CS 170: Algorithms

CS 170: Algorithms

S

CS 170: Algorithms

H

S

CS 170: Algorithms

H

H

S

CS 170: Algorithms

H

H H

S

CS 170: Algorithms

H

H H

S H

CS 170: Algorithms

H

H H

S H

CS 170: Algorithms

H

H H

S H

CS 170: Algorithms

H

H H

S H

CS 170: Algorithms

H

H H

S H

CS 170: Algorithms

H

H H

S H

CS 170: Algorithms

H

H H

S H

No laptops please.

CS 170: Algorithms

H

H H

S H

No laptops please.

Thank you

CS 170: Algorithms

H

H H

S H

No laptops please.

Thank you !

CS 170: Algorithms

H

H H

S H

No laptops please.

Thank you ! !

CS 170: Algorithms

H

H H

S H

No laptops please.

Thank you ! ! !

CS 170: Algorithms

H

H H

S H

No laptops please.

Thank you ! ! ! !

CS 170: Algorithms

H

H H

S H

No laptops please.

Thank you ! ! ! ! !

CS 170: Algorithms

H

H H

S H

No laptops please.

Thank you ! ! ! ! ! !

CS 170: Algorithms

H

H H

S H

No laptops please.

Thank you ! ! ! ! ! ! !

CS 170: Algorithms

H

H H

S H

No laptops please.

Thank you ! ! ! ! ! ! ! !

CS 170: Algorithms

H

H H

S H

No laptops please.

Thank you ! ! ! ! ! ! ! !

Today:

CS 170: Algorithms

H

H H

S H

No laptops please.

Thank you ! ! ! ! ! ! ! !

Today:

Review Inverse FFT

CS 170: Algorithms

H

H H

S H

No laptops please.

Thank you ! ! ! ! ! ! ! !

Today:

Review Inverse FFT

FFT network.

CS 170: Algorithms

H

H H

S H

No laptops please.

Thank you ! ! ! ! ! ! ! !

Today:

Review Inverse FFT

FFT network.

Other parallel systems.

Multiplying polynomials?

Coefficient representation:
$a_0, a_1, \ldots, a_{n-1}$

Value representation:
$A(x_0), A(x_1), \ldots, A(x_{n-1})$.

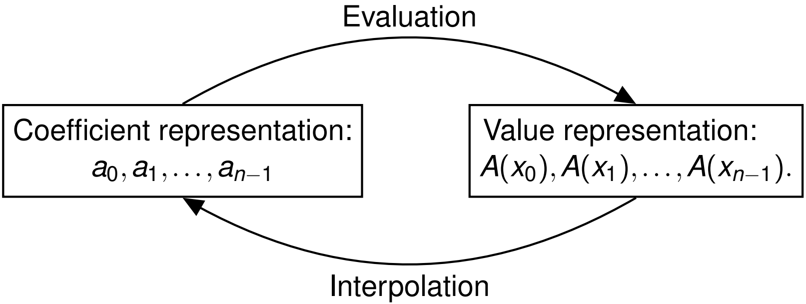
Multiplying polynomials?

Evaluation: $O(n \log n)$ if choose $1, \omega, \omega^2, \ldots, \omega^{n-1}$.

Coefficient representation: $a_0, a_1, \ldots, a_{n-1}$ $\xrightarrow{\text{Evaluation}}$ Value representation: $A(x_0), A(x_1), \ldots, A(x_{n-1})$.

Multiplying polynomials?

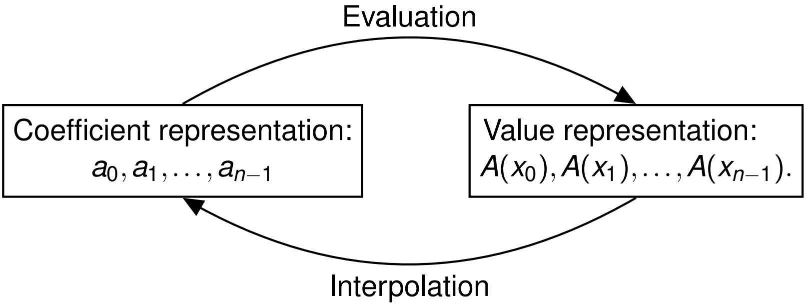
Evaluation: $O(n \log n)$ if choose $1, \omega, \omega^2, \ldots, \omega^{n-1}$.



Interpolation: From points $A(x_0), \ldots, A(x_{n-1})$ to "function".

Multiplying polynomials?

Evaluation: $O(n \log n)$ if choose $1, \omega, \omega^2, \ldots, \omega^{n-1}$.



Interpolation: From points $A(x_0), \ldots, A(x_{n-1})$ to "function".
How?

Polynomial Evaluation and Matrices

Compute $A(\cdot)$ from a_i's:

$$\begin{bmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots x_1^{n-1} \\ & & \vdots & \cdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

Compute inverse of matrix above.

Polynomial Evaluation and Matrices

Compute $A(\cdot)$ from a_i's:

$$\begin{bmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots x_1^{n-1} \\ & & \vdots & \cdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

Compute inverse of matrix above.
Multiply.

Polynomial Evaluation and Matrices

Compute $A(\cdot)$ from a_i's:

$$\begin{bmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots x_1^{n-1} \\ & & \vdots & \cdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

Compute inverse of matrix above.
Multiply. $O(n^2)$!

Polynomial Evaluation and Matrices

Compute $A(\cdot)$ from a_i's:

$$\begin{bmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots x_1^{n-1} \\ & & \vdots & \cdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

Compute inverse of matrix above.
Multiply. $O(n^2)$!

Doh!!

Polynomial Evaluation and Matrices

Compute $A(\cdot)$ from a_i's:

$$\begin{bmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots x_1^{n-1} \\ & & \vdots & \cdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

Compute inverse of matrix above.
Multiply. $O(n^2)$!

Doh!!

Also, computing inverse not even easy.

FFT Matrix

FFT: ω is complex nth root of unity

FFT Matrix

FFT: ω is complex nth root of unity
and matrix is ...

$$M_n(\omega) = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{(n-1)} \\ & & & & \\ & & & & \\ & & & & \\ & & & & \end{bmatrix}$$

FFT Matrix

FFT: ω is complex nth root of unity
and matrix is ...

$$M_n(\omega) = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{(n-1)} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ & & & & \\ & & & & \\ & & & & \end{bmatrix}$$

FFT Matrix

FFT: ω is complex nth root of unity
and matrix is ...

$$M_n(\omega) = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{(n-1)} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & & \vdots & & \\ 1 & \omega^j & \omega^{2j} & \cdots & \omega^{j(n-1)} \\ & & & & \\ & & & & \end{bmatrix}$$

FFT Matrix

FFT: ω is complex *n*th root of unity
and matrix is ...

$$M_n(\omega) = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{(n-1)} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & & \vdots & & \\ 1 & \omega^j & \omega^{2j} & \cdots & \omega^{j(n-1)} \\ \vdots & & \vdots & & \\ 1 & \omega^{(n-1)} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{bmatrix}$$

FFT Matrix

FFT: ω is complex nth root of unity
and matrix is ...

$$M_n(\omega) = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{(n-1)} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ \vdots & & \vdots & & \\ 1 & \omega^j & \omega^{2j} & \cdots & \omega^{j(n-1)} \\ \vdots & & \vdots & & \\ 1 & \omega^{(n-1)} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{bmatrix}$$

Compute inverse of $M_n(\omega)$?

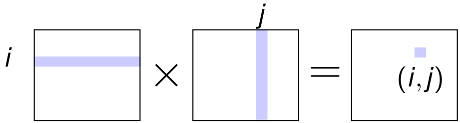
Algebraically.

Inversion formula: $(M_n(\omega))^{-1} = \frac{1}{n} M_n(\omega^{-1})$.

Algebraically.

Inversion formula: $(M_n(\omega))^{-1} = \frac{1}{n} M_n(\omega^{-1})$.

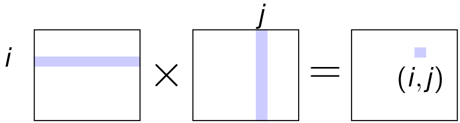
$C = M_n(\omega) \times M_n(\omega^{-1})$?



Algebraically.

Inversion formula: $(M_n(\omega))^{-1} = \frac{1}{n} M_n(\omega^{-1})$.

$C = M_n(\omega) \times M_n(\omega^{-1})$?



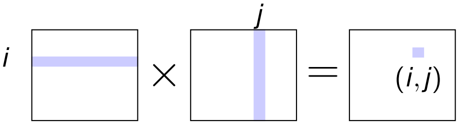
Recall: $\omega = e^{2\pi/n}$.

$$c_{ij} = \sum_k \omega^{ik} \omega^{-kj}$$

Algebraically.

Inversion formula: $(M_n(\omega))^{-1} = \frac{1}{n} M_n(\omega^{-1})$.

$$C = M_n(\omega) \times M_n(\omega^{-1})?$$



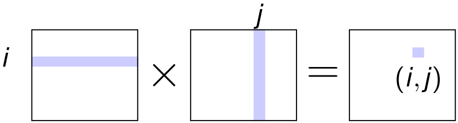
Recall: $\omega = e^{2\pi/n}$.

$$c_{ij} = \sum_k \omega^{ik} \omega^{-kj} = \sum_k \omega^{(ik-kj)}$$

Algebraically.

Inversion formula: $(M_n(\omega))^{-1} = \frac{1}{n} M_n(\omega^{-1})$.

$C = M_n(\omega) \times M_n(\omega^{-1})$?



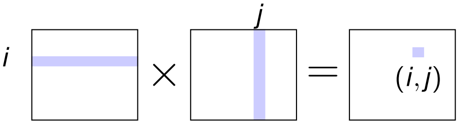
Recall: $\omega = e^{2\pi/n}$.

$$c_{ij} = \sum_k \omega^{ik}\omega^{-kj} = \sum_k \omega^{(ik-kj)} = \sum_k \omega^{k(i-j)}$$

Algebraically.

Inversion formula: $(M_n(\omega))^{-1} = \frac{1}{n} M_n(\omega^{-1})$.

$$C = M_n(\omega) \times M_n(\omega^{-1})?$$



Recall: $\omega = e^{2\pi/n}$.

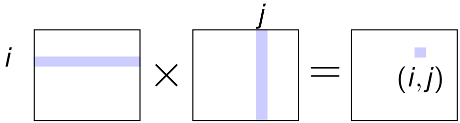
$$c_{ij} = \sum_k \omega^{ik}\omega^{-kj} = \sum_k \omega^{(ik-kj)} = \sum_k \omega^{k(i-j)}$$

If $i \neq j$, let $r = \omega^{(i-j)}$ (note: $r^n = 1$)

Algebraically.

Inversion formula: $(M_n(\omega))^{-1} = \frac{1}{n} M_n(\omega^{-1})$.

$C = M_n(\omega) \times M_n(\omega^{-1})$?



Recall: $\omega = e^{2\pi/n}$.

$$c_{ij} \quad = \quad \sum_k \omega^{ik} \omega^{-kj} = \sum_k \omega^{(ik-kj)} = \sum_k \omega^{k(i-j)}$$

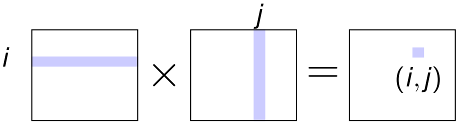
If $i \neq j$, let $r = \omega^{(i-j)}$ (note: $r^n = 1$)

$c_{ij} = 1 + r + r^2 + \cdots + r^{n-1}$

Algebraically.

Inversion formula: $(M_n(\omega))^{-1} = \frac{1}{n} M_n(\omega^{-1})$.

$$C = M_n(\omega) \times M_n(\omega^{-1})?$$



Recall: $\omega = e^{2\pi/n}$.

$$c_{ij} \;=\; \sum_k \omega^{ik}\omega^{-kj} = \sum_k \omega^{(ik-kj)} = \sum_k \omega^{k(i-j)}$$

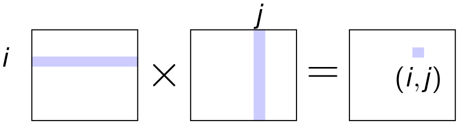
If $i \neq j$, let $r = \omega^{(i-j)}$ (note: $r^n = 1$)

$c_{ij} = 1 + r + r^2 + \cdots + r^{n-1} = \frac{1-r^n}{1-r}$

Algebraically.

Inversion formula: $(M_n(\omega))^{-1} = \frac{1}{n}M_n(\omega^{-1})$.

$$C = M_n(\omega) \times M_n(\omega^{-1})?$$



Recall: $\omega = e^{2\pi/n}$.

$$c_{ij} \quad = \quad \sum_k \omega^{ik}\omega^{-kj} = \sum_k \omega^{(ik-kj)} = \sum_k \omega^{k(i-j)}$$

If $i \neq j$, let $r = \omega^{(i-j)}$ (note: $r^n = 1$)

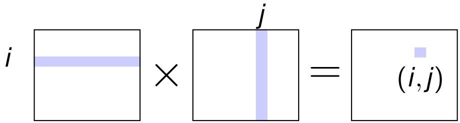
$c_{ij} = 1 + r + r^2 + \cdots + r^{n-1} = \frac{1-r^n}{1-r}$

Case $i = j$:

Algebraically.

Inversion formula: $(M_n(\omega))^{-1} = \frac{1}{n} M_n(\omega^{-1})$.

$C = M_n(\omega) \times M_n(\omega^{-1})$?



Recall: $\omega = e^{2\pi/n}$.

$$c_{ij} \;=\; \sum_k \omega^{ik}\omega^{-kj} = \sum_k \omega^{(ik-kj)} = \sum_k \omega^{k(i-j)}$$

If $i \neq j$, let $r = \omega^{(i-j)}$ (note: $r^n = 1$)

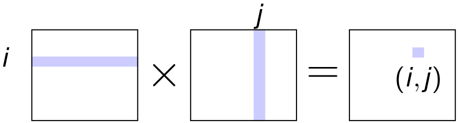
$c_{ij} = 1 + r + r^2 + \cdots + r^{n-1} = \frac{1-r^n}{1-r}$

Case $i = j$**:** $r = \omega^0 = 1$

Algebraically.

Inversion formula: $(M_n(\omega))^{-1} = \frac{1}{n} M_n(\omega^{-1})$.

$C = M_n(\omega) \times M_n(\omega^{-1})$?



Recall: $\omega = e^{2\pi/n}$.

$$c_{ij} = \sum_k \omega^{ik}\omega^{-kj} = \sum_k \omega^{(ik-kj)} = \sum_k \omega^{k(i-j)}$$

If $i \neq j$, let $r = \omega^{(i-j)}$ (note: $r^n = 1$)

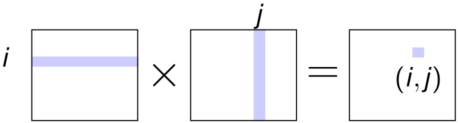
$c_{ij} = 1 + r + r^2 + \cdots + r^{n-1} = \frac{1-r^n}{1-r}$

Case $i = j$: $r = \omega^0 = 1$ and $c_{ii} = n$.

Algebraically.

Inversion formula: $(M_n(\omega))^{-1} = \frac{1}{n}M_n(\omega^{-1})$.

$C = M_n(\omega) \times M_n(\omega^{-1})$?



Recall: $\omega = e^{2\pi/n}$.

$$c_{ij} = \sum_k \omega^{ik}\omega^{-kj} = \sum_k \omega^{(ik-kj)} = \sum_k \omega^{k(i-j)}$$

If $i \neq j$, let $r = \omega^{(i-j)}$ (note: $r^n = 1$)

$c_{ij} = 1 + r + r^2 + \cdots + r^{n-1} = \frac{1-r^n}{1-r}$

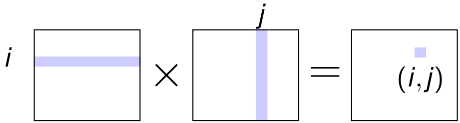
Case $i = j$: $r = \omega^0 = 1$ and $c_{ii} = n$.

Case $i \neq j$:

Algebraically.

Inversion formula: $(M_n(\omega))^{-1} = \frac{1}{n} M_n(\omega^{-1})$.

$C = M_n(\omega) \times M_n(\omega^{-1})$?



Recall: $\omega = e^{2\pi/n}$.

$$c_{ij} = \sum_k \omega^{ik}\omega^{-kj} = \sum_k \omega^{(ik-kj)} = \sum_k \omega^{k(i-j)}$$

If $i \neq j$, let $r = \omega^{(i-j)}$ (note: $r^n = 1$)

$c_{ij} = 1 + r + r^2 + \cdots + r^{n-1} = \frac{1-r^n}{1-r}$

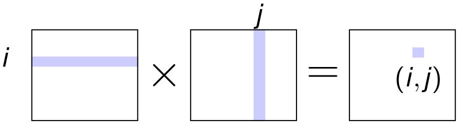
Case $i = j$**:** $r = \omega^0 = 1$ and $c_{ii} = n$.

Case $i \neq j$**:** $r^n = (\omega^{(i-j)})^n$

Algebraically.

Inversion formula: $(M_n(\omega))^{-1} = \frac{1}{n} M_n(\omega^{-1})$.

$C = M_n(\omega) \times M_n(\omega^{-1})$?



Recall: $\omega = e^{2\pi/n}$.

$$c_{ij} = \sum_k \omega^{ik}\omega^{-kj} = \sum_k \omega^{(ik-kj)} = \sum_k \omega^{k(i-j)}$$

If $i \neq j$, let $r = \omega^{(i-j)}$ (note: $r^n = 1$)

$c_{ij} = 1 + r + r^2 + \cdots + r^{n-1} = \frac{1-r^n}{1-r}$

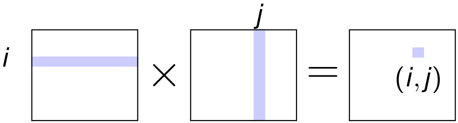
Case $i = j$**:** $r = \omega^0 = 1$ and $c_{ii} = n$.

Case $i \neq j$**:** $r^n = (\omega^{(i-j)})^n = (\omega^n)^{(i-j)} = 1^n$

Algebraically.

Inversion formula: $(M_n(\omega))^{-1} = \frac{1}{n} M_n(\omega^{-1})$.

$C = M_n(\omega) \times M_n(\omega^{-1})$?



Recall: $\omega = e^{2\pi/n}$.

$$c_{ij} = \sum_k \omega^{ik} \omega^{-kj} = \sum_k \omega^{(ik-kj)} = \sum_k \omega^{k(i-j)}$$

If $i \neq j$, let $r = \omega^{(i-j)}$ (note: $r^n = 1$)

$c_{ij} = 1 + r + r^2 + \cdots + r^{n-1} = \frac{1-r^n}{1-r}$

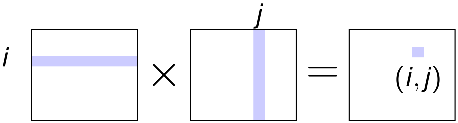
Case $i = j$**:** $r = \omega^0 = 1$ and $c_{ii} = n$.

Case $i \neq j$**:** $r^n = (\omega^{(i-j)})^n = (\omega^n)^{(i-j)} = 1^n \implies c_{ij} = 0$.

Algebraically.

Inversion formula: $(M_n(\omega))^{-1} = \frac{1}{n} M_n(\omega^{-1})$.

$$C = M_n(\omega) \times M_n(\omega^{-1})?$$



Recall: $\omega = e^{2\pi/n}$.

$$c_{ij} \quad = \quad \sum_k \omega^{ik}\omega^{-kj} = \sum_k \omega^{(ik-kj)} = \sum_k \omega^{k(i-j)}$$

If $i \neq j$, let $r = \omega^{(i-j)}$ (note: $r^n = 1$)

$c_{ij} = 1 + r + r^2 + \cdots + r^{n-1} = \frac{1-r^n}{1-r}$

Case $i = j$**:** $r = \omega^0 = 1$ and $c_{ii} = n$.

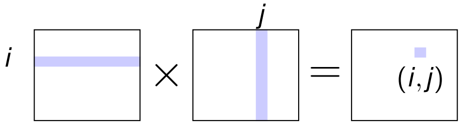
Case $i \neq j$**:** $r^n = (\omega^{(i-j)})^n = (\omega^n)^{(i-j)} = 1^n \implies c_{ij} = 0$.

For C – diagonals are n and the off-diagonals are 0.

Algebraically.

Inversion formula: $(M_n(\omega))^{-1} = \frac{1}{n} M_n(\omega^{-1})$.

$C = M_n(\omega) \times M_n(\omega^{-1})$?



Recall: $\omega = e^{2\pi/n}$.

$$c_{ij} \;=\; \sum_k \omega^{ik}\omega^{-kj} = \sum_k \omega^{(ik-kj)} = \sum_k \omega^{k(i-j)}$$

If $i \neq j$, let $r = \omega^{(i-j)}$ (note: $r^n = 1$)

$c_{ij} = 1 + r + r^2 + \cdots + r^{n-1} = \frac{1-r^n}{1-r}$

Case $i = j$**:** $r = \omega^0 = 1$ and $c_{ii} = n$.

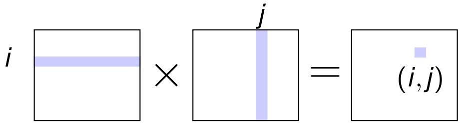
Case $i \neq j$**:** $r^n = (\omega^{(i-j)})^n = (\omega^n)^{(i-j)} = 1^n \implies c_{ij} = 0$.

For C – diagonals are n and the off-diagonals are 0.
Divide by n get identity!

Algebraically.

Inversion formula: $(M_n(\omega))^{-1} = \frac{1}{n}M_n(\omega^{-1})$.

$C = M_n(\omega) \times M_n(\omega^{-1})$?



Recall: $\omega = e^{2\pi/n}$.

$$c_{ij} \;=\; \sum_k \omega^{ik}\omega^{-kj} = \sum_k \omega^{(ik-kj)} = \sum_k \omega^{k(i-j)}$$

If $i \neq j$, let $r = \omega^{(i-j)}$ (note: $r^n = 1$)

$c_{ij} = 1 + r + r^2 + \cdots + r^{n-1} = \frac{1-r^n}{1-r}$

Case $i = j$**:** $r = \omega^0 = 1$ and $c_{ii} = n$.

Case $i \neq j$**:** $r^n = (\omega^{(i-j)})^n = (\omega^n)^{(i-j)} = 1^n \implies c_{ij} = 0$.

For C – diagonals are n and the off-diagonals are 0.
Divide by n get identity!

Inversion formula: $M_n(\omega)^{-1} = \frac{1}{n}M_n(\omega^{-1})$.

Computing inverse.

FFT works with points with basic root of unity: ω or ω^{-1}

Computing inverse.

FFT works with points with basic root of unity: ω or ω^{-1}

$1, \omega^{-1}, \omega^{-2}, \ldots, \omega^{-(n-1)}$.

Computing inverse.

FFT works with points with basic root of unity: ω or ω^{-1}

$1, \omega^{-1}, \omega^{-2}, \ldots, \omega^{-(n-1)}$.

ω^{-1} is a primitive *n*th root of unity!

Computing inverse.

FFT works with points with basic root of unity: ω or ω^{-1}

$1, \omega^{-1}, \omega^{-2}, \ldots, \omega^{-(n-1)}$.

ω^{-1} is a primitive nth root of unity!

Evaluation: FFT(A,ω).

Computing inverse.

FFT works with points with basic root of unity: ω or ω^{-1}

$1, \omega^{-1}, \omega^{-2}, \ldots, \omega^{-(n-1)}$.

ω^{-1} is a primitive *n*th root of unity!

Evaluation: FFT(A,ω).

Interpolation: $\frac{1}{n}$ FFT(A,ω^{-1}).

Computing inverse.

FFT works with points with basic root of unity: ω or ω^{-1}

$1, \omega^{-1}, \omega^{-2}, \ldots, \omega^{-(n-1)}$.

ω^{-1} is a primitive nth root of unity!

Evaluation: FFT(A,ω).
Interpolation: $\frac{1}{n}$ FFT(A,ω^{-1}).

$\implies O(n \log n)$ time for multiplying degree n polynomials.

Computing inverse.

FFT works with points with basic root of unity: ω or ω^{-1}

$1, \omega^{-1}, \omega^{-2}, \ldots, \omega^{-(n-1)}$.

ω^{-1} is a primitive nth root of unity!

Evaluation: FFT(A,ω).
Interpolation: $\frac{1}{n}$ FFT(A,ω^{-1}).

$\implies O(n \log n)$ time for multiplying degree n polynomials.

Fast convolution!

Computing inverse.

FFT works with points with basic root of unity: ω or ω^{-1}

$1, \omega^{-1}, \omega^{-2}, \ldots, \omega^{-(n-1)}$.

ω^{-1} is a primitive nth root of unity!

Evaluation: FFT(A,ω).
Interpolation: $\frac{1}{n}$ FFT(A,ω^{-1}).

$\implies O(n \log n)$ time for multiplying degree n polynomials.

Fast convolution! Digital signal processing.

Computing inverse.

FFT works with points with basic root of unity: ω or ω^{-1}

$1, \omega^{-1}, \omega^{-2}, \ldots, \omega^{-(n-1)}$.

ω^{-1} is a primitive nth root of unity!

Evaluation: FFT(A,ω).
Interpolation: $\frac{1}{n}$ FFT(A,ω^{-1}).

$\implies O(n \log n)$ time for multiplying degree n polynomials.

Fast convolution! Digital signal processing. Many other things.

Computing inverse.

FFT works with points with basic root of unity: ω or ω^{-1}
  $1, \omega^{-1}, \omega^{-2}, \ldots, \omega^{-(n-1)}$.
ω^{-1} is a primitive nth root of unity!

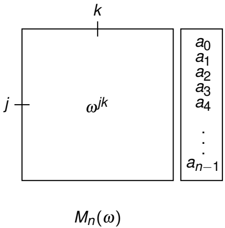
Evaluation: FFT(A,ω).
Interpolation: $\frac{1}{n}$ FFT(A,ω^{-1}).

$\implies O(n \log n)$ time for multiplying degree n polynomials.

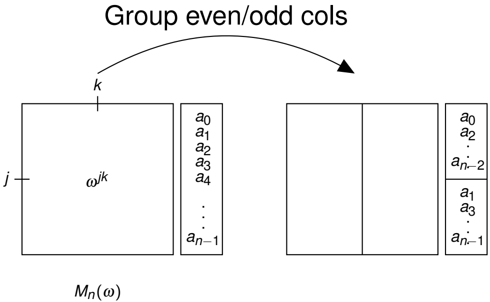
Fast convolution! Digital signal processing. Many other things.

Cool!!

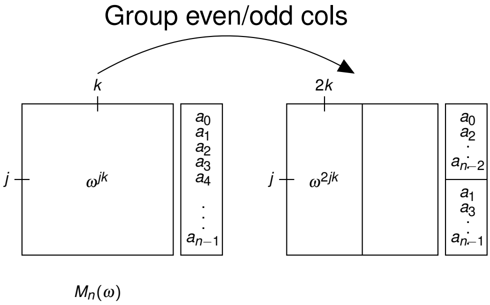
FFT: a closer look.



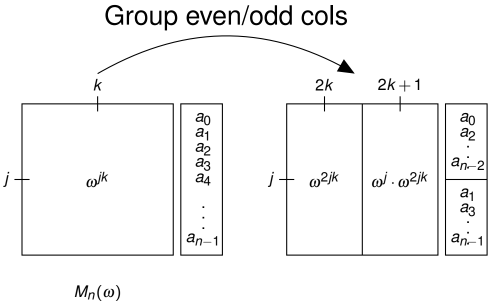
FFT: a closer look.



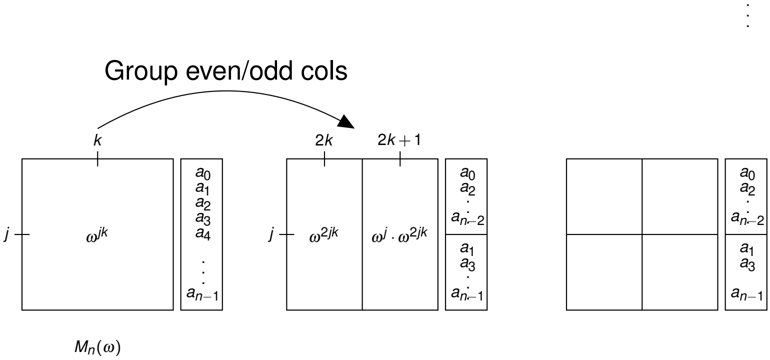
FFT: a closer look.



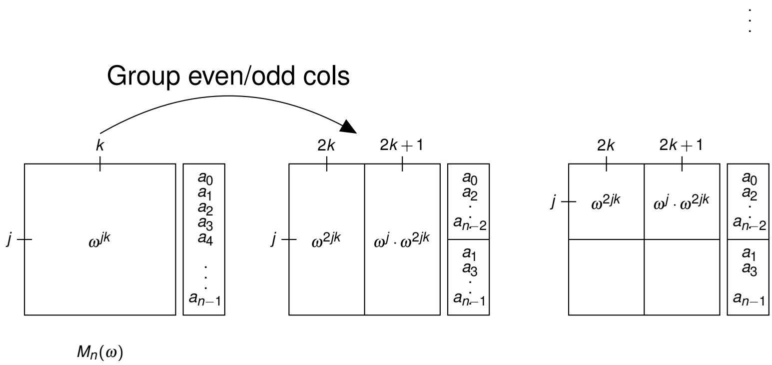
FFT: a closer look.



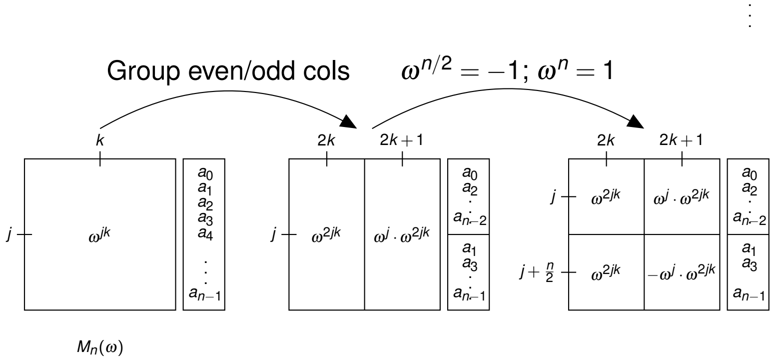
FFT: a closer look.



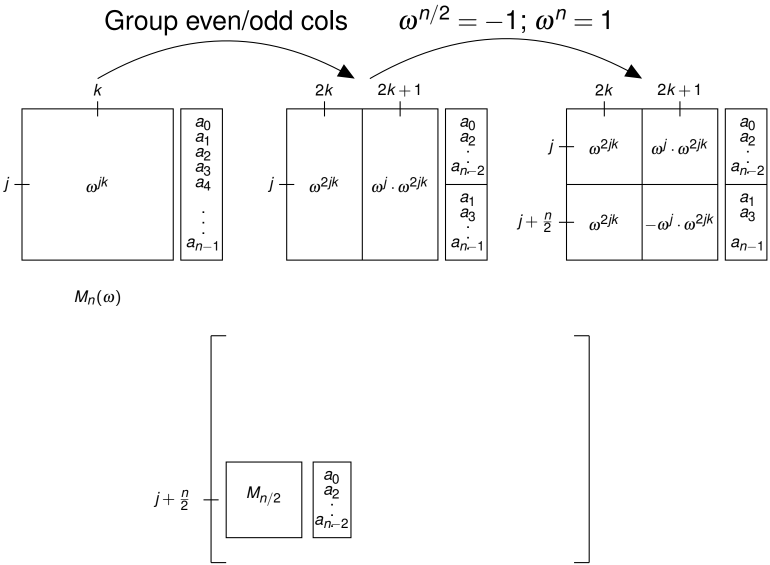
FFT: a closer look.



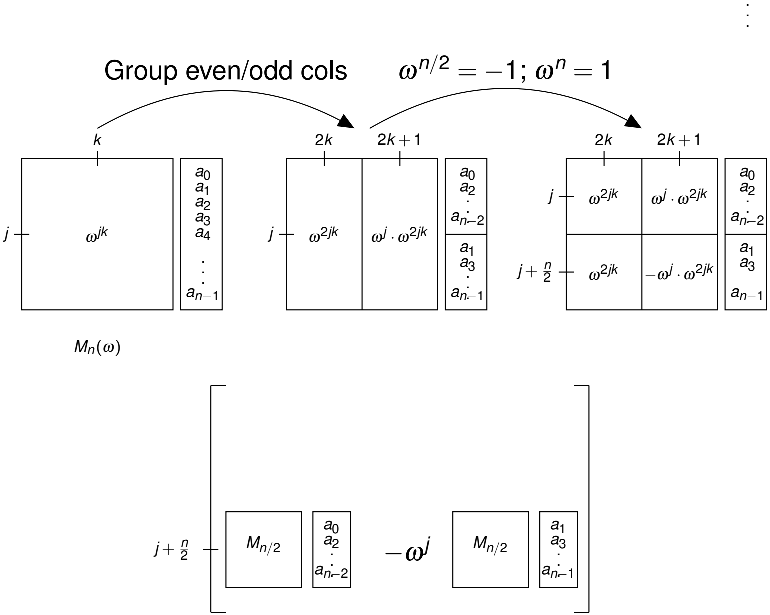
FFT: a closer look.



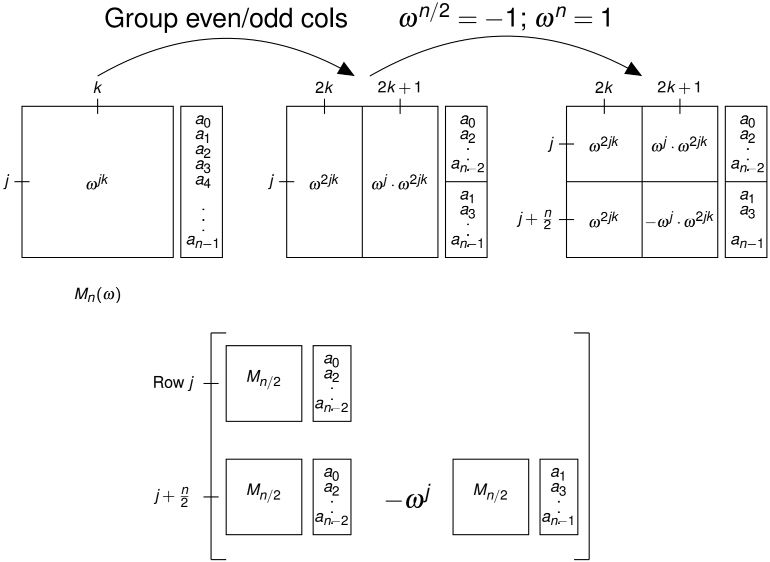
FFT: a closer look.



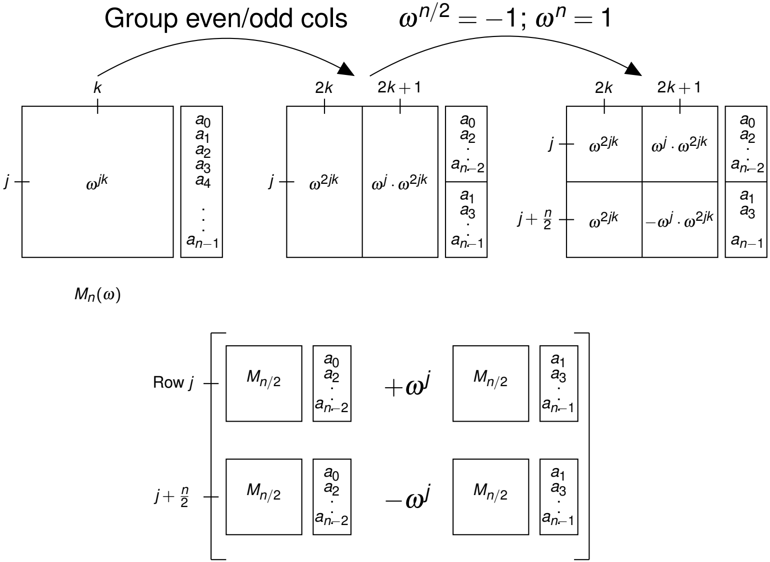
FFT: a closer look.



FFT: a closer look.



FFT: a closer look.

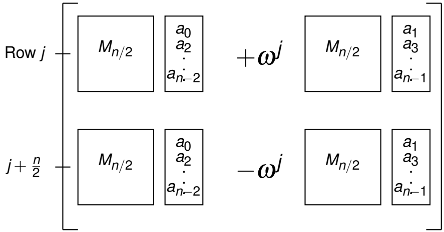


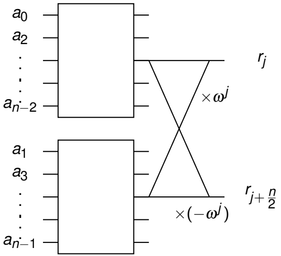
Unfolding FFT.

$$\begin{array}{c} \text{Row } j \\ \\ j+\frac{n}{2} \end{array} \begin{bmatrix} \boxed{M_{n/2}} \begin{bmatrix} a_0 \\ a_2 \\ \vdots \\ a_{n-2} \end{bmatrix} + \omega^j \boxed{M_{n/2}} \begin{bmatrix} a_1 \\ a_3 \\ \vdots \\ a_{n-1} \end{bmatrix} \\ \\ \boxed{M_{n/2}} \begin{bmatrix} a_0 \\ a_2 \\ \vdots \\ a_{n-2} \end{bmatrix} - \omega^j \boxed{M_{n/2}} \begin{bmatrix} a_1 \\ a_3 \\ \vdots \\ a_{n-1} \end{bmatrix} \end{bmatrix}$$

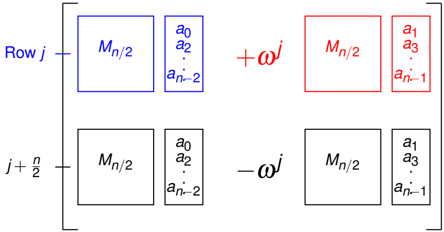
Butterfly switches!

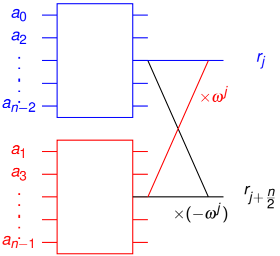
Unfolding FFT.





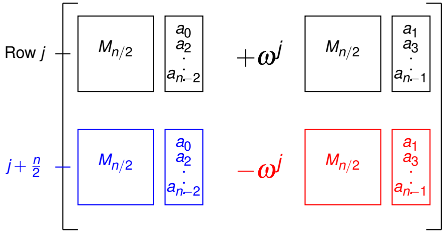
Unfolding FFT.

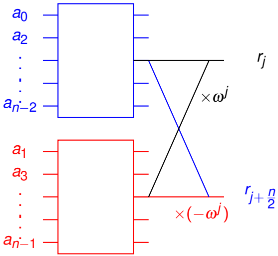




Butterfly switches!

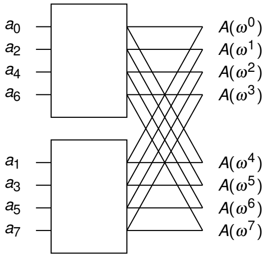
Unfolding FFT.



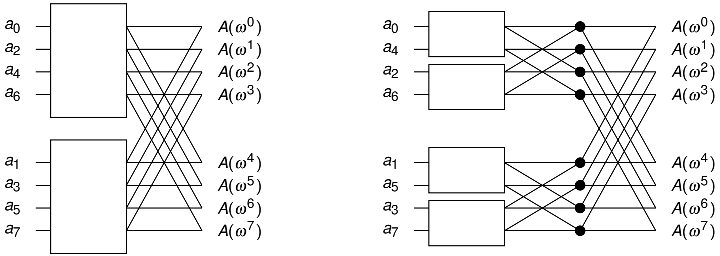


Butterfly switches!

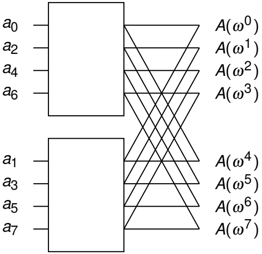
Expanding FFT...

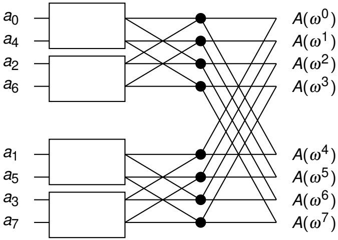


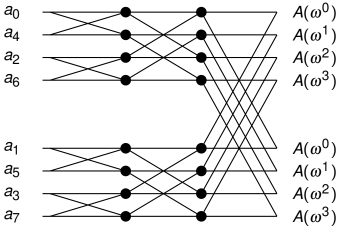
Expanding FFT...



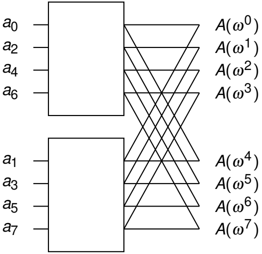
Expanding FFT...

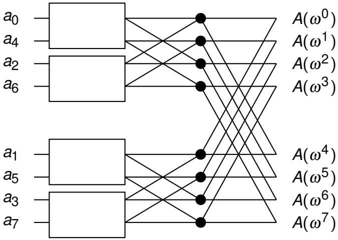


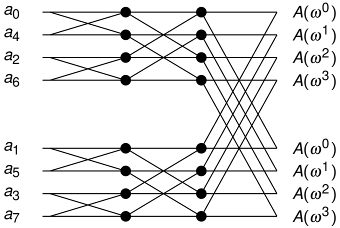




Expanding FFT...

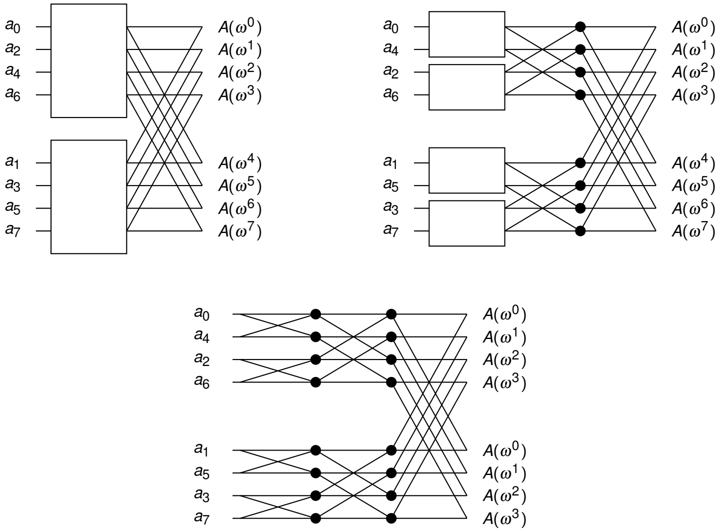






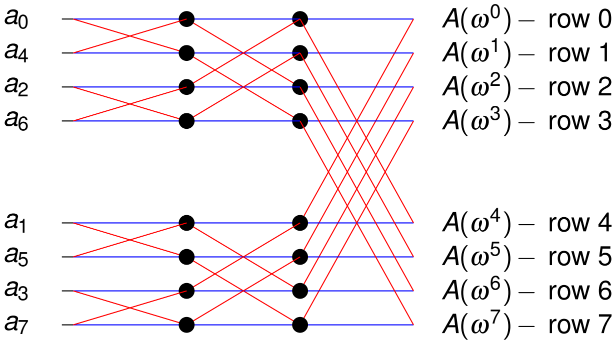
Edges from lower half of FFT have multipliers!

Expanding FFT...



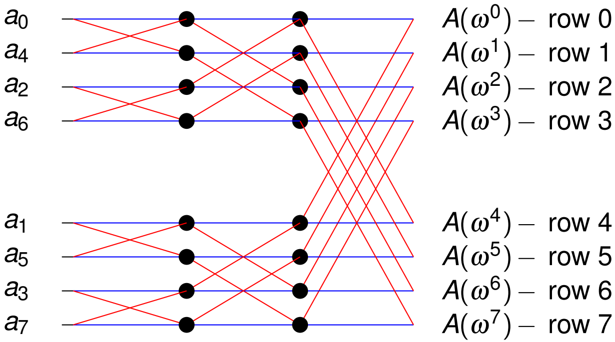
Edges from lower half of FFT have multipliers!

FFT Network.



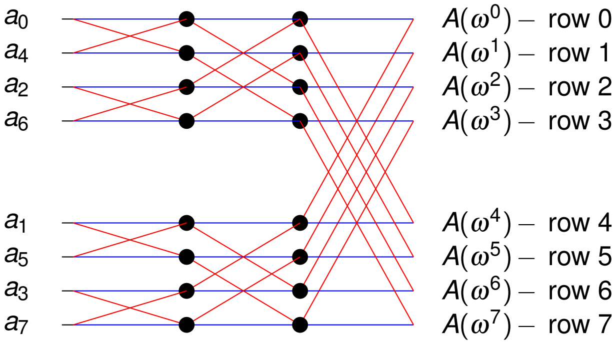
$\log N$ - levels.

FFT Network.



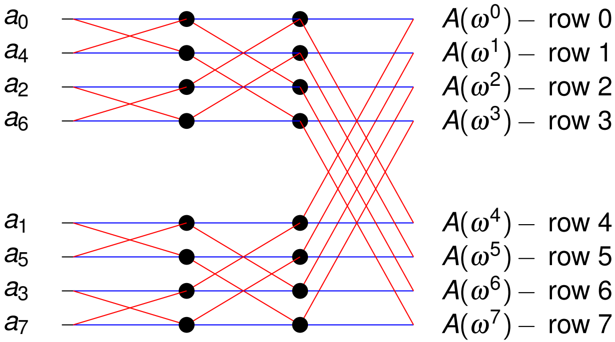
$\log N$ - levels.
N - rows.

FFT Network.



$\log N$ - levels.
N - rows.
In level i:

FFT Network.



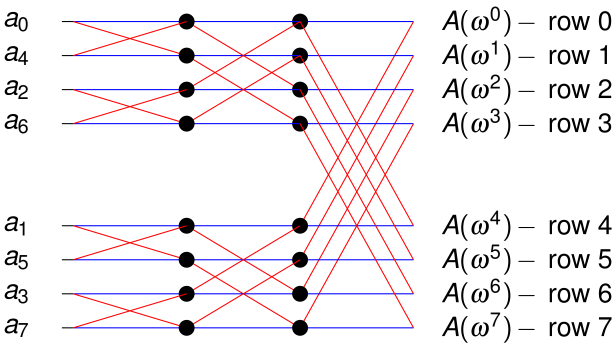
$\log N$ - levels.

N - rows.

In level i:

Row r node is connected to row r node in level $i+1$.

FFT Network.



$\log N$ - levels.

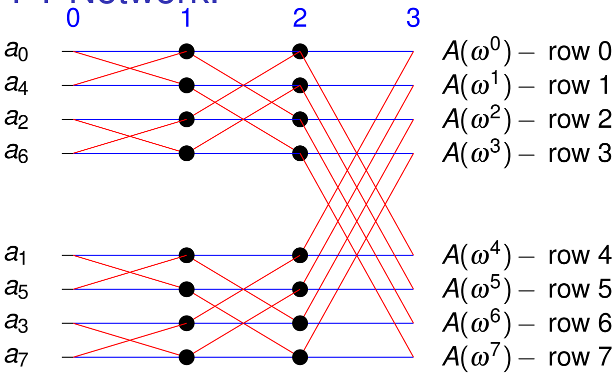
N - rows.

In level i:

Row r node is connected to row r node in level $i+1$.

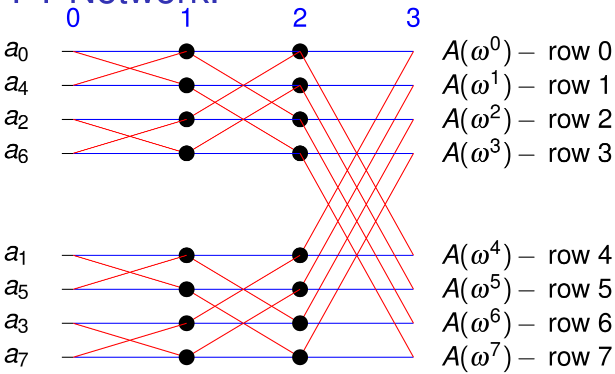
Row r node connected to row $r \pm 2^i$ node in level $i+1$

FFT Network.



Row r node connected to row $r \pm 2^i$ node in level $i+1$

FFT Network.



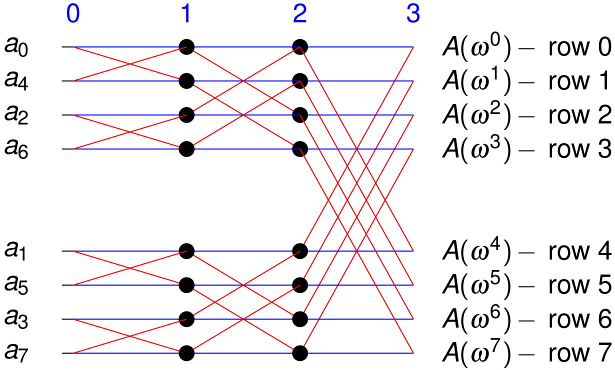
Row r node connected to row $r \pm 2^i$ node in level $i+1$

When is it $r + 2^i$?

(A) When $\lfloor r/2^i \rfloor$ is odd.

(B) When $\lfloor r/2^i \rfloor$ is even.

FFT Network.



Row r node connected to row $r \pm 2^i$ node in level $i+1$

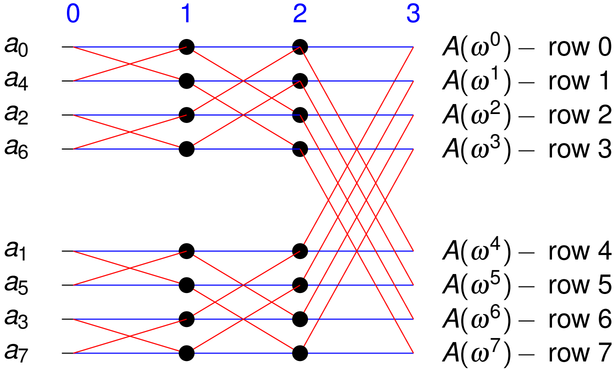
When is it $r + 2^i$?

(A) When $\lfloor r/2^i \rfloor$ is odd.

(B) When $\lfloor r/2^i \rfloor$ is even.

(B).

FFT Network.



Row r node connected to row $r \pm 2^i$ node in level $i+1$

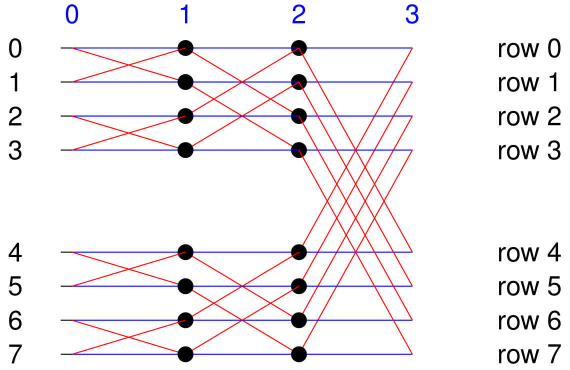
When is it $r + 2^i$?

(A) When $\lfloor r/2^i \rfloor$ is odd.

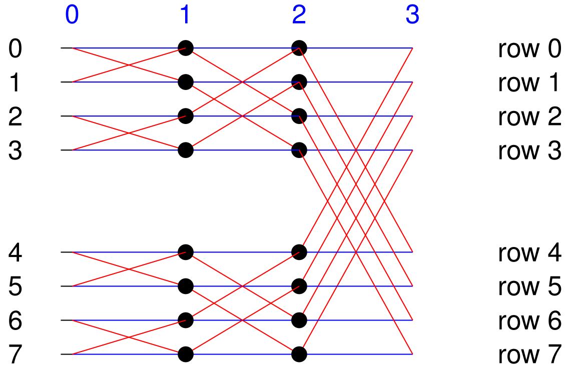
(B) When $\lfloor r/2^i \rfloor$ is even.

(B). Red edges flip bit!

Communication Architecture.

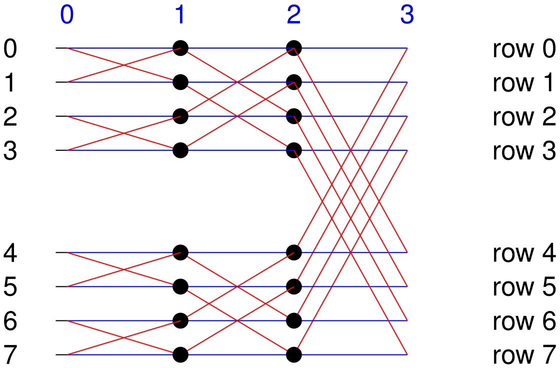


Communication Architecture.



Route from input $i = 101$ to output $j = 000$?

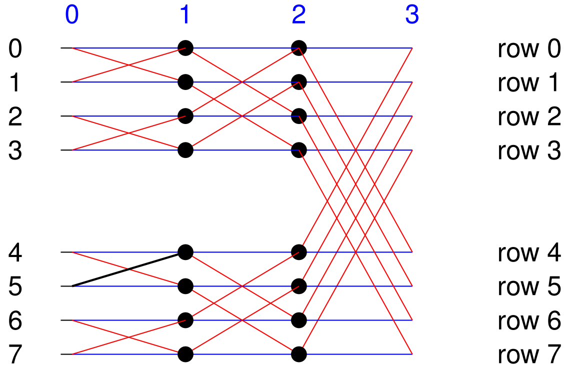
Communication Architecture.



Route from input $i = 101$ to output $j = 000$?

Flip first bit.

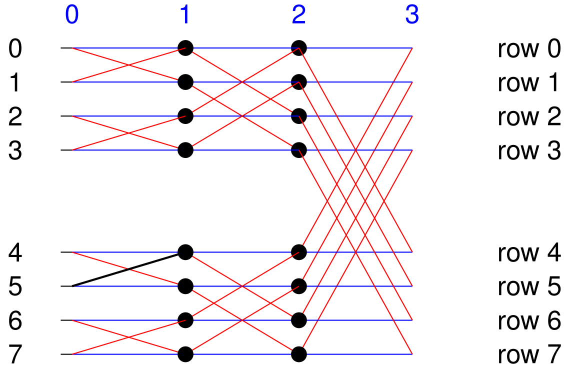
Communication Architecture.



Route from input $i = 101$ to output $j = 000$?

Flip first bit. Red (cross) edge.

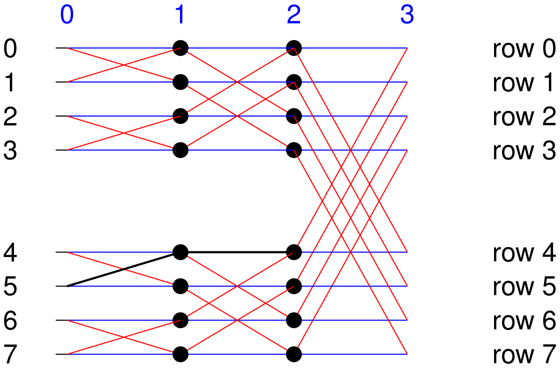
Communication Architecture.



Route from input $i = 101$ to output $j = 000$?

Flip first bit. Red (cross) edge.
Keep second bit.

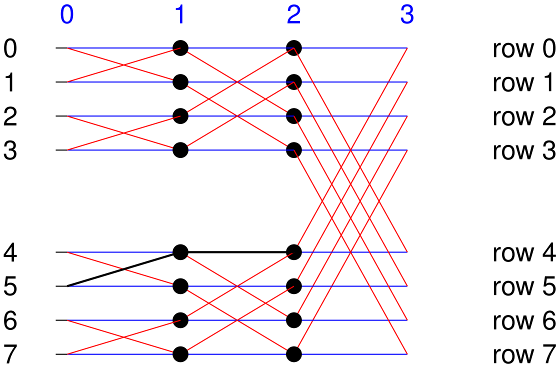
Communication Architecture.



Route from input $i = 101$ to output $j = 000$?

Flip first bit. Red (cross) edge.
Keep second bit. Blue (straight) edge.

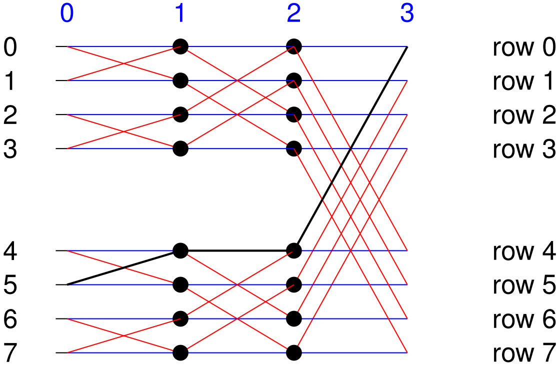
Communication Architecture.



Route from input $i = 101$ to output $j = 000$?

Flip first bit. Red (cross) edge.
Keep second bit. Blue (straight) edge.
Flip third bit.

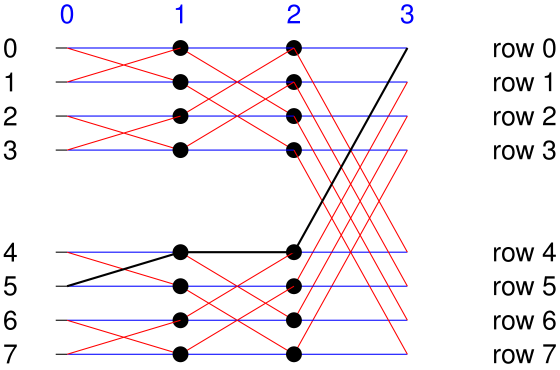
Communication Architecture.



Route from input $i = 101$ to output $j = 000$?

Flip first bit. Red (cross) edge.
Keep second bit. Blue (straight) edge.
Flip third bit. Red (cross edge).

Communication Architecture.



Route from input $i = 101$ to output $j = 000$?

Flip first bit. Red (cross) edge.
Keep second bit. Blue (straight) edge.
Flip third bit. Red (cross edge).

Basis of communication switches!

Algorithms, circuits, and parallelism.

Add up n numbers; $a_0, \ldots a_{n-1}$.

Algorithms, circuits, and parallelism.

Add up n numbers; $a_0, \ldots a_{n-1}$.

$O(n)$ time.

Algorithms, circuits, and parallelism.

Add up n numbers; $a_0, \ldots a_{n-1}$.

$O(n)$ time.

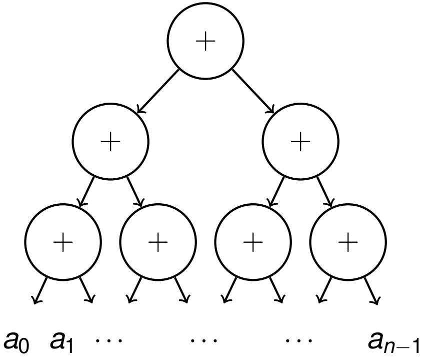
Circuit:

Algorithms, circuits, and parallelism.

Add up n numbers; $a_0, \ldots a_{n-1}$.

$O(n)$ time.

Circuit:



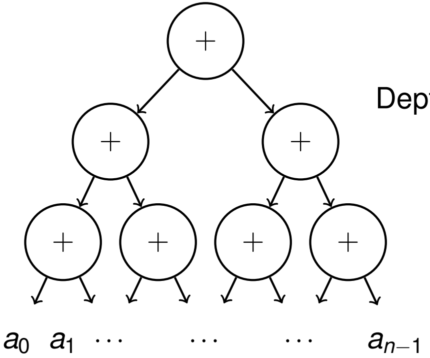
Depth $logn$.

Algorithms, circuits, and parallelism.

Add up n numbers; $a_0, \ldots a_{n-1}$.

$O(n)$ time.

Circuit:



Depth $log n$.

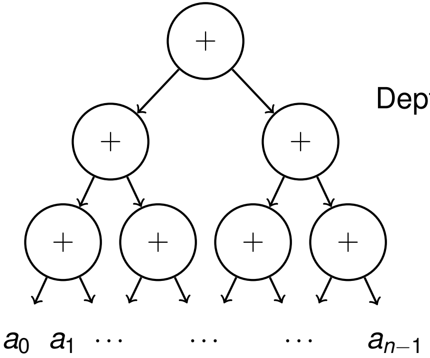
Depth $\log n \times$ (depth of "$+$").

Algorithms, circuits, and parallelism.

Add up n numbers; $a_0, \ldots a_{n-1}$.

$O(n)$ time.

Circuit:



Depth $logn$.

Depth $\log n \times$ (depth of "$+$").

Size: $O(n)$.

Other Circuits.

FFT Algorithm:

Other Circuits.

FFT Algorithm:
Sequential: $O(n \log n)$

Other Circuits.

FFT Algorithm:
Sequential: $O(n \log n)$

Depth: $O(\log n)$.

Other Circuits.

FFT Algorithm:
Sequential: $O(n \log n)$

Depth: $O(\log n)$.
Size: $O(n \log n)$.

Other Circuits.

FFT Algorithm:
Sequential: $O(n \log n)$

Depth: $O(\log n)$.
Size: $O(n \log n)$.

How to build a circuit for Matrix Multiplication?

Other Circuits.

FFT Algorithm:
Sequential: $O(n \log n)$

Depth: $O(\log n)$.
Size: $O(n \log n)$.

How to build a circuit for Matrix Multiplication?

Who makes circuits anyway!!

Other Circuits.

FFT Algorithm:
Sequential: $O(n \log n)$

Depth: $O(\log n)$.
Size: $O(n \log n)$.

How to build a circuit for Matrix Multiplication?

Who makes circuits anyway!!

Not too many people.

Other Circuits.

FFT Algorithm:
Sequential: $O(n \log n)$

Depth: $O(\log n)$.
Size: $O(n \log n)$.

How to build a circuit for Matrix Multiplication?

Who makes circuits anyway!!

Not too many people.

Parallel?

Other Circuits.

FFT Algorithm:
Sequential: $O(n \log n)$

Depth: $O(\log n)$.
Size: $O(n \log n)$.

How to build a circuit for Matrix Multiplication?

Who makes circuits anyway!!

Not too many people.

Parallel?

Everywhere!

One model for parallel.

Shared memory model (SMM): Processors.

One model for parallel.

Shared memory model (SMM): Processors.
Any processor accesses memory location in 1 time step.

One model for parallel.

Shared memory model (SMM): Processors.
Any processor accesses memory location in 1 time step.

One model for parallel.

Shared memory model (SMM): Processors.
Any processor accesses memory location in 1 time step.

Models parallelism.

One model for parallel.

Shared memory model (SMM): Processors.
Any processor accesses memory location in 1 time step.

Models parallelism.
"Ignores communication cost"

One model for parallel.

Shared memory model (SMM): Processors.
Any processor accesses memory location in 1 time step.

Models parallelism.
"Ignores communication cost"

Algorithm:

One model for parallel.

Shared memory model (SMM): Processors.
Any processor accesses memory location in 1 time step.

Models parallelism.
"Ignores communication cost"

Algorithm:
Time:

One model for parallel.

Shared memory model (SMM): Processors.
Any processor accesses memory location in 1 time step.

Models parallelism.
"Ignores communication cost"

Algorithm:
Time: How many steps?

One model for parallel.

Shared memory model (SMM): Processors.
Any processor accesses memory location in 1 time step.

Models parallelism.
"Ignores communication cost"

Algorithm:
Time: How many steps?
Work:

One model for parallel.

Shared memory model (SMM): Processors.
Any processor accesses memory location in 1 time step.

Models parallelism.
"Ignores communication cost"

Algorithm:
Time: How many steps?
Work: How much total work over processors.

One model for parallel.

Shared memory model (SMM): Processors.
Any processor accesses memory location in 1 time step.

Models parallelism.
"Ignores communication cost"

Algorithm:
Time: How many steps?
Work: How much total work over processors.

Circuits $\implies$ SMM
Time on SMM = Depth of circuit.

One model for parallel.

Shared memory model (SMM): Processors.
Any processor accesses memory location in 1 time step.

Models parallelism.
"Ignores communication cost"

Algorithm:
Time: How many steps?
Work: How much total work over processors.

Circuits $\implies$ SMM
Time on SMM = Depth of circuit.
Work on SMM = Size of circuit

Adding n numbers

$[a_0, a_1, a_2, \ldots, a_{n-1}]$

Adding n numbers

$[a_0, a_1, a_2, \ldots, a_{n-1}]$

Pair up and add.

Adding n numbers

$[a_0, a_1, a_2, \ldots, a_{n-1}]$

Pair up and add.

$[(a_0 + a_1), (a_2 + a_3), \ldots, (a_{n-2} + a_{n-1})]$.

Adding n numbers

$[a_0, a_1, a_2, \ldots, a_{n-1}]$

Pair up and add.

$[(a_0 + a_1), (a_2 + a_3), \ldots, (a_{n-2} + a_{n-1})]$.

And so on...

Adding n numbers

$[a_0, a_1, a_2, \ldots, a_{n-1}]$

Pair up and add.

$[(a_0 + a_1), (a_2 + a_3), \ldots, (a_{n-2} + a_{n-1})]$.

And so on...

Time: $\log n$

Adding n numbers

$[a_0, a_1, a_2, \ldots, a_{n-1}]$

Pair up and add.

$[(a_0 + a_1), (a_2 + a_3), \ldots, (a_{n-2} + a_{n-1})]$.

And so on...

Time: $\log n$

Work: $\frac{n}{2} + \frac{n}{4} + \cdots 1$

Adding n numbers

$[a_0, a_1, a_2, \ldots, a_{n-1}]$

Pair up and add.

$[(a_0 + a_1), (a_2 + a_3), \ldots, (a_{n-2} + a_{n-1})]$.

And so on...

Time: $\log n$

Work: $\frac{n}{2} + \frac{n}{4} + \cdots 1 = O(n)$.

Mergesort.

Algorithm:

Split into two lists.

Mergesort.

Algorithm:

Split into two lists.

Sort sublists

Mergesort.

Algorithm:
Split into two lists.
Sort sublists
Merge sublists.

Mergesort.

Algorithm:
Split into two lists.
Sort sublists
Merge sublists.

Mergesort.

Algorithm:
Split into two lists.
Sort sublists
Merge sublists.

Recursion depth: $O(\log n)$.

Mergesort.

Algorithm:
 Split into two lists.
 Sort sublists
 Merge sublists.

Recursion depth: $O(\log n)$.

Time: $O(\log n)$

Mergesort.

Algorithm:
Split into two lists.
Sort sublists
Merge sublists.

Recursion depth: $O(\log n)$.

Time: $O(\log n)$??
Merge was sequential!

Mergesort.

Algorithm:
Split into two lists.
Sort sublists
Merge sublists.

Recursion depth: $O(\log n)$.

Time: $O(\log n)$??
Merge was sequential!
Time: $O(n) + O(n/2) + O(n/4) \cdots$

Mergesort.

Algorithm:

Split into two lists.

Sort sublists

Merge sublists.

Recursion depth: $O(\log n)$.

Time: $O(\log n)$??

Merge was sequential!

Time: $O(n) + O(n/2) + O(n/4) \cdots = O(n)$.

Mergesort.

Algorithm:
Split into two lists.
Sort sublists
Merge sublists.

Recursion depth: $O(\log n)$.

Time: $O(\log n)$??
Merge was sequential!
Time: $O(n) + O(n/2) + O(n/4) \cdots = O(n)$.

Make Merge faster?

Mergesort.

Algorithm:
Split into two lists.
Sort sublists
Merge sublists.

Recursion depth: $O(\log n)$.

Time: $O(\log n)$??
Merge was sequential!
Time: $O(n) + O(n/2) + O(n/4) \cdots = O(n)$.

Make Merge faster?
In $O(\log n)$ time using Mirrored FFT of comparators!
Bitonic Sort.

Mergesort.

Algorithm:
 Split into two lists.
 Sort sublists
 Merge sublists.

Recursion depth: $O(\log n)$.

Time: $O(\log n)$??
Merge was sequential!
Time: $O(n) + O(n/2) + O(n/4) \cdots = O(n)$.

Make Merge faster?
In $O(\log n)$ time using Mirrored FFT of comparators!
Bitonic Sort.

Mergesort.

Algorithm:
Split into two lists.
Sort sublists
Merge sublists.

Recursion depth: $O(\log n)$.

Time: $O(\log n)$??
Merge was sequential!
Time: $O(n) + O(n/2) + O(n/4) \cdots = O(n)$.

Make Merge faster?
In $O(\log n)$ time using Mirrored FFT of comparators!
Bitonic Sort.

Time: $O(\log^2 n)$.

Mergesort.

Algorithm:
 Split into two lists.
 Sort sublists
 Merge sublists.

Recursion depth: $O(\log n)$.

Time: $O(\log n)$??
Merge was sequential!
Time: $O(n) + O(n/2) + O(n/4) \cdots = O(n)$.

Make Merge faster?
In $O(\log n)$ time using Mirrored FFT of comparators!
Bitonic Sort.

Time: $O(\log^2 n)$.
Work: $O(n \log^2 n)$

Mergesort.

Algorithm:
Split into two lists.
Sort sublists
Merge sublists.

Recursion depth: $O(\log n)$.

Time: $O(\log n)$??
Merge was sequential!
Time: $O(n) + O(n/2) + O(n/4) \cdots = O(n)$.

Make Merge faster?
In $O(\log n)$ time using Mirrored FFT of comparators!
Bitonic Sort.

Time: $O(\log^2 n)$.
Work: $O(n \log^2 n)$ Worse than sequential!

Mergesort.

Algorithm:
 Split into two lists.
 Sort sublists
 Merge sublists.

Recursion depth: $O(\log n)$.

Time: $O(\log n)$??
Merge was sequential!
Time: $O(n) + O(n/2) + O(n/4) \cdots = O(n)$.

Make Merge faster?
In $O(\log n)$ time using Mirrored FFT of comparators!
Bitonic Sort.

Time: $O(\log^2 n)$.
Work: $O(n \log^2 n)$ Worse than sequential!
Cole's Mergesort:

Mergesort.

Algorithm:
 Split into two lists.
 Sort sublists
 Merge sublists.

Recursion depth: $O(\log n)$.

Time: $O(\log n)$??
Merge was sequential!
Time: $O(n) + O(n/2) + O(n/4) \cdots = O(n)$.

Make Merge faster?
In $O(\log n)$ time using Mirrored FFT of comparators!
Bitonic Sort.

Time: $O(\log^2 n)$.
Work: $O(n \log^2 n)$ Worse than sequential!
Cole's Mergesort: $O(\log n)$ time;

Mergesort.

Algorithm:
 Split into two lists.
 Sort sublists
 Merge sublists.

Recursion depth: $O(\log n)$.

Time: $O(\log n)$??
Merge was sequential!
Time: $O(n) + O(n/2) + O(n/4) \cdots = O(n)$.

Make Merge faster?
In $O(\log n)$ time using Mirrored FFT of comparators!
Bitonic Sort.

Time: $O(\log^2 n)$.
Work: $O(n \log^2 n)$ Worse than sequential!
Cole's Mergesort: $O(\log n)$ time; $O(n \log n)$ work.

Mergesort.

Algorithm:
 Split into two lists.
 Sort sublists
 Merge sublists.

Recursion depth: $O(\log n)$.

Time: $O(\log n)$??
Merge was sequential!
Time: $O(n) + O(n/2) + O(n/4) \cdots = O(n)$.

Make Merge faster?
In $O(\log n)$ time using Mirrored FFT of comparators!
Bitonic Sort.

Time: $O(\log^2 n)$.
Work: $O(n \log^2 n)$ Worse than sequential!
Cole's Mergesort: $O(\log n)$ time; $O(n \log n)$ work. Optimal!

Large integer multiplication.

Algorithm:

Split n bit numbers into $n/2$ bit numbers.

Large integer multiplication.

Algorithm:

Split n bit numbers into $n/2$ bit numbers.

Do three (or four) multiplications of those $n/2$ bit numbers.

Large integer multiplication.

Algorithm:

Split n bit numbers into $n/2$ bit numbers.

Do three (or four) multiplications of those $n/2$ bit numbers.

Shift and Add the numbers.

Large integer multiplication.

Algorithm:

Split n bit numbers into $n/2$ bit numbers.
Do three (or four) multiplications of those $n/2$ bit numbers.
Shift and Add the numbers.

Large integer multiplication.

Algorithm:

Split n bit numbers into $n/2$ bit numbers.
Do three (or four) multiplications of those $n/2$ bit numbers.
Shift and Add the numbers.

Recursion depth: $\log n$.

Large integer multiplication.

Algorithm:

Split n bit numbers into $n/2$ bit numbers.
Do three (or four) multiplications of those $n/2$ bit numbers.
Shift and Add the numbers.

Recursion depth: $\log n$.
Addition of n-bit numbers: $O(n)$ time.

Large integer multiplication.

Algorithm:

Split n bit numbers into $n/2$ bit numbers.
Do three (or four) multiplications of those $n/2$ bit numbers.
Shift and Add the numbers.

Recursion depth: $\log n$.
Addition of n-bit numbers: $O(n)$ time.

Large integer multiplication.

Algorithm:

Split n bit numbers into $n/2$ bit numbers.
Do three (or four) multiplications of those $n/2$ bit numbers.
Shift and Add the numbers.

Recursion depth: $\log n$.
Addition of n-bit numbers: $O(n)$ time.

Work mostly at the leaves:

Large integer multiplication.

Algorithm:

Split n bit numbers into $n/2$ bit numbers.
Do three (or four) multiplications of those $n/2$ bit numbers.
Shift and Add the numbers.

Recursion depth: $\log n$.
Addition of n-bit numbers: $O(n)$ time.

Work mostly at the leaves:
How many leaves?

Large integer multiplication.

Algorithm:

Split n bit numbers into $n/2$ bit numbers.
Do three (or four) multiplications of those $n/2$ bit numbers.
Shift and Add the numbers.

Recursion depth: $\log n$.
Addition of n-bit numbers: $O(n)$ time.

Work mostly at the leaves:
How many leaves? $O(n^{\log_2 3})$.

Large integer multiplication.

Algorithm:

Split n bit numbers into $n/2$ bit numbers.
Do three (or four) multiplications of those $n/2$ bit numbers.
Shift and Add the numbers.

Recursion depth: $\log n$.
Addition of n-bit numbers: $O(n)$ time.

Work mostly at the leaves:
How many leaves? $O(n^{\log_2 3})$.

Depth: $O(n) + O(n/2) + \cdots + O(1)$

Large integer multiplication.

Algorithm:

Split n bit numbers into $n/2$ bit numbers.
Do three (or four) multiplications of those $n/2$ bit numbers.
Shift and Add the numbers.

Recursion depth: $\log n$.
Addition of n-bit numbers: $O(n)$ time.

Work mostly at the leaves:
How many leaves? $O(n^{\log_2 3})$.

Depth: $O(n) + O(n/2) + \cdots + O(1) = O(n)$

Large integer multiplication.

Algorithm:

Split n bit numbers into $n/2$ bit numbers.
Do three (or four) multiplications of those $n/2$ bit numbers.
Shift and Add the numbers.

Recursion depth: $\log n$.
Addition of n-bit numbers: $O(n)$ time.

Work mostly at the leaves:
How many leaves? $O(n^{\log_2 3})$.

Depth: $O(n) + O(n/2) + \cdots + O(1) = O(n)$
Work: $O(n^{\log_2 3})$.

Large integer multiplication.

Algorithm:

Split n bit numbers into $n/2$ bit numbers.
Do three (or four) multiplications of those $n/2$ bit numbers.
Shift and Add the numbers.

Recursion depth: $\log n$.
Addition of n-bit numbers: $O(n)$ time.

Work mostly at the leaves:
How many leaves? $O(n^{\log_2 3})$.

Depth: $O(n) + O(n/2) + \cdots + O(1) = O(n)$
Work: $O(n^{\log_2 3})$.

Depth: $O(\log^2 n)$ easily done.

Large integer multiplication.

Algorithm:

Split n bit numbers into $n/2$ bit numbers.
Do three (or four) multiplications of those $n/2$ bit numbers.
Shift and Add the numbers.

Recursion depth: $\log n$.
Addition of n-bit numbers: $O(n)$ time.

Work mostly at the leaves:
How many leaves? $O(n^{\log_2 3})$.

Depth: $O(n) + O(n/2) + \cdots + O(1) = O(n)$
Work: $O(n^{\log_2 3})$.

Depth: $O(\log^2 n)$ easily done. Better is possible.

Large integer multiplication.

Algorithm:

Split n bit numbers into $n/2$ bit numbers.
Do three (or four) multiplications of those $n/2$ bit numbers.
Shift and Add the numbers.

Recursion depth: $\log n$.
Addition of n-bit numbers: $O(n)$ time.

Work mostly at the leaves:
How many leaves? $O(n^{\log_2 3})$.

Depth: $O(n) + O(n/2) + \cdots + O(1) = O(n)$
Work: $O(n^{\log_2 3})$.

Depth: $O(\log^2 n)$ easily done. Better is possible.

Better Sequential Algorithm as well!

Large integer multiplication.

Algorithm:

Split n bit numbers into $n/2$ bit numbers.
Do three (or four) multiplications of those $n/2$ bit numbers.
Shift and Add the numbers.

Recursion depth: $\log n$.
Addition of n-bit numbers: $O(n)$ time.

Work mostly at the leaves:
How many leaves? $O(n^{\log_2 3})$.

Depth: $O(n) + O(n/2) + \cdots + O(1) = O(n)$
Work: $O(n^{\log_2 3})$.

Depth: $O(\log^2 n)$ easily done. Better is possible.

Better Sequential Algorithm as well!
Time: $O(n \log n \log\log n)$ Schönhage-Strassen.

Large integer multiplication.

Algorithm:

Split n bit numbers into $n/2$ bit numbers.
Do three (or four) multiplications of those $n/2$ bit numbers.
Shift and Add the numbers.

Recursion depth: $\log n$.
Addition of n-bit numbers: $O(n)$ time.

Work mostly at the leaves:
How many leaves? $O(n^{\log_2 3})$.

Depth: $O(n) + O(n/2) + \cdots + O(1) = O(n)$
Work: $O(n^{\log_2 3})$.

Depth: $O(\log^2 n)$ easily done. Better is possible.

Better Sequential Algorithm as well!
Time: $O(n \log n \log\log n)$ Schönhage-Strassen.
Care to guess how?

Large integer multiplication.

Algorithm:

Split n bit numbers into $n/2$ bit numbers.
Do three (or four) multiplications of those $n/2$ bit numbers.
Shift and Add the numbers.

Recursion depth: $\log n$.
Addition of n-bit numbers: $O(n)$ time.

Work mostly at the leaves:
How many leaves? $O(n^{\log_2 3})$.

Depth: $O(n) + O(n/2) + \cdots + O(1) = O(n)$
Work: $O(n^{\log_2 3})$.

Depth: $O(\log^2 n)$ easily done. Better is possible.

Better Sequential Algorithm as well!
Time: $O(n \log n \log\log n)$ Schönhage-Strassen.
Care to guess how?

Using the FFT!

Brent's Theorem.

What if I have fewer processors?

Brent's Theorem.

What if I have fewer processors?
Do complexity analysis again?

Brent's Theorem.

What if I have fewer processors?
Do complexity analysis again?

Simulation:

Brent's Theorem.

What if I have fewer processors?
Do complexity analysis again?

Simulation:
If some step uses P processors.

Brent's Theorem.

What if I have fewer processors?
Do complexity analysis again?

Simulation:
If some step uses P processors.
Simulate with p processors for $\lceil \frac{P}{p} \rceil$ steps.

Brent's Theorem.

What if I have fewer processors?
Do complexity analysis again?

Simulation:
If some step uses P processors.
Simulate with p processors for $\lceil \frac{P}{p} \rceil$ steps.

Theorem:
Any work W, time T parallel algorithm can run on a p-processor shared memory machine in time $\lfloor \frac{W}{p} \rfloor + T$ and work W.

Brent's Theorem.

What if I have fewer processors?
Do complexity analysis again?

Simulation:
If some step uses P processors.
Simulate with p processors for $\lceil \frac{P}{p} \rceil$ steps.

Theorem:
Any work W, time T parallel algorithm can run on a p-processor shared memory machine in time $\lfloor \frac{W}{p} \rfloor + T$ and work W.

Show Some Code.

Multiplication on my

Show Some Code.

Multiplication on my little laptop.

Show Some Code.

Multiplication on my little laptop.