepler's law, I know the orbit has to be an ellipse and I should try to find the best ellipse that matches the data. This led to the invention of the method of least squares. In the process, he used other things that existed but are now named after him.

Going back to our task, we can now convert from coefficients to samples quickly

representation	evaluate	add	multiply	
coefficients	$O(n)$	$O(n)$	$\mathbf{O(n^2)}$	← Fast Evaluation
~~roots + scale~~	~~$O(n)$~~	~~∞~~	~~$O(n)$~~	
samples	$\mathbf{O(n^2)}$	$O(n)$	$O(n)$	← Fast Multiplication

$O(n \log n)$ time (coefficients → samples)

Can we convert back quickly as well? i.e. Can we undo the FFT?

Recall that to convert from coefficients to sample values, we computed the matrix-vector product

$$\vec{y} = V\vec{a} \quad \text{where}$$

$$V = \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)^2} \end{bmatrix}$$

and $\vec{a}$ = array of coefficients

To go back, we need to compute the inverse-transformation V^{-1} and the corresponding matrix-vector product

$$\vec{a} = V^{-1}\vec{y}$$

It turns out that the inverse of this matrix is almost the same matrix

Lemma A.1. $V^{-1} = \overline{V}/n$ ← Short proof can be found in the lecture notes

To compute the inverse FFT, we can just compute the conjugates

```
InverseFFT(P*[0..n−1]):
  P[0..n−1] ← FFT(P*)
  for j ← 0 to n−1
      P[j] ← P̄[j]/n
  return P[0..n−1]
```

```
InverseRadix2FFT(P*[0..n−1]):
  if n = 1
      return P

  for j ← 0 to n/2−1
      U*[j] ← P*[2j]
      V*[j] ← P*[2j+1]

  U ← InverseRadix2FFT(U*[0..n/2−1])
  V ← InverseRadix2FFT(V*[0..n/2−1])

  ω̄_n ← cos(2π/n) − i sin(2π/n)          ← complex conjugate
  ω̄ ← 1

  for j ← 0 to n/2−1
      P[j]       ← (U[j] + ω̄ · V[j])/2     ← takes care of the scaling factor of n
      P[j+n/2]   ← (U[j] − ω̄ · V[j])/2
      ω̄ ← ω̄ · ω̄_n

  return P[0..n−1]
```

So, going back from samples to coefficients is essentially the same process

representation	evaluate	add	multiply
coefficients	$O(n)$	$O(n)$	~~$O(n^2)$~~
~~roots + scale~~	~~$O(n)$~~	~~∞~~	~~$O(n)$~~
samples	~~$O(n^2)$~~	$O(n)$	$O(n)$

$O(n\log n)$ time (coefficients → samples)

$O(n \cdot \log n)$ time (samples → coefficients)

So, to multiply two polynomials in $O(n \cdot \log n)$ time

```
FFTMultiply(P[0..m−1], Q[0..n−1]):
    for j ← m to m+n−1
        P[j] ← 0
    for j ← n to m+n−1
        Q[j] ← 0
    P* ← FFT(P)
    Q* ← FFT(Q)
    for j ← 0 to m+n−1
        R*[j] ← P*[j] · Q*[j]
    return InverseFFT(R*)
```

$O(n \cdot \log n)$ time (annotation covering the two FFT calls and the InverseFFT call)

More generally, the same operation as multiplying two polynomials on sequences or vectors or arrays is called a convolution

If we have two sequences

$$a = (a_0, a_1, \ldots, a_{m-1})$$

$$b = (b_0, b_1, \ldots, b_{n-1})$$

← Think of these as coefficients of two polynomials

then, the convolution of a and b is given by

$$a * b = c = (c_0, \ldots\ldots, c_{m+n-2})$$

$$\text{where } c_k = \sum_{i+j=k} a_i b_j$$

← This is then the coefficient of the product polynomial

Convolution has other applications as we will see on the homeworks and the next lecture.

NEXT LECTURE More Convolution and Dynamic Programming