Fast Fourier Transform

Michael Levin

Higher School of Economics,
Computer Science Department

Outline

FFT

- The most surprising algorithm

FFT

- The most surprising algorithm
- Used in many advanced competitive programming problems, and also...

FFT

- The most surprising algorithm
- Used in many advanced competitive programming problems, and also...
- ...Audio and video processing

FFT

- The most surprising algorithm
- Used in many advanced competitive programming problems, and also...
- ...Audio and video processing
- Hardware

FFT

- The most surprising algorithm
- Used in many advanced competitive programming problems, and also...
- ...Audio and video processing
- Hardware
- Bioinformatics

FFT

- The most surprising algorithm
- Used in many advanced competitive programming problems, and also...
- ...Audio and video processing
- Hardware
- Bioinformatics
- Convolutional Neural Networks

Outline

Summation

$$
\begin{array}{r|l}
1111111111111111116789 & \\
 & + \\
222222222222222212345 & \\
\hline
3333333333333333319134 &
\end{array}
$$

Summation

$$
\begin{array}{r|l}
1111111111111111116789 & \\
 & + \\
222222222222222212345 & \\
\hline
3333333333333333319134 &
\end{array}
$$

$O(n)$, and can't be faster

Multiplication

$$\begin{array}{r|l} 6789 & \\ & \times \\ 12345 & \\ \hline & \\ & + \\ & \\ \hline & \end{array}$$

Multiplication

$$
\begin{array}{r|c}
6789 & \\
 & \times \\
12345 & \\
\hline
33945 & \\
 & \\
 & + \\
 & \\
\hline
 &
\end{array}
$$

Multiplication

$$
\begin{array}{r|l}
6789 & \\
 & \times \\
12345 & \\
\hline
33945 & \\
27156 & \\
 & + \\
 & \\
\hline
 &
\end{array}
$$

Multiplication

$$
\begin{array}{r|l}
6789 & \\
 & \times \\
12345 & \\
\hline
33945 & \\
27156 & \\
20367 & + \\
 & \\
 & \\
\hline
 &
\end{array}
$$

Multiplication

$$
\begin{array}{r|c}
6789 & \\
 & \times \\
12345 & \\
\hline
33945 & \\
27156 & \\
20367 & + \\
13578 & \\
 & \\
\hline
 &
\end{array}
$$

Multiplication

$$
\begin{array}{r|c}
6789 & \\
 & \times \\
12345 & \\
\hline
33945 & \\
27156 & \\
20367 & + \\
13578 & \\
6789 & \\
\hline
 & \\
\end{array}
$$

Multiplication

$$
\begin{array}{r|c}
6789 & \\
 & \times \\
12345 & \\
\hline
33945 & \\
27156 & \\
20367 & + \\
13578 & \\
6789 & \\
\hline
83810205 &
\end{array}
$$

Multiplication

$$
\begin{array}{r|c}
6789 & \\
 & \times \\
12345 & \\
\hline
33945 & \\
27156 & \\
20367 & + \\
13578 & \\
6789 & \\
\hline
83810205 &
\end{array}
$$

$O(n^2)$ — can we do better?

Faster Multiplication

- Yes, we can

Faster Multiplication

- Yes, we can
- Karatsuba's algorithm $O(n^{\log_2 3})$

Faster Multiplication

- Yes, we can
- Karatsuba's algorithm $O(n^{\log_2 3})$
- You should use it for 1000s and low 10000s of digits

Faster Multiplication

- Yes, we can
- Karatsuba's algorithm $O(n^{\log_2 3})$
- You should use it for 1000s and low 10000s of digits
- But we can do even better asymptotically

Faster Multiplication

- Yes, we can
- Karatsuba's algorithm $O(n^{\log_2 3})$
- You should use it for 1000s and low 10000s of digits
- But we can do even better asymptotically
- And this solution is more general

Outline

$$P(x) = \sum_{i=0}^{n} p_i x^i, \deg(P) = n$$

$$Q(x) = \sum_{j=0}^{m} q_j x^j, \deg(Q) = m$$

$$R(x) = P(x)Q(x)$$

$$P(x) = \sum_{i=0}^{n} p_i x^i, \deg(P) = n$$

$$Q(x) = \sum_{j=0}^{m} q_j x^j, \deg(Q) = m$$

$$R(x) = P(x)Q(x)$$

$$R(x) = \sum_{k=0}^{m+n} \left(\sum_{i=\max(0,k-m)}^{\min(n,k)} p_i q_{k-i} \right) x^k$$

From Polynomials to Integers

- Assume we can multiply polynomials fast
- How to multiply integers fast?

From Polynomials to Integers

- Assume we can multiply polynomials fast
- How to multiply integers fast?

$$a = \overline{d_n d_{n-1} \ldots d_3 d_2 d_1 d_0} \to P_a(x) = \sum_{i=0}^{n} d_i x^i$$

From Polynomials to Integers

- Assume we can multiply polynomials fast
- How to multiply integers fast?

$$a = \overline{d_n d_{n-1} \dots d_3 d_2 d_1 d_0} \to P_a(x) = \sum_{i=0}^{n} d_i x^i$$

$$P_a(10) = a$$

From Polynomials to Integers

- Assume we can multiply polynomials fast
- How to multiply integers fast?

$$a = \overline{d_n d_{n-1} \dots d_3 d_2 d_1 d_0} \to P_a(x) = \sum_{i=0}^{n} d_i x^i$$

$$P_a(10) = a$$

Coeffs of $P_a P_b$ are **almost** the digits of ab

NaivePolyMult(P, Q)

$n \leftarrow \deg(P), m \leftarrow \deg(Q)$
$R \leftarrow$ array of length $m + n + 1$
for k from 0 to $m + n$:
  $R[k] \leftarrow 0$
  $l \leftarrow \max(0, k - m)$
  $r \leftarrow \min(n, k)$
  for i from l to r:
    $R[k] \leftarrow R[k] + P[i]Q[k - i]$
return R

NaivePolyMult(P, Q)

```
n ← deg(P), m ← deg(Q)
R ← array of length m + n + 1
for k from 0 to m + n:
    R[k] ← 0
    l ← max(0, k − m)
    r ← min(n, k)
    for i from l to r:
        R[k] ← R[k] + P[i]Q[k − i]
return R
```

$O(n^2)$ — too slow again

Alternative Representations

By coefficients: $P(x) = \sum_{i=0}^{n} a_i x^i$

Alternative Representations

By coefficients: $P(x) = \sum_{i=0}^{n} a_i x^i$

By values: $P(x_i) = y_i$

Alternative Representations

By coefficients: $P(x) = \sum_{i=0}^{n} a_i x^i$

By values: $P(x_i) = y_i$

How many values uniquely determine a polynomial of degree n?

Idea

- Instead of computing $R(x) = P(x)Q(x)$,

Idea

- Instead of computing $R(x) = P(x)Q(x)$,
- First compute values $P(x_i)$ and $Q(x_i)$ for some $x_i, 0 \leq i \leq k$

Idea

- Instead of computing $R(x) = P(x)Q(x)$,
- First compute values $P(x_i)$ and $Q(x_i)$ for some $x_i, 0 \leq i \leq k$
- Then compute $R(x_i) = P(x_i)Q(x_i)$

Idea

- Instead of computing $R(x) = P(x)Q(x)$,
- First compute values $P(x_i)$ and $Q(x_i)$ for some $x_i, 0 \le i \le k$
- Then compute $R(x_i) = P(x_i)Q(x_i)$
- Then compute $R(x)$ given its values

Idea

- Instead of computing $R(x) = P(x)Q(x)$,
- First compute values $P(x_i)$ and $Q(x_i)$ for some $x_i, 0 \leq i \leq k$
- Then compute $R(x_i) = P(x_i)Q(x_i)$
- Then compute $R(x)$ given its values
- How to choose k?

From Coefficients to Values

- Just compute the values
- Use Horner-Schema

```
ComputePoly(P, x)

n ← deg(P)
result ← 0
for i from n down to 0:
    result ← result · x + P[i]
return result
```

From Values to Coefficients

Given:

$$\deg(P) = n,$$

$$P(x_0) = y_0, P(x_1) = y_1, \ldots, P(x_n) = y_n$$

Find:

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

Interpolation

Solution:

$$\begin{aligned}P(x) = {} & y_0 \frac{(x - x_1)(x - x_2)\dots(x - x_n)}{(x_0 - x_1)(x_0 - x_2)\dots(x_0 - x_n)} + \\ & + y_1 \frac{(x - x_0)(x - x_2)\dots(x - x_n)}{(x_1 - x_0)(x_1 - x_2)\dots(x_1 - x_n)} + \dots \\ & + y_n \frac{(x - x_0)(x - x_1)\dots(x - x_{n-1})}{(x_n - x_0)(x_n - x_1)\dots(x_n - x_{n-1})}\end{aligned}$$

Alternative Multiplication

$$\begin{tikzcd}
P(x), Q(x) \arrow[r, "O(n^2)"] \arrow[d] & PQ(x) \\
(P(x_i), Q(x_i))_{i=1}^{k} \arrow[r, "O(n)"] & (PQ(x_i))_{i=1}^{k} \arrow[u]
\end{tikzcd}$$

Alternative Multiplication

$$\begin{tikzcd}[column sep=huge, row sep=huge]
P(x), Q(x) \arrow[r, "O(n^2)"] \arrow[d, "O(n^2)"] & PQ(x) \\
(P(x_i), Q(x_i))_{i=1}^{k} \arrow[r, "O(n)"] & (PQ(x_i))_{i=1}^{k} \arrow[u, "O(n^2)"']
\end{tikzcd}$$

- Horner-Schema is $O(\deg(P))$

- Horner-Schema is $O(\deg(P))$
- Computing $O(n)$ values of degree n polynomial is $O(n^2)$

- Horner-Schema is $O(\deg(P))$
- Computing $O(n)$ values of degree n polynomial is $O(n^2)$
- Interpolation is $O(n^3)$ or even $O(2^n)$:) if implemented in a straightforward way

- Horner-Schema is $O(\deg(P))$
- Computing $O(n)$ values of degree n polynomial is $O(n^2)$
- Interpolation is $O(n^3)$ or even $O(2^n)$:) if implemented in a straightforward way
- Can interpolate in $O(n^2)$: precompute $(x - x_0)(x - x_1)\dots(x - x_{n-1})(x - x_n)$, then only divide

- Horner-Schema is $O(\deg(P))$
- Computing $O(n)$ values of degree n polynomial is $O(n^2)$
- Interpolation is $O(n^3)$ or even $O(2^n)$:) if implemented in a straightforward way
- Can interpolate in $O(n^2)$: precompute $(x - x_0)(x - x_1) \dots (x - x_{n-1})(x - x_n)$, then only divide
- Still too slow

Idea

- This approach enables us to choose any different $x_i, 0 \leq i \leq k$
- Let's select the most convenient x_is!

Take Consecutive Points

Compute consecutive values

Input: Polynomial $P(x)$ of degree n given by values $P(0), P(1), \ldots, P(n)$

Output: $P(n+1), P(n+2), \ldots, P(n+k)$

Take Consecutive Points

Compute consecutive values

Input: Polynomial $P(x)$ of degree n given by values $P(0), P(1), \ldots, P(n)$

Output: $P(n+1), P(n+2), \ldots, P(n+k)$

Idea: note that $Q(x) = P(x+1) - P(x)$ is a polynomial of degree $n-1$

Take Consecutive Points

Compute consecutive values

Input: Polynomial $P(x)$ of degree n given by values $P(0), P(1), \ldots, P(n)$

Output: $P(n+1), P(n+2), \ldots, P(n+k)$

Idea: note that $Q(x) = P(x+1) - P(x)$ is a polynomial of degree $n-1$

Unfortunately, this would still take $O(n^2)$ for the first $n+1$ values

Outline

Definition

ε is called a complex root of unity of degree n iff $\varepsilon^n = 1$

There are exactly n complex roots of unity

Definition

ε_n is called a primitive root of unity of degree n iff $\varepsilon_n^n = 1$ and all complex roots of unity are $\varepsilon_n, \varepsilon_n^2, \varepsilon_n^3, \ldots, \varepsilon_n^{n-1}, \varepsilon_n^n = 1$.

Lemma

There exists a primitive root of unity

Proof

$\varepsilon_n \stackrel{def}{=} e^{\frac{2\pi i}{n}} = \cos(\frac{2\pi}{n}) + i\sin(\frac{2\pi}{n})$ is a primitive root of unity $\square$

Lemma

If ε_{2k} is a primitive root of unity of degree $2k$, then $\varepsilon_k = \varepsilon_{2k}^2$ is a primitive root of unity of degree k. Moreover,

$$\left[\varepsilon_{2k}^0, \varepsilon_{2k}^1, \ldots, \varepsilon_{2k}^{2k-1}\right]^2 \stackrel{def}{=} \left[(\varepsilon_{2k}^0)^2, (\varepsilon_{2k}^1)^2, (\varepsilon_{2k}^2)^2, \ldots, (\varepsilon_{2k}^{2k})^2\right] = \left[\varepsilon_k^0, \varepsilon_k^1, \varepsilon_k^2, \ldots, \varepsilon_k^{k-1}, \varepsilon_k^0, \varepsilon_k^1, \ldots, \varepsilon_k^{k-1}\right]$$

Proof

$$(e^{\frac{2\pi i}{2k}})^2 = e^{\frac{2\pi i}{k}}$$

□

Outline

Unexpected Idea

- Let's take $x_i = \varepsilon_n^i$ and compute $(P(x_i))_{i=1}^n$ faster than $O(n^2)$

Unexpected Idea

- Let's take $x_i = \varepsilon_n^i$ and compute $(P(x_i))_{i=1}^n$ faster than $O(n^2)$
- Assume for simplicity that $\deg(P) = 2^k - 1$ (otherwise pad with leading zero coefficients)

Matrix Transform

$$P(x) = \sum_{i=0}^{n-1} p_i x^i$$

$$\begin{pmatrix} P(x_0) \\ P(x_1) \\ \dots \\ P(x_{n-1}) \end{pmatrix} =$$

$$\begin{pmatrix} (\varepsilon_n^0)^0 & (\varepsilon_n^0)^1 & \dots & (\varepsilon_n^0)^{n-1} \\ (\varepsilon_n^1)^0 & (\varepsilon_n^1)^1 & \dots & (\varepsilon_n^1)^{n-1} \\ \dots & & & \\ (\varepsilon_n^{n-1})^0 & (\varepsilon_n^{n-1})^1 & \dots & (\varepsilon_n^{n-1})^{n-1} \end{pmatrix} \begin{pmatrix} p_0 \\ p_1 \\ \dots \\ p_{n-1} \end{pmatrix}$$

Inverse Transform

$$\overline{\varepsilon_n} = \varepsilon_n^{-1} = \cos(\frac{2\pi}{n}) - i\sin(\frac{2\pi}{n})$$

Inverse Transform

$$\overline{\varepsilon_n} = \varepsilon_n^{-1} = \cos(\frac{2\pi}{n}) - i\sin(\frac{2\pi}{n})$$

$$F = \begin{pmatrix} (\varepsilon_n^0)^0 & (\varepsilon_n^0)^1 & \dots & (\varepsilon_n^0)^{n-1} \\ (\varepsilon_n^1)^0 & (\varepsilon_n^1)^1 & \dots & (\varepsilon_n^1)^{n-1} \\ \dots & & & \\ (\varepsilon_n^{n-1})^0 & (\varepsilon_n^{n-1})^1 & \dots & (\varepsilon_n^{n-1})^{n-1} \end{pmatrix}$$

Inverse Transform

$$\overline{\varepsilon_n} = \varepsilon_n^{-1} = \cos(\tfrac{2\pi}{n}) - i\sin(\tfrac{2\pi}{n})$$

$$F = \begin{pmatrix} (\varepsilon_n^0)^0 & (\varepsilon_n^0)^1 & \dots & (\varepsilon_n^0)^{n-1} \\ (\varepsilon_n^1)^0 & (\varepsilon_n^1)^1 & \dots & (\varepsilon_n^1)^{n-1} \\ \dots & & & \\ (\varepsilon_n^{n-1})^0 & (\varepsilon_n^{n-1})^1 & \dots & (\varepsilon_n^{n-1})^{n-1} \end{pmatrix}$$

$$F^{-1} = \frac{1}{n} \begin{pmatrix} (\overline{\varepsilon_n}^0)^0 & (\overline{\varepsilon_n}^0)^1 & \dots & (\overline{\varepsilon_n}^0)^{n-1} \\ (\overline{\varepsilon_n}^1)^0 & (\overline{\varepsilon_n}^1)^1 & \dots & (\overline{\varepsilon_n}^1)^{n-1} \\ \dots & & & \\ (\overline{\varepsilon_n}^{n-1})^0 & (\overline{\varepsilon_n}^{n-1})^1 & \dots & (\overline{\varepsilon_n}^{n-1})^{n-1} \end{pmatrix}$$

Proof

Consider any element of the matrix product FF^{-1}:

$$x_{ij} = \sum_{k=1}^{n} (\varepsilon_n^{i-1})^{k-1} (\overline{\varepsilon_n}^{k-1})^{j-1} =$$

$$= \sum_{k=1}^{n} \varepsilon_n^{(k-1)(i-1)-(k-1)(j-1)} = \sum_{k=1}^{n} (\varepsilon_n^{i-j})^{k-1}$$

Proof

If $i = j$, $x_{ij} = \sum_{k=1}^{n} 1^{k-1} = n$.

Otherwise, $x_{ij} = 0$: denote $\alpha = \varepsilon_n^{i-j}$. Then $\alpha \neq 1$ is a root of unity of degree n. Then

$$0 = (\alpha^n - 1) = (\alpha - 1) \sum_{k=1}^{n} \alpha^{k-1} \Rightarrow$$

$$\sum_{k=1}^{n} \alpha^{k-1} = 0$$

$\square$

Inverse Transform

- Inverse transform is almost the same as direct transform
- Just replace ε_n with $\overline{\varepsilon_n}$, do the transform with such ε and multiply the result by $\frac{1}{n}$
- If we learn to perform the direct transform fast, we'll be able to perform the inverse transform fast as well

Outline

Divide and Conquer

$$P(\varepsilon) = p_0\varepsilon^0 + p_1\varepsilon^1 + \cdots + p_{2^k-1}\varepsilon^{2^k-1} =$$

$$= (p_0\varepsilon^0 + p_2\varepsilon^2 + \cdots + p_{2^k-2}\varepsilon^{2^k-2}) +$$

$$+\varepsilon(p_1\varepsilon^0 + p_3\varepsilon^2 + \cdots + p_{2^k-1}\varepsilon^{2^k-2}) =$$

Divide and Conquer

$$P_{even} \stackrel{def}{=} p_0 + p_2 x^2 + p_4 x^4 + \cdots + p_{2^k-2} x^{2^k-2}$$
$$P_{odd} \stackrel{def}{=} p_1 + p_3 x^2 + p_5 x^4 + \cdots + p_{2^k-1} x^{2^k-2}$$
$$P = P_{even} + x P_{odd}$$

Divide and Conquer

$$FFT(P) = \begin{pmatrix} P(\varepsilon_{2^k}^0) \\ P(\varepsilon_{2^k}^1) \\ \dots \\ P(\varepsilon_{2^k}^{2^k-1}) \end{pmatrix} = \begin{pmatrix} P_{even}((\varepsilon_{2^k}^0)^2) \\ P_{even}((\varepsilon_{2^k}^1)^2) \\ \dots \\ P_{even}((\varepsilon_{2^k}^{2^k-1})^2) \end{pmatrix} +$$

$$\varepsilon_{2^k} \begin{pmatrix} P_{odd}((\varepsilon_{2^k}^0)^2) \\ P_{odd}((\varepsilon_{2^k}^1)^2) \\ \dots \\ P_{odd}((\varepsilon_{2^k}^{2^k-1})^2) \end{pmatrix}$$

Divide and Conquer

$$\begin{pmatrix} P_{even}((\varepsilon_{2^k}^0)^2) \\ P_{even}((\varepsilon_{2^k}^1)^2) \\ \cdots \\ P_{even}((\varepsilon_{2^k}^{2^k-1})^2) \end{pmatrix} = \begin{pmatrix} P_{even}((\varepsilon_{2^k}^2)^0) \\ P_{even}((\varepsilon_{2^k}^2)^1) \\ \cdots \\ P_{even}((\varepsilon_{2^k}^2)^{2^k-1}) \end{pmatrix} =$$

$$= \begin{pmatrix} P_{even}(\varepsilon_{2^{k-1}}^0) \\ P_{even}(\varepsilon_{2^{k-1}}^1) \\ \cdots \\ P_{even}(\varepsilon_{2^{k-1}}^{2^{k-1}-1}) \\ \texttt{Same } 2^{k-1} \texttt{ rows again} \end{pmatrix} = \begin{pmatrix} FFT(P_{even}) \\ FFT(P_{even}) \end{pmatrix}$$

Divide and Conquer

$$FFT(P) =$$

$$\begin{pmatrix} FFT(P_{even}) \\ FFT(P_{even}) \end{pmatrix} + \begin{pmatrix} \varepsilon_{2^k}^{0} \\ \varepsilon_{2^k}^{1} \\ \dots \\ \varepsilon_{2^k}^{2^k-1} \end{pmatrix} \odot \begin{pmatrix} FFT(P_{odd}) \\ FFT(P_{odd}) \end{pmatrix}$$

$\odot$ — component-wise multiplication

Divide and Conquer

Note that $\varepsilon_{2^k}^{i+2^{k-1}} = -\varepsilon_{2^k}^{i}$

FFT(P, $invert$)

$k \leftarrow 1$

while $2^k \leq \deg(P)$:

  $k \leftarrow k + 1$

P.resize(2^k)

$\varepsilon \leftarrow \cos\left(\frac{2\pi}{2^k}\right) + i \sin\left(\frac{2\pi}{2^k}\right)$

C $\leftarrow 1$

if invert:

  $\varepsilon \leftarrow \cos\left(\frac{2\pi}{2^k}\right) - i \sin\left(\frac{2\pi}{2^k}\right)$

  C $\leftarrow \frac{1}{2^k}$

return C $\cdot$ FFTRecursive(P, k, ε)

FFTRecursive(P, k, ε)

```
if k == 1:
   return [P[0] + P[1], P[0] − P[1]]
P_even ← [P[0], P[2], ..., P[2^k − 2]]
P_odd ← [P[1], P[3], ..., P[2^k − 1]]
F_even ← FFTRecursive(P_even, k − 1, ε^2)
F_odd ← FFTRecursive(P_odd, k − 1, ε^2)
for i from 0 to 2^(k−1) − 1:
   F[i] ← F_even[i] + ε^i F_odd[i]
   F[i + 2^(k−1)] ← F_even[i] − ε^i F_odd[i]
return F
```

Asymptotics

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

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

Fast Multiplication

$$\begin{tikzcd}
P(x), Q(x) \arrow[r, "O(n^2)"] \arrow[d, "O(n \log n)"] & PQ(x) \\
(P(x_i), Q(x_i))_{i=1}^{k} \arrow[r, "O(n)"] & (PQ(x_i))_{i=1}^{k} \arrow[u, "O(n \log n)"']
\end{tikzcd}$$

Convolution

$$R(x) = \sum_{k=0}^{m+n} \left(\sum_{i=\max(0,k-m)}^{\min(n,k)} p_i q_{k-i} \right) x^k$$

Convolution

$$P = (p_i)_{i=1}^{n}, Q = (q_j)_{j=1}^{m}$$

$$P \circ Q = \left(\sum_{i=\max(0,k-m)}^{\min(n,k)} p_i q_{k-i} \right)$$

Operation $\circ$ — 1-d convolution

Convolution

$$P = (p_i)_{i=1}^{n}, Q = (q_j)_{j=1}^{m}$$

$$P \circ Q = \left(\sum_{i=\max(0,k-m)}^{\min(n,k)} p_i q_{k-i} \right)$$

Operation $\circ$ — 1-d convolution

Fourier Transform maps convolution to product and vice versa

Outline

0	1	2	3	4	5	6	7

0	1	2	3	4	5	6	7

0	2	4	6	1	3	5	7

0	1	2	3	4	5	6	7

0	2	4	6	1	3	5	7

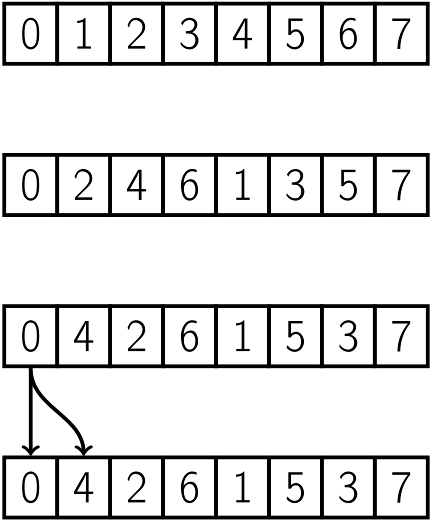
0	4	2	6	1	5	3	7

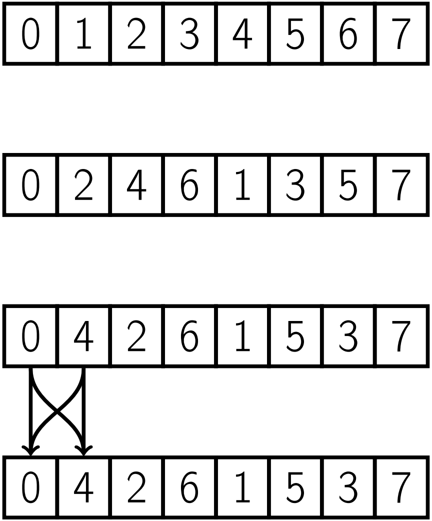
0	1	2	3	4	5	6	7

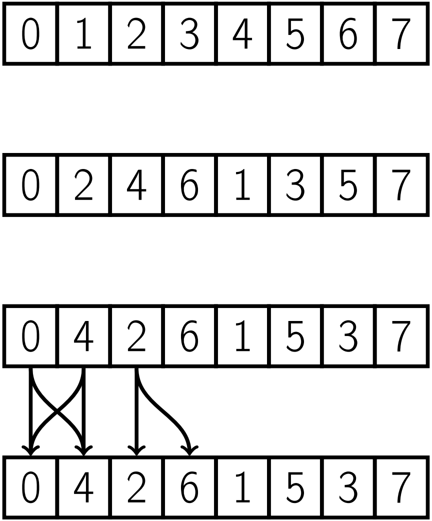
0	2	4	6	1	3	5	7

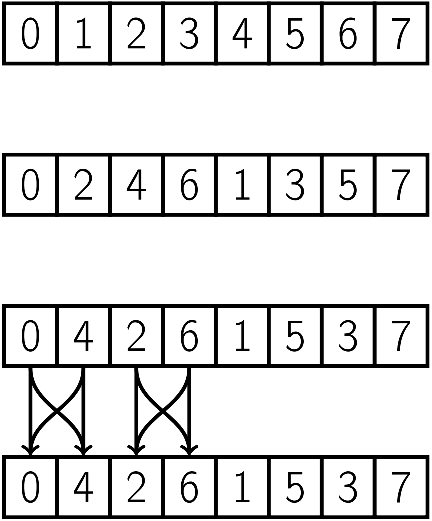
0	4	2	6	1	5	3	7

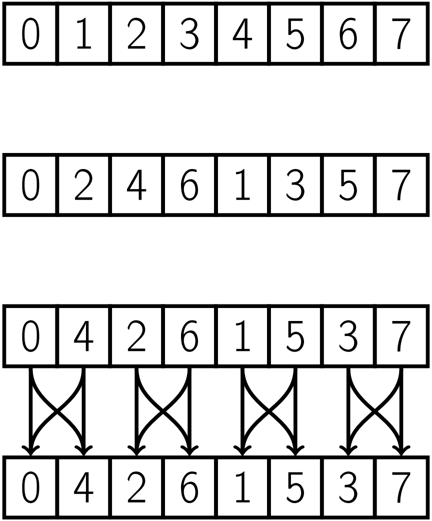
0	4	2	6	1	5	3	7

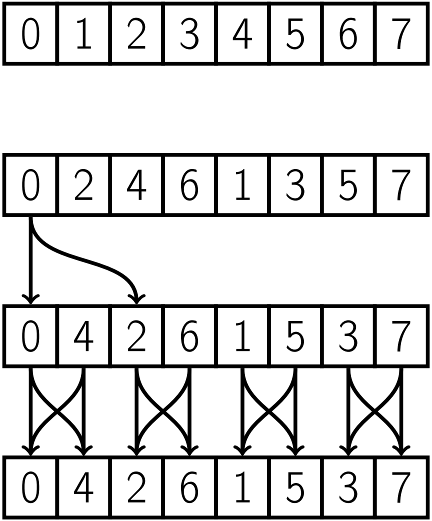

0 1 2 3 4 5 6 7
0 2 4 6 1 3 5 7
0 4 2 6 1 5 3 7
0 4 2 6 1 5 3 7


0 1 2 3 4 5 6 7
0 2 4 6 1 3 5 7
0 4 2 6 1 5 3 7
0 4 2 6 1 5 3 7

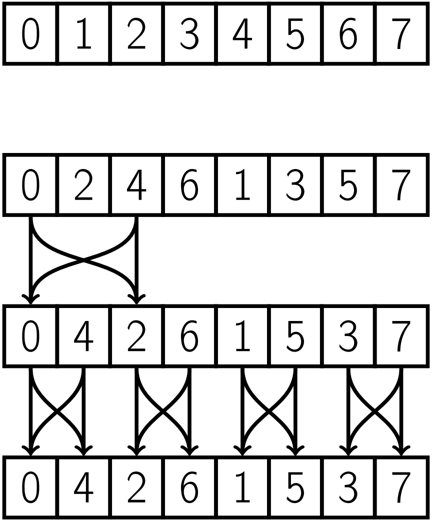

0 1 2 3 4 5 6 7
0 2 4 6 1 3 5 7
0 4 2 6 1 5 3 7
0 4 2 6 1 5 3 7


0 1 2 3 4 5 6 7
0 2 4 6 1 3 5 7
0 4 2 6 1 5 3 7
0 4 2 6 1 5 3 7

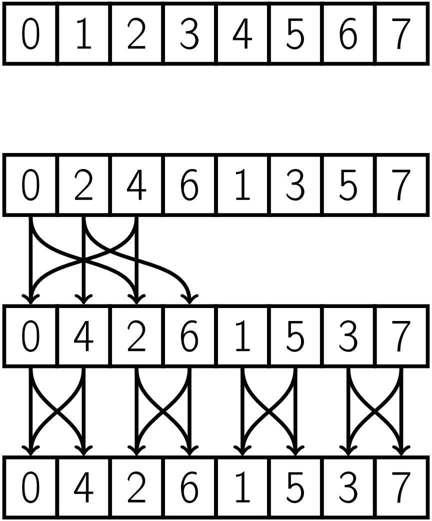

0 1 2 3 4 5 6 7
0 2 4 6 1 3 5 7
0 4 2 6 1 5 3 7
0 4 2 6 1 5 3 7

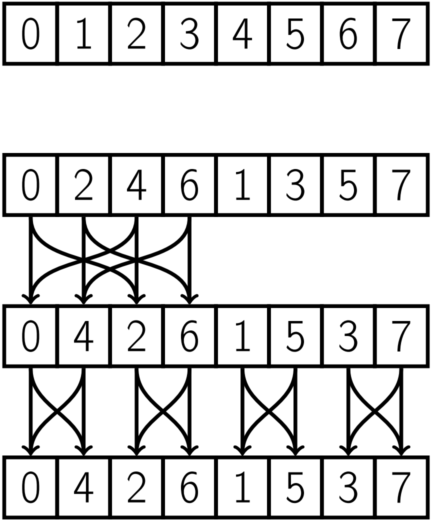


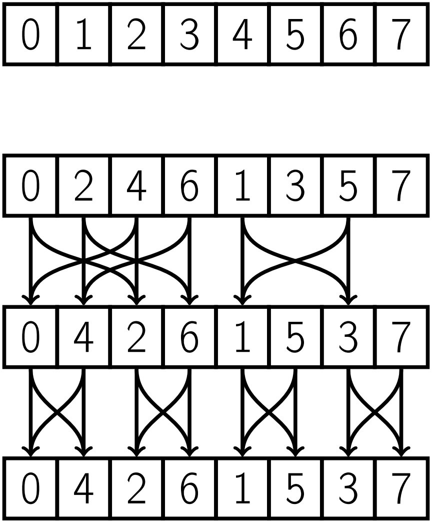
0 1 2 3 4 5 6 7
0 2 4 6 1 3 5 7
0 4 2 6 1 5 3 7
0 4 2 6 1 5 3 7

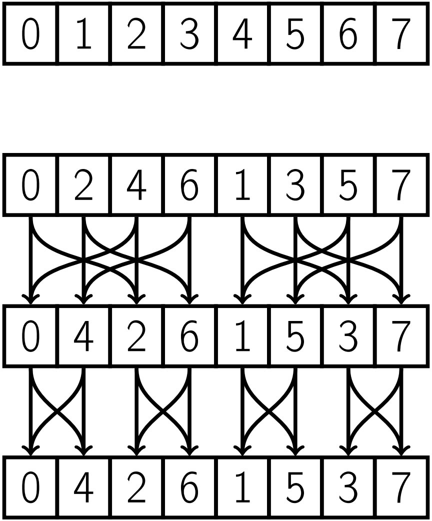


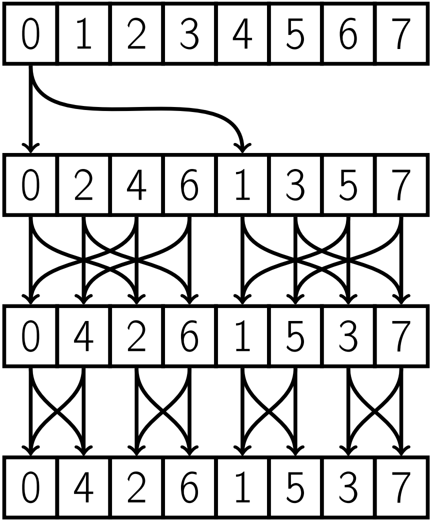
0 1 2 3 4 5 6 7
0 2 4 6 1 3 5 7
0 4 2 6 1 5 3 7
0 4 2 6 1 5 3 7

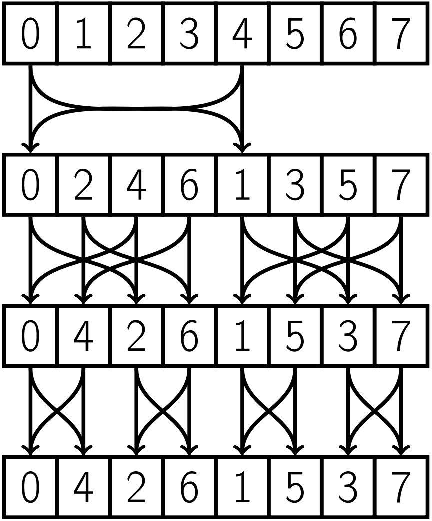

0 1 2 3 4 5 6 7
0 2 4 6 1 3 5 7
0 4 2 6 1 5 3 7
0 4 2 6 1 5 3 7

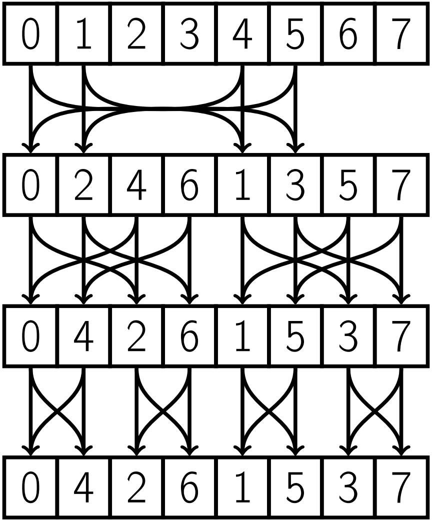

0 1 2 3 4 5 6 7
0 2 4 6 1 3 5 7
0 4 2 6 1 5 3 7
0 4 2 6 1 5 3 7

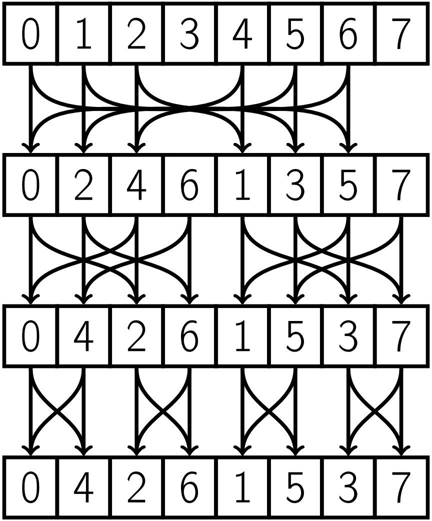

0 1 2 3 4 5 6 7
0 2 4 6 1 3 5 7
0 4 2 6 1 5 3 7
0 4 2 6 1 5 3 7

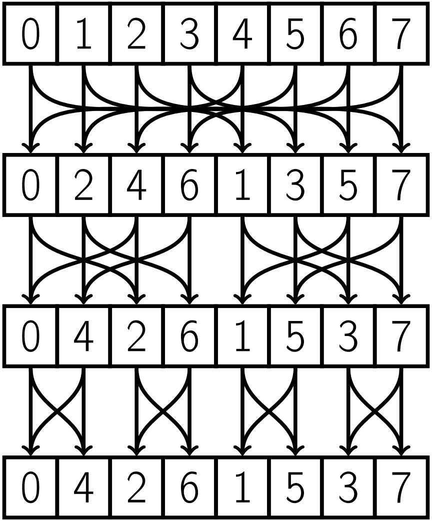

0 1 2 3 4 5 6 7
0 2 4 6 1 3 5 7
0 4 2 6 1 5 3 7
0 4 2 6 1 5 3 7


0 1 2 3 4 5 6 7
0 2 4 6 1 3 5 7
0 4 2 6 1 5 3 7
0 4 2 6 1 5 3 7


0 1 2 3 4 5 6 7
0 2 4 6 1 3 5 7
0 4 2 6 1 5 3 7
0 4 2 6 1 5 3 7


0 1 2 3 4 5 6 7
0 2 4 6 1 3 5 7
0 4 2 6 1 5 3 7
0 4 2 6 1 5 3 7


0 1 2 3 4 5 6 7
0 2 4 6 1 3 5 7
0 4 2 6 1 5 3 7
0 4 2 6 1 5 3 7


0 1 2 3 4 5 6 7
0 2 4 6 1 3 5 7
0 4 2 6 1 5 3 7
0 4 2 6 1 5 3 7

Final Order

$$\begin{pmatrix} 0 \\ 4 \\ 2 \\ 6 \\ 1 \\ 5 \\ 3 \\ 7 \end{pmatrix} = \begin{pmatrix} 000_2 \\ 100_2 \\ 010_2 \\ 110_2 \\ 001_2 \\ 101_2 \\ 011_2 \\ 111_2 \end{pmatrix} = \begin{pmatrix} 000^r \\ 001^r \\ 010^r \\ 011^r \\ 100^r \\ 101^r \\ 110^r \\ 111^r \end{pmatrix} = BitRev \begin{pmatrix} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{pmatrix}$$

$\mathtt{rev}_k(a)$ — reverse a as k-bit binary number

$$\mathtt{rev}_k(\overline{a0}) = \overline{0\mathtt{rev}_{k-1}(a)} = \mathtt{rev}_{k-1}(a)$$

$$\mathtt{rev}_k(\overline{a1}) = \overline{1\mathtt{rev}_{k-1}(a)} = 2^{k-1} + \mathtt{rev}_{k-1}(a)$$

```
RevBits(k)
```

$a[0] \leftarrow 0$
for i from 1 to k:
  for j from $2^k - 1$ down to 0:
    $a[j] \leftarrow a[\lfloor \frac{j}{2} \rfloor] + 2^{k-1} \cdot \texttt{is\_odd}(j)$
return a

$O(2^k)$

`PrepareFFT(`P, *invert*, k, ε, *roots*`)`

$k \leftarrow 1$
`while` $2^k \leq \deg(P)$:
  $k \leftarrow k + 1$
`P.resize(`2^k`)`
$r \leftarrow RevBits(k)$
`for` i `from` 0 `to` $2^k - 1$:
  `if` $i < r[i]$:
    `swap(`$P[i], P[r[i]]$`)`
$\varepsilon \leftarrow \cos(\frac{2\pi}{2^k}) + i \sin(\frac{2\pi}{2^k})$
`if` *invert*:
  $\varepsilon \leftarrow \cos(\frac{2\pi}{2^k}) - i \sin(\frac{2\pi}{2^k})$
$roots[k] \leftarrow \varepsilon$
`for` i `from` $k - 1$ `down to` 1:
  $roots[i] \leftarrow roots[i+1] \cdot roots[i+1]$

PrepareFFT(P, $invert$, k, ε, $roots$)

```
k ← 1
while 2^k ≤ deg(P):
  k ← k + 1
P.resize(2^k)
r ← RevBits(k)
for i from 0 to 2^k − 1:
  if i < r[i]:
    swap(P[i], P[r[i]])
ε ← cos(2π/2^k) + i sin(2π/2^k)
if invert:
  ε ← cos(2π/2^k) − i sin(2π/2^k)
roots[k] ← ε
for i from k − 1 down to 1:
  roots[i] ← roots[i + 1] · roots[i + 1]
```

PrepareFFT(P, $invert$, k, ε, $roots$)

```
k ← 1
while 2^k ≤ deg(P):
  k ← k + 1
P.resize(2^k)
r ← RevBits(k)
for i from 0 to 2^k − 1:
  if i < r[i]:
    swap(P[i], P[r[i]])
ε ← cos(2π/2^k) + i sin(2π/2^k)
if invert:
  ε ← cos(2π/2^k) − i sin(2π/2^k)
roots[k] ← ε
for i from k − 1 down to 1:
  roots[i] ← roots[i + 1] · roots[i + 1]
```

PrepareFFT(P, $invert$, k, ε, $roots$)

```
k ← 1
while 2^k ≤ deg(P):
  k ← k + 1
P.resize(2^k)
r ← RevBits(k)
for i from 0 to 2^k − 1:
  if i < r[i]:
    swap(P[i], P[r[i]])
ε ← cos(2π/2^k) + i sin(2π/2^k)
if invert:
  ε ← cos(2π/2^k) − i sin(2π/2^k)
roots[k] ← ε
for i from k − 1 down to 1:
  roots[i] ← roots[i + 1] · roots[i + 1]
```

PrepareFFT(P, *invert*, k, ε, *roots*)

```
k ← 1
while 2^k ≤ deg(P):
  k ← k + 1
P.resize(2^k)
r ← RevBits(k)
for i from 0 to 2^k − 1:
  if i < r[i]:
    swap(P[i], P[r[i]])
ε ← cos(2π/2^k) + i sin(2π/2^k)
if invert:
  ε ← cos(2π/2^k) − i sin(2π/2^k)
roots[k] ← ε
for i from k − 1 down to 1:
  roots[i] ← roots[i + 1] · roots[i + 1]
```

PrepareFFT(P, *invert*, k, ε, *roots*)

$k \leftarrow 1$
`while` $2^k \leq \deg(P)$:
  $k \leftarrow k + 1$
`P.resize(`2^k`)`
$r \leftarrow RevBits(k)$
`for` i `from 0 to` $2^k - 1$:
  `if` $i < r[i]$:
    `swap(`$P[i], P[r[i]]$`)`
$\varepsilon \leftarrow \cos(\frac{2\pi}{2^k}) + i\sin(\frac{2\pi}{2^k})$
`if` *invert*:
  $\varepsilon \leftarrow \cos(\frac{2\pi}{2^k}) - i\sin(\frac{2\pi}{2^k})$
$roots[k] \leftarrow \varepsilon$
`for` i `from` $k-1$ `down to 1`:
  $roots[i] \leftarrow roots[i+1] \cdot roots[i+1]$

PrepareFFT(P, $invert$, k, ε, $roots$)

$k \leftarrow 1$
while $2^k \leq \deg(P)$:
  $k \leftarrow k + 1$
P.resize(2^k)
$r \leftarrow RevBits(k)$
for i from 0 to $2^k - 1$:
  if $i < r[i]$:
    swap($P[i], P[r[i]]$)
$\varepsilon \leftarrow \cos(\frac{2\pi}{2^k}) + i\sin(\frac{2\pi}{2^k})$
if $invert$:
  $\varepsilon \leftarrow \cos(\frac{2\pi}{2^k}) - i\sin(\frac{2\pi}{2^k})$
$roots[k] \leftarrow \varepsilon$
for i from $k - 1$ down to 1:
  $roots[i] \leftarrow roots[i+1] \cdot roots[i+1]$

0	1	2	3	4	5	6	7

0	1	2	3	4	5	6	7

0	2	4	6	1	3	5	7

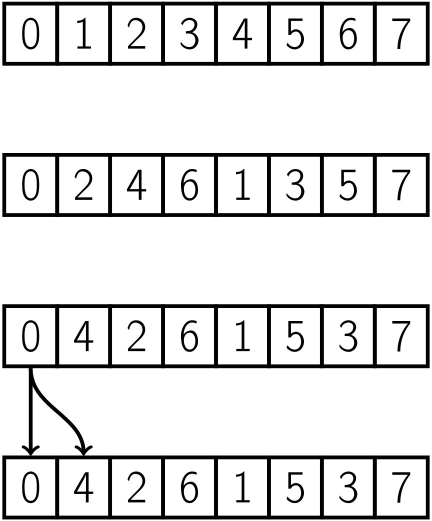
0	1	2	3	4	5	6	7
0	2	4	6	1	3	5	7
0	4	2	6	1	5	3	7

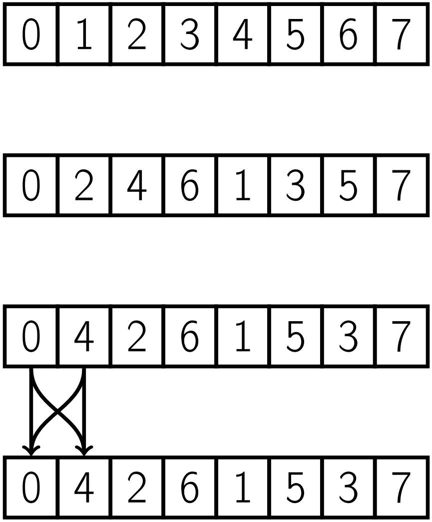
0	1	2	3	4	5	6	7

0	2	4	6	1	3	5	7

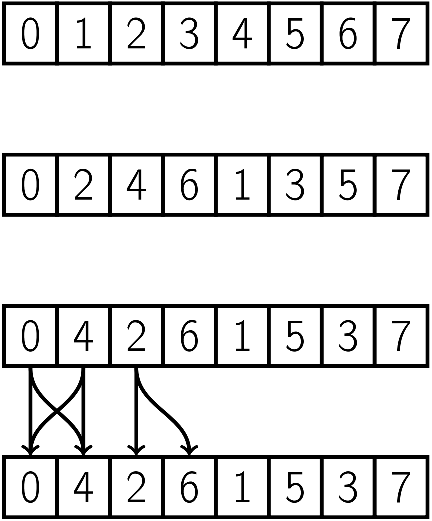
0	4	2	6	1	5	3	7

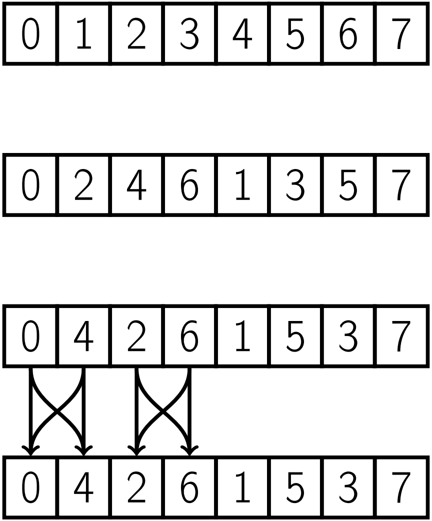
0	4	2	6	1	5	3	7

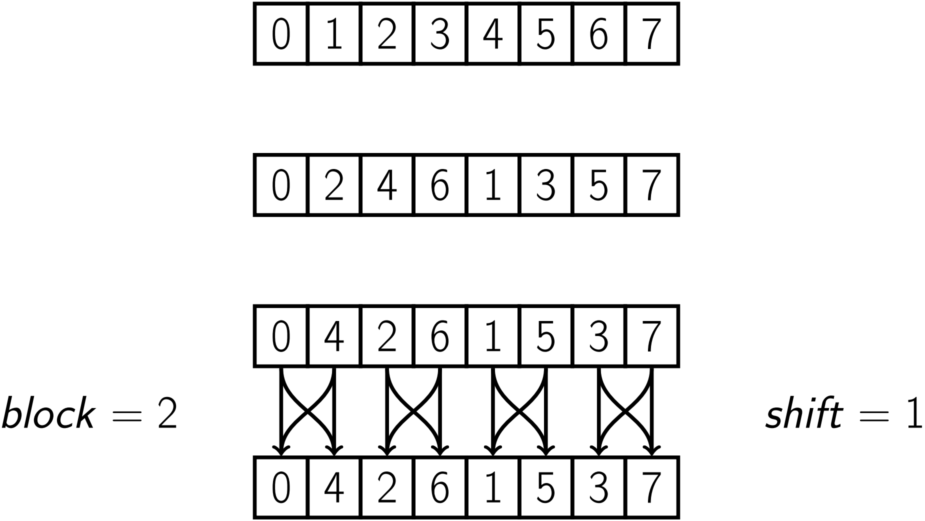

0 1 2 3 4 5 6 7
0 2 4 6 1 3 5 7
0 4 2 6 1 5 3 7
0 4 2 6 1 5 3 7



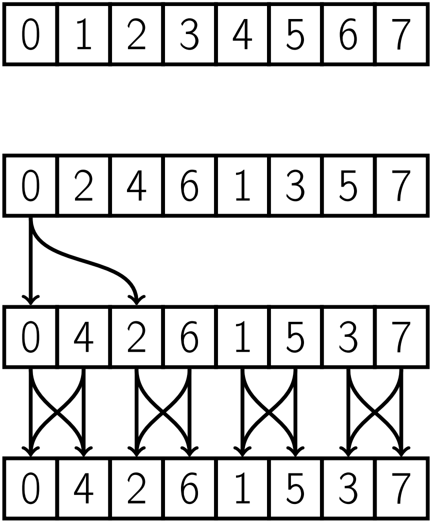
0 1 2 3 4 5 6 7
0 2 4 6 1 3 5 7
0 4 2 6 1 5 3 7
0 4 2 6 1 5 3 7


0 1 2 3 4 5 6 7
0 2 4 6 1 3 5 7
0 4 2 6 1 5 3 7
0 4 2 6 1 5 3 7

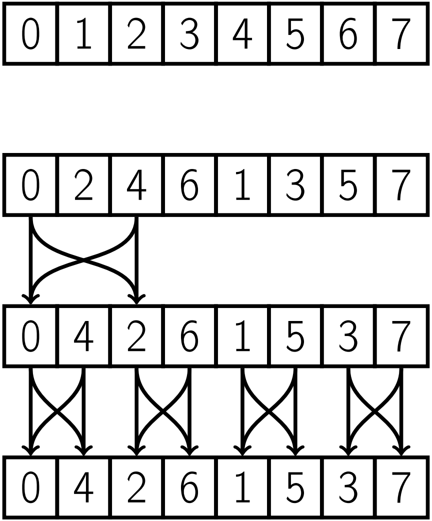

0 1 2 3 4 5 6 7
0 2 4 6 1 3 5 7
0 4 2 6 1 5 3 7
0 4 2 6 1 5 3 7


0 1 2 3 4 5 6 7
0 2 4 6 1 3 5 7
0 4 2 6 1 5 3 7
block = 2
shift = 1
0 4 2 6 1 5 3 7

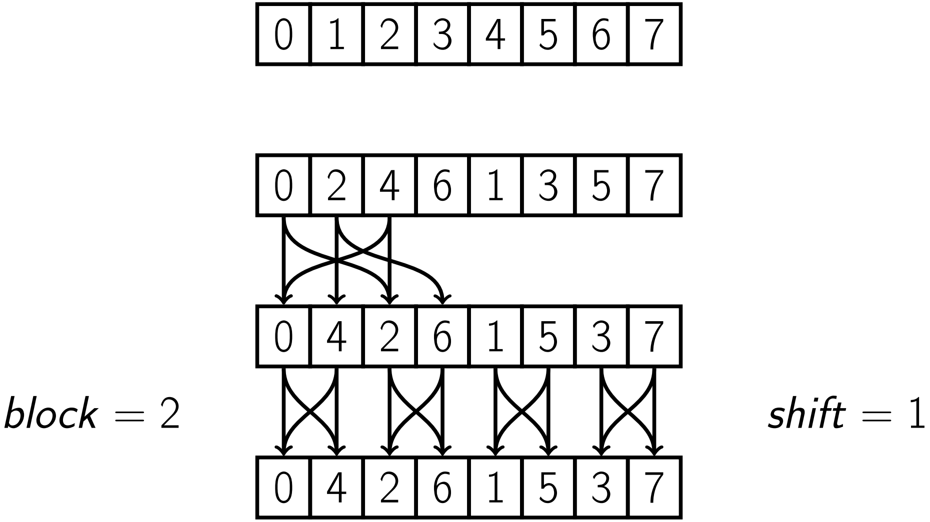
block $= 2$

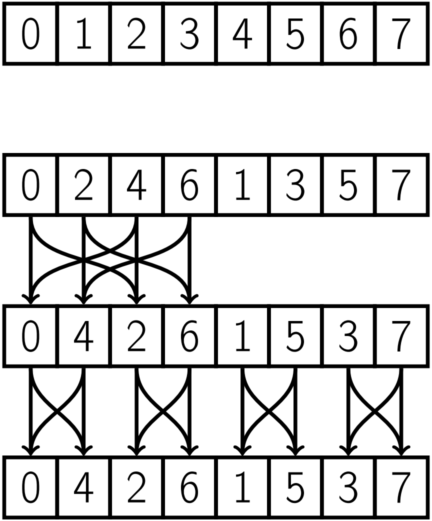


shift $= 1$

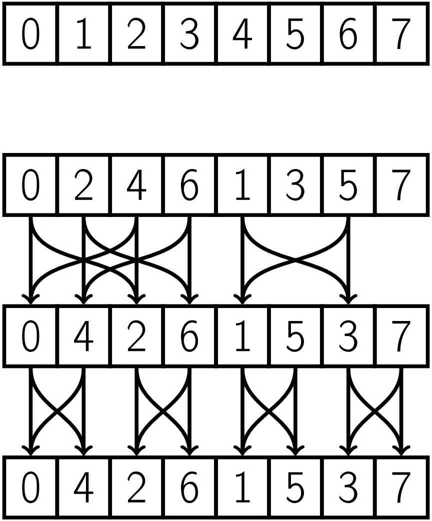


$block = 2$ $shift = 1$

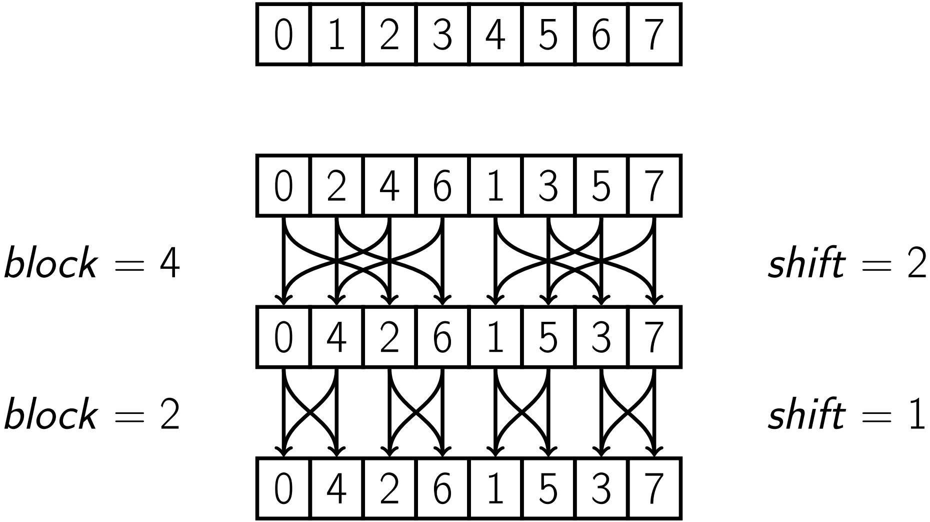

0 1 2 3 4 5 6 7
0 2 4 6 1 3 5 7
0 4 2 6 1 5 3 7
block = 2
shift = 1
0 4 2 6 1 5 3 7

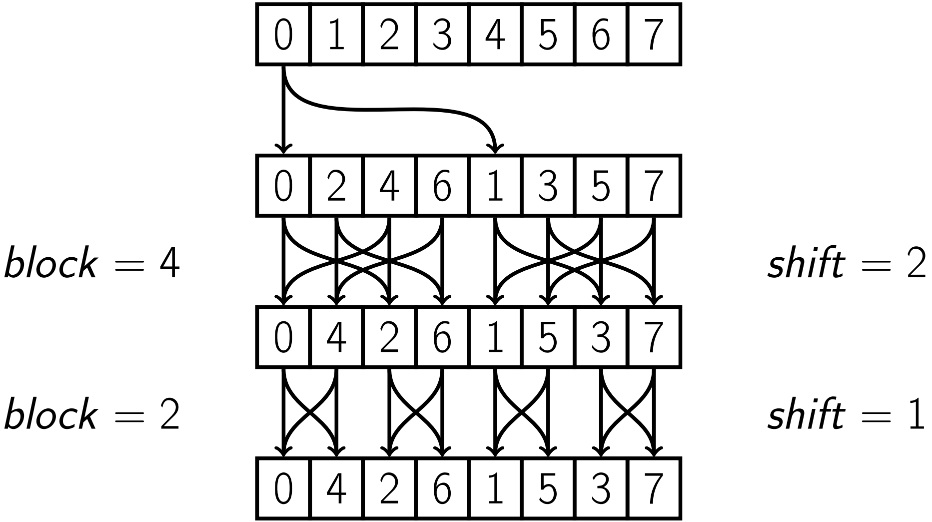


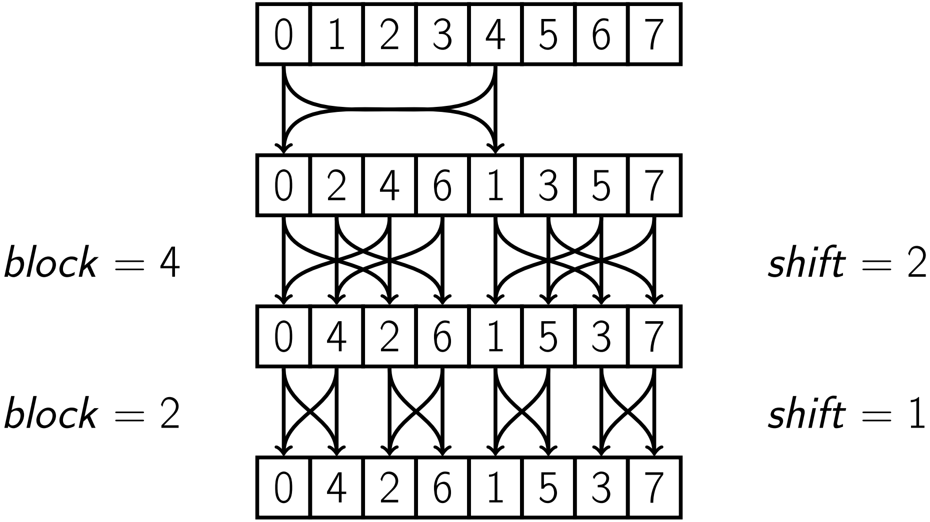
$block = 2$ $shift = 1$

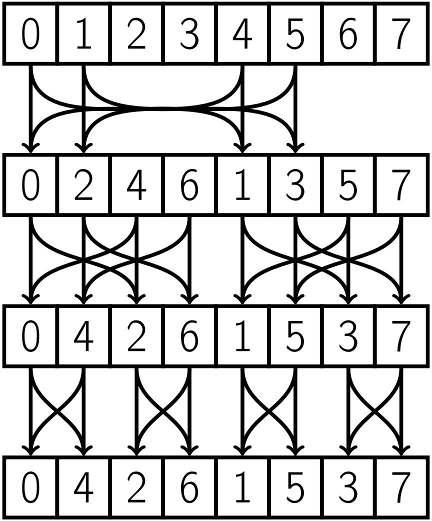


$block = 2$ $shift = 1$


0 1 2 3 4 5 6 7
0 2 4 6 1 3 5 7
block = 4
shift = 2
0 4 2 6 1 5 3 7
block = 2
shift = 1
0 4 2 6 1 5 3 7

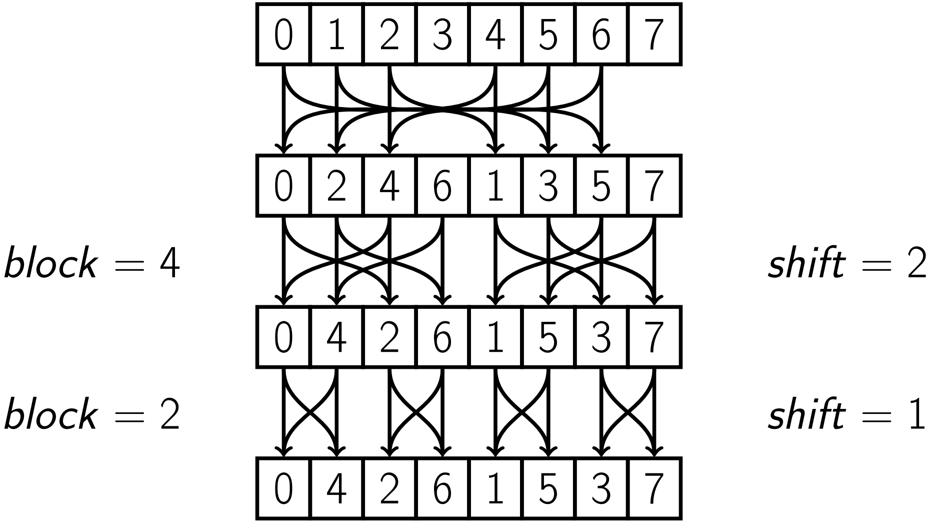

0 1 2 3 4 5 6 7
0 2 4 6 1 3 5 7
block = 4
shift = 2
0 4 2 6 1 5 3 7
block = 2
shift = 1
0 4 2 6 1 5 3 7

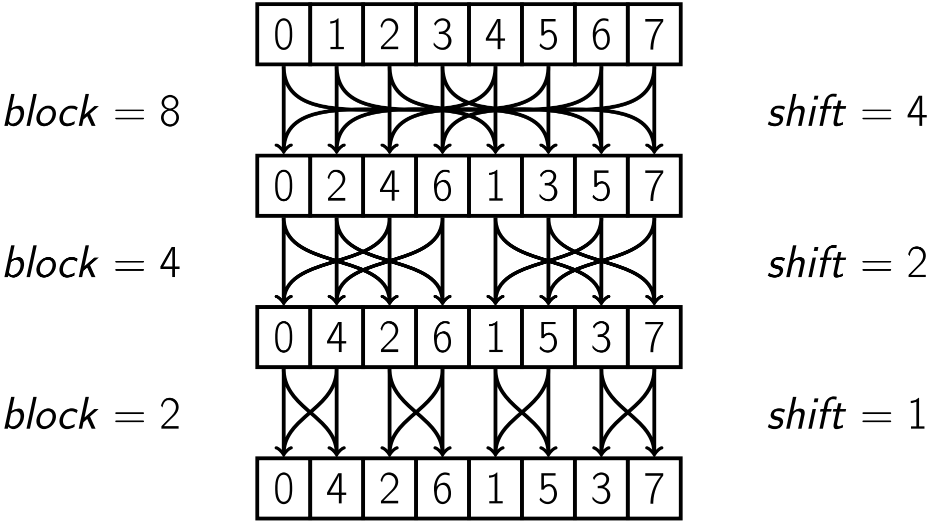

0 1 2 3 4 5 6 7
0 2 4 6 1 3 5 7
block = 4
shift = 2
0 4 2 6 1 5 3 7
block = 2
shift = 1
0 4 2 6 1 5 3 7



$block = 4$ $shift = 2$

$block = 2$ $shift = 1$


0 1 2 3 4 5 6 7
0 2 4 6 1 3 5 7
block = 4
shift = 2
0 4 2 6 1 5 3 7
block = 2
shift = 1
0 4 2 6 1 5 3 7


block = 8
block = 4
block = 2
shift = 4
shift = 2
shift = 1
0 1 2 3 4 5 6 7
0 2 4 6 1 3 5 7
0 4 2 6 1 5 3 7
0 4 2 6 1 5 3 7

$\texttt{FFT}(P, invert)$

```
PrepareFFT(P, invert, k, ε, roots)
for level from 1 to k:
  block ← 2^level
  for start from 0 to 2^k with step block:
    w ← 1
    shift ← block/2
    for i from start to start + shift − 1:
      l ← P[i] + w · P[i + shift]
      r ← P[i] − w · P[i + shift]
      P[i] ← l
      P[i + shift] ← r
      w ← w · roots[level]
if invert:
  for i from 0 to 2^k − 1:
    P[i] ← P[i]/n
return P
```

$\texttt{FFT}(P, invert)$

```
PrepareFFT(P, invert, k, ε, roots)
for level from 1 to k:
  block ← 2^level
  for start from 0 to 2^k with step block:
    w ← 1
    shift ← block/2
    for i from start to start + shift − 1:
      l ← P[i] + w · P[i + shift]
      r ← P[i] − w · P[i + shift]
      P[i] ← l
      P[i + shift] ← r
      w ← w · roots[level]
if invert:
  for i from 0 to 2^k − 1:
    P[i] ← P[i]/n
return P
```

FFT(P, $invert$)

```
PrepareFFT(P, invert, k, ε, roots)
for level from 1 to k:
  block ← 2^level
  for start from 0 to 2^k with step block:
    w ← 1
    shift ← block/2
    for i from start to start + shift − 1:
      l ← P[i] + w · P[i + shift]
      r ← P[i] − w · P[i + shift]
      P[i] ← l
      P[i + shift] ← r
      w ← w · roots[level]
if invert:
  for i from 0 to 2^k − 1:
    P[i] ← P[i]/n
return P
```

FFT(P, *invert*)

```
PrepareFFT(P, invert, k, ε, roots)
for level from 1 to k:
  block ← 2^level
  for start from 0 to 2^k with step block:
    w ← 1
    shift ← block/2
    for i from start to start + shift − 1:
      l ← P[i] + w · P[i + shift]
      r ← P[i] − w · P[i + shift]
      P[i] ← l
      P[i + shift] ← r
      w ← w · roots[level]
if invert:
  for i from 0 to 2^k − 1:
    P[i] ← P[i]/n
return P
```

FFT(P, $invert$)

```
PrepareFFT(P, invert, k, ε, roots)
for level from 1 to k:
  block ← 2^level
  for start from 0 to 2^k with step block:
    w ← 1
    shift ← block/2
    for i from start to start + shift − 1:
      l ← P[i] + w · P[i + shift]
      r ← P[i] − w · P[i + shift]
      P[i] ← l
      P[i + shift] ← r
      w ← w · roots[level]
if invert:
  for i from 0 to 2^k − 1:
    P[i] ← P[i]/n
return P
```

$\texttt{FFT}(P, \mathit{invert})$

```
PrepareFFT(P, invert, k, ε, roots)
for level from 1 to k:
  block ← 2^level
  for start from 0 to 2^k with step block:
    w ← 1
    shift ← block/2
    for i from start to start + shift − 1:
      l ← P[i] + w · P[i + shift]
      r ← P[i] − w · P[i + shift]
      P[i] ← l
      P[i + shift] ← r
      w ← w · roots[level]
if invert:
  for i from 0 to 2^k − 1:
    P[i] ← P[i]/n
return P
```

FFT(P, $invert$)

```
PrepareFFT(P, invert, k, ε, roots)
for level from 1 to k:
  block ← 2^level
  for start from 0 to 2^k with step block:
    w ← 1
    shift ← block/2
    for i from start to start + shift − 1:
      l ← P[i] + w · P[i + shift]
      r ← P[i] − w · P[i + shift]
      P[i] ← l
      P[i + shift] ← r
      w ← w · roots[level]
if invert:
  for i from 0 to 2^k − 1:
    P[i] ← P[i]/n
return P
```

`FFT`(P, *invert*)

```
PrepareFFT(P, invert, k, ε, roots)
for level from 1 to k:
  block ← 2^level
  for start from 0 to 2^k with step block:
    w ← 1
    shift ← block/2
    for i from start to start + shift − 1:
      l ← P[i] + w · P[i + shift]
      r ← P[i] − w · P[i + shift]
      P[i] ← l
      P[i + shift] ← r
      w ← w · roots[level]
if invert:
  for i from 0 to 2^k − 1:
    P[i] ← P[i]/n
return P
```

Example

Input: Integer n

Output: For every $i = 1..n$, the number of ways to represent i as sum of two squares $i = a^2 + b^2$.

Naive Solution

- For every i, try every $a^2 \leq \frac{i}{2}$ as the smaller square

Naive Solution

- For every i, try every $a^2 \leq \frac{i}{2}$ as the smaller square
- Test whether $i - a^2$ is a square

Naive Solution

- For every i, try every $a^2 \leq \frac{i}{2}$ as the smaller square
- Test whether $i - a^2$ is a square
- $O(n\sqrt{n})$

Solution with FFT

- Create polynomial $P(x) = \sum_{i=0}^{n} p_i x^i$, such that $p_{a^2} = 1$ for all $a \leq \sqrt{n}$, and all other coefficients are 0

Solution with FFT

- Create polynomial $P(x) = \sum_{i=0}^{n} p_i x^i$, such that $p_{a^2} = 1$ for all $a \leq \sqrt{n}$, and all other coefficients are 0
- Compute $Q = P^2(x)$ using FFT

Solution with FFT

- Create polynomial $P(x) = \sum_{i=0}^{n} p_i x^i$, such that $p_{a^2} = 1$ for all $a \le \sqrt{n}$, and all other coefficients are 0
- Compute $Q = P^2(x)$ using FFT
- Coefficients $(Q[x^i])_{i=1}^{n}$ are the answers

Solution with FFT

- Create polynomial $P(x) = \sum_{i=0}^{n} p_i x^i$, such that $p_{a^2} = 1$ for all $a \leq \sqrt{n}$, and all other coefficients are 0
- Compute $Q = P^2(x)$ using FFT
- Coefficients $(Q[x^i])_{i=1}^{n}$ are the answers
- $O(n \log n)$

Solution with FFT

- Create polynomial $P(x) = \sum_{i=0}^{n} p_i x^i$, such that $p_{a^2} = 1$ for all $a \le \sqrt{n}$, and all other coefficients are 0
- Compute $Q = P^2(x)$ using FFT
- Coefficients $(Q[x^i])_{i=1}^{n}$ are the answers
- $O(n \log n)$
- What if we needed sums of 4 squares?

Precision Issues

Theorem

$$\|fl(y) - y\|_\infty \,/\, \|y\|_2 < C\sqrt{n}\log_2 n\varepsilon + O(\varepsilon^2),$$

where ε is machine precision,

$\|x\|_\infty = \max\limits_{i=1..n} |x_i|$ — L_∞-norm,

$\|x\|_2 = \sqrt{\frac{1}{n}\sum\limits_{i=1}^{n} x_i^2}$ — L_2-norm, C — some constant.

In practical tests, precision is around 100 times more precise.

Precision Issues

For example, if multiplying two polynomials of degree $n = 10^6$ with coefficients up to 10^9, with double precision $\varepsilon = 10^{-15}$, the error in the final coefficients of the product is below $C \cdot \sqrt{10^6} \cdot 10^9 \cdot \log_2 10^6 \varepsilon < 10 \cdot 10^3 \cdot 10^9 \cdot 20 \cdot 10^{-15} < 2 \cdot 10^{14} \cdot 10^{-15} = 2 \cdot 10^{-1} < 0.5$, so the result will be precise after rounding to the closest integer.

Precision Issues

Instead of using $\varepsilon = e^{\frac{2\pi i}{2^k}}$, we can choose a huge prime p of the form $p = a \cdot 2^k + 1$, choose ε as the primitive root modulo p and perform all computations modulo p.

"Two in One" Trick

If we only need FFT for real-valued vectors, we can do two FFTs at once.

$P(x), Q(x) \in \mathbb{R}[x]$

Take $R(x) = P(x) + iQ(x)$

Then
$Re(P(\varepsilon^j)) = \frac{1}{2} Re(R(\varepsilon^j) + R(\varepsilon^{2^k - j}))$,
$Im(P(\varepsilon^j)) = \frac{1}{2} Im(R(\varepsilon^j) - R(\varepsilon^{2^k - j}))$, and symmetrically with $Q(\varepsilon^j)$.