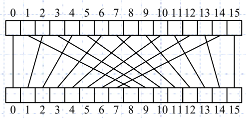
Presentation for use with the textbook, Algorithm Design and Applications, by M. T. Goodrich and R. Tamassia, Wiley, 2015

The Fast Fourier Transform



Polynomials

- Polynomial:

$$p(x) = 5 + 2x + 8x^2 + 3x^3 + 4x^4$$

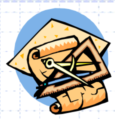
- In general,

$$p(x) = \sum_{i=0}^{n-1} a_i x^i$$

or

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

Polynomial Evaluation



- **Horner's Rule:**
  - Given coefficients $(a_0,a_1,a_2,\ldots,a_{n-1})$, defining polynomial

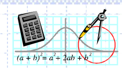
$$p(x) = \sum_{i=0}^{n-1} a_i x^i$$

  - Given x, we can evaluate p(x) in O(n) time using the equation

$$p(x) = a_0 + x(a_1 + x(a_2 + \quad + x(a_{n-2} + xa_{n-1}) \quad))$$

- **Eval**(A,x): [Where $A=(a_0,a_1,a_2,\ldots,a_{n-1})$]
  - If n=1, then return a_0
  - Else,
    - Let $A'=(a_1,a_2,\ldots,a_{n-1})$ [assume this can be done in constant time]
    - return a_0+x***Eval**(A',x)

Polynomial Multiplication Problem



- Given coefficients $(a_0,a_1,a_2,\ldots,a_{n-1})$ and $(b_0,b_1,b_2,\ldots,b_{n-1})$ defining two polynomials, p() and q(), and number x, compute p(x)q(x).
- Horner's rule doesn't help, since

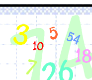
$$p(x)q(x) = \sum_{i=0}^{2n-2} c_i x^i$$

where

$$c_i = \sum_{j=0}^{i} a_j b_{i-j}$$

- A straightforward evaluation would take $O(n^2)$ time. The "magical" FFT will do it in O(n log n) time.

Polynomial Interpolation & Polynomial Multiplication



- Given a set of n points in the plane with distinct x-coordinates, there is **exactly one** (n-1)-degree polynomial going through all these points.
- Alternate approach to computing p(x)q(x):
  - Calculate p() on 2n x-values, $x_0,x_1,\ldots,x_{2n-1}$.
  - Calculate q() on the same 2n x values.
  - Find the (2n-1)-degree polynomial that goes through the points $\{(x_0,p(x_0)q(x_0)), (x_1,p(x_1)q(x_1)), \ldots, (x_{2n-1},p(x_{2n-1})q(x_{2n-1}))\}$.
- Unfortunately, a straightforward evaluation would still take $O(n^2)$ time, as we would need to apply an O(n)-time Horner's Rule evaluation to 2n different points.
- The "magical" FFT will do it in O(n log n) time, by picking 2n points that are easy to evaluate…

Primitive Roots of Unity

- A number ω is a ***primitive n-th root of unity***, for n>1, if
  - $\omega^n = 1$
  - The numbers $1, \omega, \omega^2, \ldots, \omega^{n-1}$ are all distinct
- Example 1:
  - Z^*_{11}:

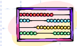
x	x^2	x^3	x^4	x^5	x^6	x^7	x^8	x^9	x^10
1	1	1	1	1	1	1	1	1	1
2	4	8	5	10	9	7	3	6	1
3	9	5	4	1	3	9	5	4	1
4	5	9	3	1	4	5	9	3	1
5	3	4	9	1	5	3	4	9	1
6	3	7	9	10	5	8	4	2	1
7	5	2	3	10	4	6	9	8	1
8	9	6	4	10	3	2	5	7	1
9	4	3	5	1	9	4	3	5	1
10	1	10	1	10	1	10	1	10	1

  - 2, 6, 7, 8 are 10-th roots of unity in Z^*_{11}
  - $2^2=4$, $6^2=3$, $7^2=5$, $8^2=9$ are 5-th roots of unity in Z^*_{11}
  - $2^{-1}=6$, $3^{-1}=4$, $4^{-1}=3$, $5^{-1}=9$, $6^{-1}=2$, $7^{-1}=8$, $8^{-1}=7$, $9^{-1}=5$
- Example 2: The complex number $e^{2\pi i/n}$ is a primitive n-th root of unity, where $i = \sqrt{-1}$

Properties of Primitive Roots of Unity

- **Inverse Property:** If ω is a primitive root of unity, then $\omega^{-1}=\omega^{n-1}$
  - Proof: $\omega\omega^{n-1}=\omega^n=1$
- **Cancellation Property:** For non-zero $-n<k<n$, $\sum_{j=0}^{n-1}\omega^{kj}=0$
  - Proof:
$$\sum_{j=0}^{n-1}\omega^{kj}=\frac{(\omega^k)^n-1}{\omega^k-1}=\frac{(\omega^n)^k-1}{\omega^k-1}=\frac{(1)^k-1}{\omega^k-1}=\frac{1-1}{\omega^k-1}=0$$
- **Reduction Property:** If w is a primitive (2n)-th root of unity, then ω^2 is a primitive n-th root of unity.
  - Proof: If $1,\omega,\omega^2,\ldots,\omega^{2n-1}$ are all distinct, so are $1,\omega^2,(\omega^2)^2,\ldots,(\omega^2)^{n-1}$
- **Reflective Property:** If n is even, then $\omega^{n/2} = -1$.
  - Proof: By the cancellation property, for k=n/2:
$$0=\sum_{j=0}^{n-1}\omega^{(n/2)j}=\omega^0+\omega^{n/2}+\omega^0+\omega^{n/2}+\cdots+\omega^0+\omega^{n/2}=(n/2)(1+\omega^{n/2})$$
  - Corollary: $\omega^{k+n/2}= -\omega^k$.

The Discrete Fourier Transform

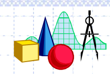


- Given coefficients $(a_0,a_1,a_2,\ldots,a_{n-1})$ for an (n-1)-degree polynomial p(x)
- The **Discrete Fourier Transform** is to evaluate p at the values
  - $1,\omega,\omega^2,\ldots,\omega^{n-1}$
  - We produce $(y_0,y_1,y_2,\ldots,y_{n-1})$, where $y_j=p(\omega^j)$
  - That is, $y_j=\sum_{i=0}^{n-1}a_i\omega^{ij}$
  - Matrix form: **y=Fa**, where **F**$[i,j]=\omega^{ij}$.
- The **Inverse Discrete Fourier Transform** recovers the coefficients of an (n-1)-degree polynomial given its values at $1,\omega,\omega^2,\ldots,\omega^{n-1}$
  - Matrix form: **a=F⁻¹y**, where $\mathbf{F}^{-1}[i,j]=\omega^{-ij}/n$.

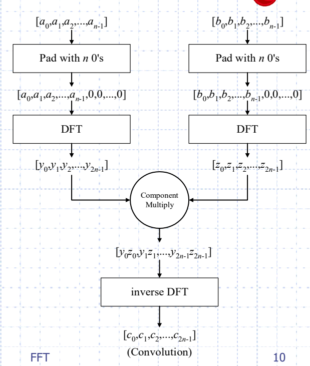
Correctness of the inverse DFT

- The DFT and inverse DFT really are inverse operations
- Proof: Let $\mathbf{A}=\mathbf{F}^{-1}\mathbf{F}$. We want to show that **A=I**, where
$$A[i,j]=\frac{1}{n}\sum_{k=0}^{n-1}\omega^{-ki}\omega^{kj}$$
- If i=j, then
$$A[i,i]=\frac{1}{n}\sum_{k=0}^{n-1}\omega^{-ki}\omega^{ki}=\frac{1}{n}\sum_{k=0}^{n-1}\omega^0=\frac{1}{n}n=1$$
- If i and j are different, then
$$A[i,j]=\frac{1}{n}\sum_{k=0}^{n-1}\omega^{(j-i)k}=0 \qquad \text{(by Cancellation Property)}$$

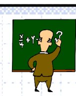
Convolution



- The DFT and the inverse DFT can be used to multiply two polynomials
- So we can get the coefficients of the product polynomial quickly if we can compute the DFT (and its inverse) quickly…



The Fast Fourier Transform



- The FFT is an efficient algorithm for computing the DFT
- The FFT is based on the divide-and-conquer paradigm:
  - If n is even, we can divide a polynomial
$$p(x)=a_0+a_1x+a_2x^2+\cdots+a_{n-1}x^{n-1}$$
    into two polynomials
$$p^{\text{even}}(x) = a_0+a_2x+a_4x^2+\cdots+a_{n-2}x^{n/2-1}$$
$$p^{\text{odd}}(x) = a_1+a_3x+a_5x^2+\cdots+a_{n-1}x^{n/2-1}$$
    and we can write
$$p(x)=p^{\text{even}}(x^2)+xp^{\text{odd}}(x^2).$$

The FFT Algorithm

Algorithm FFT($\mathbf{a},\omega$):
  Input: An n-length coefficient vector $\mathbf{a}=[a_0,a_1,\ldots,a_{n-1}]$ and a primitive nth root of unity ω, where n is a power of 2
  Output: A vector $\mathbf{y}$ of values of the polynomial for $\mathbf{a}$ at the nth roots of unity

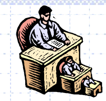
```
if n = 1 then
    return y = a.
x ← ω^0            {x will store powers of ω, so initially x = 1.}
{Divide Step, which separates even and odd indices}
a^even ← [a_0, a_2, a_4, ..., a_{n-2}]
a^odd ← [a_1, a_3, a_5, ..., a_{n-1}]
{Recursive Calls, with ω^2 as (n/2)th root of unity, by the reduction property}
y^even ← FFT(a^even, ω^2)
y^odd ← FFT(a^odd, ω^2)
{Combine Step, using x = ω^i}
for i ← 0 to n/2 − 1 do
    y_i ← y_i^even + x · y_i^odd
    y_{i+n/2} ← y_i^even − x · y_i^odd        {Uses reflective property}
    x ← x · ω
return y
```

The running time is O(n log n). [inverse FFT is similar]

Multiplying Big Integers

- Given N-bit integers I and J, compute IJ.
- Assume: we can multiply words of O(log N) bits in constant time.
- Setup: Find a prime p=cn+1 that can be represented in one word, and set m=(log p)/3, so that we can view I and J as n-length vectors of m-bit words.
- Finding a primitive root of unity.
  - Find a generator x of Z^*_p.
  - Then $\omega=x^c$ is a primitive n-th root of unity in Z^*_p (arithmetic is mod p)
- Apply convolution and FFT algorithm to compute the convolution C of the vector representations of I and J.
- Then compute $K = \sum_{i=0}^{n-1} c_i 2^{mi}$
- K is a vector representing IJ, and takes O(n log n) time to compute.

Non-recursive FFT



- There is also a non-recursive version of the FFT
  - Performs the FFT in place
  - Precomputes all roots of unity
  - Performs a cumulative collection of shuffles on A and on B prior to the FFT, which amounts to assigning the value at index i to the index bit-reverse(i).
- The code is a bit more complex, but the running time is faster by a constant, due to improved overhead

