Polynomial arithmetic and the FFT

For algorithms on polynomials in $R[x]$, running time = number of operations in $R$

Naive algorithms:

- $f + g$: $O(n)$ operations in $R$
- $f \cdot g$: $O(n^2)$ operations in $R$

where $n$ bounds the input degrees

Karatsuba for polynomials:

- uses $O(n^{\log_2 3})$ operations in $R$

We can do better: almost linear time!
Addition Algorithm:

**input:** coeff. vector \((a_0, \ldots, a_{n-1})\) for \(f = \sum_i a_i x^i\),
coeff. vector \((b_0, \ldots, b_{n-1})\) for \(g = \sum_i b_i x^i\)
**output:** coeff. vector \((c_0, \ldots, c_{n-1})\) for \(h = f + g\)

for \(i\) in \([0..n]\) do
\[
c_i \leftarrow a_i + b_i
\]

Naive Multiplication:

**input:** coeff. vector \((a_0, \ldots, a_{n-1})\) for \(f = \sum_i a_i x^i\),
coeff. vector \((b_0, \ldots, b_{n-1})\) for \(g = \sum_i b_i x^i\)
**output:** coeff. vector \((c_0, \ldots, c_{2n-2})\) for \(h = f \cdot g\)

initialize \(c_i \leftarrow 0\) for \(i\) in \([0..2n-2]\)
for \(i\) in \([0..n]\) do
\[
// h \leftarrow h + x^i b_i \cdot f
\]
for \(j \in [0..n]\)
\[
c_{i+j} \leftarrow c_{i+j} + b_i \cdot a_j
\]
Karatsuba multiplication for polynomials

Divide input polynomials $f, g$ into two pieces:

\[
    f = f_{hi}x^k + f_{lo}
\]

\[
    g = g_{hi}x^k + g_{lo},
\]

where $k := \lfloor n/2 \rfloor$

Compute $F \leftarrow f_{hi} + f_{lo}, G \leftarrow g_{hi} + g_{lo}$

Recursively compute three products:

\[
    U \leftarrow f_{hi}g_{hi}, \quad V \leftarrow f_{lo}g_{lo}, \quad W \leftarrow FG
\]

Return $UX^{2k} + (W - U - V)x^k + V$
O(n log n) Polynomial Multiplication

**Domain transformation:** polynomial evaluation and interpolation

We will work over a field $F$, rather than a general ring $R$

Let $p_0, \ldots, p_{n-1}$ be fixed, distinct points in $F$

For any polynomial $f \in F[x]$ of degree $< n$, we define its ***evaluation vector***

$$ (f(p_0), \ldots, f(p_{n-1})) \in F^n $$

We know that $f$ is uniquely determined by its evaluation vector

We can recover $f$ from its coefficient vector using a "polynomial interpolation" algorithm (Lagrange Interpolation)
The high-level strategy:

• Let $f, g \in F[x]$, with $\text{deg}(f) + \text{deg}(g) < n$

• Evaluate $f$ and $g$ at $n$ points, obtaining their evaluation vectors

• Multiply the evaluation vectors element-wise, obtaining

  $$(f(p_0)g(p_0), \ldots, f(p_{n-1})g(p_{n-1}))$$

  This is the evaluation vector of $h := f \cdot g$, and can be computed using $n$ multiplications in $F$

• interpolate $h$’s evaluation vector to obtain $h$
Polynomial evaluation

A matrix point of view

Let \((a_0, \ldots, a_{n-1})\) be \(f\)'s coefficient vector

We can write:

\[
\begin{pmatrix}
  f(p_0) \\
  f(p_1) \\
  \vdots \\
  f(p_{n-1})
\end{pmatrix}
= \begin{pmatrix}
  1 & p_0 & \cdots & p_0^{n-1} \\
  1 & p_1 & \cdots & p_1^{n-1} \\
  \vdots & \vdots & \ddots & \vdots \\
  1 & p_{n-1} & \cdots & p_{n-1}^{n-1}
\end{pmatrix}
\begin{pmatrix}
  a_0 \\
  a_1 \\
  \vdots \\
  a_{n-1}
\end{pmatrix}
\]

Vandermonde matrix \(V(p_0, \ldots, p_{n-1})\)
Computation: a general matrix-vector product takes $O(n^2)$ operations in $F$, so this doesn’t help

But:

• the matrix has a special structure
• we are free to choose the points $p_0, \ldots, p_{n-1}$ to make our lives easier

We will see how to do the evaluation and interpolation steps using $O(n \log n)$ operations in $F$

This immediately gives us a polynomial multiplication algorithm that takes $O(n \log n)$ operations in $F$
The Fast Fourier Transform (FFT)

Assume $n = 2^k$ and $2_F \neq 0_F$

Assume $\omega \in F^*$ has multiplicative order $n$ (sometimes also called a \textit{primitive nth root of unity})

Simple algebraic facts:

- $\omega^n = 1$
- $\omega^i = \omega^j \iff i \equiv j \,(\text{mod } n)$
- $\omega^2$ has multiplicative order $n/2$ (for $n \neq 1$)

Our evaluation points will be

$$p_i := \omega^i \text{ for } i \in [0..n)$$
Example: roots of unity in finite fields

Let $m$ be a prime with $m \equiv 1 \pmod{n}$

Algebraic facts:

- $\mathbb{Z}_m$ is a field with $m$ elements
- $\mathbb{Z}_m^*$ contains elements of order $n$

Example: $m = 17$

- $4$ has order $4$ mod $17$ ($4^2 = 16 \equiv -1 \pmod{17}$)
- $2$ has order $8$ mod $17$ ($2^2 = 4$)
- $6$ has order $16$ mod $17$ ($6^2 \equiv 2 \pmod{17}$)
Example: complex roots of unity

\[ \omega = e^{2\pi i/n} \in \mathbb{C} \]

For \( n = 8 \), \( \omega = \omega_8 \):
FFT: basic idea

Let $f = \sum_{i=0}^{n-1} a_i x^i \in F[x]$.

Split $f$ into “even” and “odd” parts:

\[ f_0 = \sum_{i=0}^{n/2-1} a_{2i} x^i, \quad f_1 = \sum_{i=0}^{n/2-1} a_{2i+1} x^i \]

so that

\[ f(x) = f_0(x^2) + x \cdot f_1(x^2) \]

Plug in $\omega^j$:

\[ f(\omega^j) = f_0(\omega^{2j}) + \omega^j f_1(\omega^{2j}) \]

Key observations: $\omega^2$ has order $n/2$, and $\omega^{2j} = (\omega^2)^k$, where $k = j \mod n/2$. 

Algorithm FFT:

**input:** a primitive $n$th root of unity $\omega$, and a coefficient vector $(a_0, \ldots, a_{n-1})$ for $f = \sum_i a_i x^i$

**output:** the evaluation vector $(f(\omega^0), \ldots, f(\omega^{n-1}))$

if $n = 1$ then
    return $(a_0)$
else
    $(\alpha_0, \alpha_1, \ldots, \alpha_{n/2-1}) \leftarrow FFT(\omega^2, (a_0, a_2, a_4, \ldots, a_{n-2}))$
    $(\beta_0, \beta_1, \ldots, \beta_{n/2-1}) \leftarrow FFT(\omega^2, (a_1, a_3, a_5, \ldots, a_{n-1}))$
    for $j$ in $[0..n)$ do
        $k \leftarrow j \mod n/2$
        $\gamma_j \leftarrow \alpha_k + \omega^j \beta_k$
    return $(\gamma_0, \gamma_1, \ldots, \gamma_{n-1})$
Divide and conquer analysis:

• split into 2 subproblems each of size \( n/2 \)
• local computation cost \( O(n) \)
• Therefore, total time \( O(n \log n) \)
Recursion tree:

Butterfly network:
The inverse FFT

Recall matrix point of view:

\[
\begin{pmatrix}
  f(\omega^0) \\
  f(\omega^1) \\
  \vdots \\
  f(\omega^{n-1}) \\
\end{pmatrix}
= \mathbf{V}
\begin{pmatrix}
  a_0 \\
  a_1 \\
  \vdots \\
  a_{n-1} \\
\end{pmatrix}
\]

where \( \mathbf{V} = \mathbf{V}(\omega^0, \ldots, \omega^{n-1}) \) is the Vandermonde matrix for the points \( \omega^0, \ldots, \omega^{n-1} \).

If \( \mathbf{V} \) is invertible, then

\[
\begin{pmatrix}
  a_0 \\
  a_1 \\
  \vdots \\
  a_{n-1} \\
\end{pmatrix}
= \mathbf{V}^{-1}
\begin{pmatrix}
  f(\omega^0) \\
  f(\omega^1) \\
  \vdots \\
  f(\omega^{n-1}) \\
\end{pmatrix}
\]
So ... interpolation corresponds to multiplication by an inverse Vandermonde matrix

**Theorem.** We have

\[ V^{-1}(\omega^0, \ldots, \omega^{n-1}) = n^{-1} \cdot V(\zeta^0, \ldots, \zeta^{n-1}), \]

where \( \zeta = \omega^{-1} \).

**Notes:**

- \( \zeta = \omega^{n-1} = \omega^{-1} \) is also a primitive \( n \)th root of unity
- since 2 is a unit, so is \( n = 2^k \)

**Implication:** interpolation is just another FFT!! ... plus \( n \) multiplications in \( F \) (by \( n^{-1} \))
Proof of Theorem

We can write $V = (\omega_{ij})$, with indices $i, j \in [0..n)$

Let $W = (\omega^{-ij})$

We want to show that $VW = nI$, where $I$ is the identity matrix

Let $VW = (Z_{ij})$

Using the usual rules of matrix multiplication,

$$z_{ij} = \sum_s \omega^i s \omega^{-sj} = \sum_s \omega^{s(i-j)}$$

For diagonal entries $(i = j)$, we have $z_{ii} = n$

We want to show that $z_{ij} = 0$ for off-diagonal entries $(i \neq j)$
Let $\Delta := i - j \not\equiv 0 \pmod{n}$.

Want to show:

$$\sum_{s=0}^{n-1} \omega^{s\Delta} = 0$$

General fact:

$$(x^n - 1) = (x - 1) \sum_{s=0}^{n-1} x^s$$

Plug in $\omega^\Delta$:

$$(\omega^{n\Delta} - 1) = (\omega^\Delta - 1) \sum_{s=0}^{n-1} \omega^{s\Delta}$$

Finally,

$$\omega^{n\Delta} - 1 = 0 \quad \text{and} \quad \omega^\Delta - 1 \neq 0 \implies \sum_{s=0}^{n-1} \omega^{s\Delta} = 0$$
Summary:

Using the FFT, we can multiply polynomials over $F$ of length less than $n$ using $O(n \log n)$ operations in $F$.

This assumes $2 \in F^*$ and $F$ contains a primitive $n$th root of unity.

For complex roots of unity, one can use floating point approximations. Rounding error has been extensively analyzed in the literature.

For roots of unity in finite fields, there are no issues with rounding.

It is possible to generalize to arbitrary rings: operation count becomes $O(n \log n \log \log n)$.

FFTs were first invented in signal processing. See:

Application: quasi-linear time integer multiplication

Say we want to multiply million ($2^{20}$) bit integers $c = a \cdot b$

Write $a = \sum_i a_i R^i$ and $b = \sum_i b_i R^i$, where $R = 2^{20}$

Set $g := \sum_i a_i x^i \in \mathbb{Z}[x]$ and $h := \sum_i b_i x^i \in \mathbb{Z}[x]$ 

Compute $f := g \cdot h \in \mathbb{Z}[x]$ 

Compute $f(R)$ (linear time) 

- Note that $f(R) = g(R)h(R) = ab$

How to compute $f = g \cdot h$?

Choose a 60-bit prime $p$, with $p \equiv 1 \pmod{2^{20}}$, so $\mathbb{Z}_p^*$ contains an element of order $2^{20}$

Let $\bar{g}$ and $\bar{h}$ be the images of $g$ and $h$ in $\mathbb{Z}_p[x]$ 

Compute $\bar{f} = \bar{g}\bar{h} \in \mathbb{Z}_p[x]$ using FFT

$p$ is large enough so that the coefficients of $f$ are in $[0..p)$, so we can just read them off from the coefficients of $\bar{f}$