Introduction to Algorithms

Topic 6-4 : Divide and Conquer

Xiang-Yang Li and Haisheng Tan

School of Computer Science and Technology
University of Science and Technology of China (USTC)

Fall Semester 2022

Outline

Overview

In divide and conquer approach, a problem is divided into smaller problems, then the smaller problems are solved independently, and finally the solutions of smaller problems are combined into a solution for the large problem.

Generally, divide-and-conquer algorithms have three parts:

- **Divide:** Divide the problem into a number of sub-problems that are smaller instances of the same problem.
- **Conquer:** Conquer the sub-problems by solving them recursively. If they are small enough, solve the sub-problems as base cases.
- **Combine:** Combine the solutions to the sub-problems into the solution for the original problem.

Pros and cons

Divide and conquer approach supports parallelism as sub-problems are independent. Hence, an algorithm, which is designed using this technique, can run on the multiprocessor system or in different machines simultaneously.

In this approach, most of the algorithms are designed using recursion, hence memory management is very high. For recursive function stack is used, where function state needs to be stored.

Contents

Problem Description

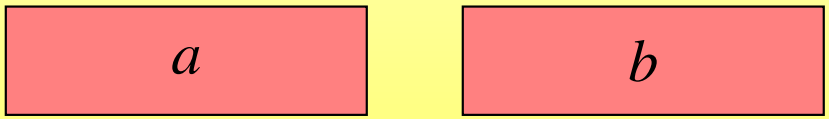
- **Input:** two $n-bit$ integers X and Y.
- **Output:** product of X and Y.
- **Example:** $X = 1980$ $Y = 2315$

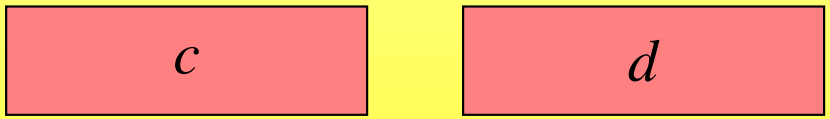
$$\begin{array}{rr} & 1980 \\ \times & 2315 \\ \hline & 9900 \\ & 1980 \\ & 5940 \\ + & 3960 \\ \hline = & 4573700 \end{array}$$

This is the algorithm you learned in grade school. Notice it takes $O(n^2)$ time.

Solution

Divide and Conquer

$X =$ 

$Y =$ 

$$X = a \cdot 2^{n/2} + b \qquad Y = c \cdot 2^{n/2} + d$$
$$XY = ac \cdot 2^n + (ad + bc) \cdot 2^{n/2} + bd$$

Being Clever

Time Complexity

$$T(n) = \begin{cases} O(1) & n = 1 \\ 4T(n/2) + O(n) & n > 1 \end{cases}$$

whose solution we claimed to be $T(n) = O(n^2)$. Compared to the original method, this method has not been significantly improved.In order to reduce the time complexity, we must reduce the number of multiplications.

$$XY = ac \cdot 2^n + (ad + bc) \cdot 2^{n/2} + bd \tag{1}$$

$$XY = ac \cdot 2^n - ((a-b)(c-d) - ac - bd) \cdot 2^{n/2} + bd \tag{2}$$

$$XY = ac \cdot 2^n + ((a+b)(c+d) - ac - bd) \cdot 2^{n/2} + bd \tag{3}$$

Being Clever

Analysis

The complexity of the last two XY multiplication schemes is $O(n^{log3})$, but considering $a+c$,$b+d$ may get the result of $n+1$ bit, which makes the size of the problem bigger, so it does not choose the third option.

$$T(n) = \begin{cases} O(1) & n = 1 \\ 3T(n/2) + O(n) & n > 1 \end{cases}$$

Thus, $T(n) = O(n^{log3}) = O(n^{1.59})$

Being Clever

- Is there a faster way?
- If you divide a large integer into more segments, combine them in a more complicated way,it will be possible to get a better algorithm.
- In the end, this idea led to the fast Fourier transform. This method can also be seen as a complex divide calculation method, for large integer multiplication, it can be solved in $O(nlogn)$ time.

Contents

Problem Description

Let us consider two matrices A and B. We want to calculate the resultant matrix C by multiplying A and B.

Naive Method

If $A = (a_{ij})$ and $B = (b_{ij})$ are square $n \times n$ matrices, then in the product $C = A \cdot B$, we define the entry c_{ij} , for $i, j = 1, 2, ..., n$, by $c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$. We must compute n^2 matrix entries, and each is the sum of n values.

Naive Method

SQUARE-MATRIX-MULTIPLY(A, B)

1: $n = A.rows$
2: let C be a new $n \times n$ matrix
3: **for** $i = 1$ to n **do**
4: **for** $j = 1$ to n **do**
5: $c_{ij} = 0$
6: **for** $k = 1$ to n **do**
7: $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
8: **return** C

Complexity

Here, we assume that integer operations take $O(1)$ time. There are three for loops in this algorithm and one is nested in other. Hence, the algorithm takes $O(n^3)$ time to execute

Simple Divide and Conquer

Following is simple Divide and Conquer method to multiply two square matrices.

- Divide matrices A and B in 4 sub-matrices of size $N/2 \times N/2$ as shown in the below diagram.
- Calculate following values recursively.

$$\begin{pmatrix} A_1 & A_2 \\ A_3 & A_4 \end{pmatrix} \times \begin{pmatrix} B_1 & B_2 \\ B_3 & B_4 \end{pmatrix} = \begin{pmatrix} A_1B_1 + A_2B_3 & A_1B_2 + A_2B_4 \\ A_3B_1 + A_4B_3 & A_3B_2 + A_4B_4 \end{pmatrix}$$

In the above method, we do 8 multiplications for matrices of size $N/2 \times N/2$ and 4 additions. Addition of two matrices takes $O(n^2)$ time. So the time complexity is $T(n) = 8T(n/2) + O(n^2)$. From Master's Theorem, time complexity of above method is $O(n^3)$ which is unfortunately same as the above naive method.

Simple Divide and Conquer also leads to $O(n^3)$, can there be a better way?

Strassens Matrix Multiplication

In the above divide and conquer method, the main component for high time complexity is 8 recursive calls. The idea of Strassens method is to reduce the number of recursive calls to 7.

Strassens method is similar to above simple divide and conquer method in the sense that this method also divide matrices to sub-matrices of size $N/2 \times N/2$ as shown in the above slide, but in Strassens method, the four sub-matrices of result are calculated using following formula.

Strassens Matrix Multiplication

The Strassen algorithm defines instead new matrices:

$$M_1 = (A_1 + A_4)(B_1 + B_4) \qquad M_2 = (A_3 + A_4)B_1$$
$$M_3 = A_1(B_2 - B_4) \qquad M_4 = A_4(B_3 - B_1)$$
$$M_5 = (A_1 + A_2)B_4 \qquad M_6 = (A_3 - A_1)(B_1 + B_2)$$
$$M_7 = (A_2 - A_4)(B_3 + B_4)$$

only using 7 multiplications(one for each M_k) instead of 8. We may now express the C_i in terms of M_k:

$$C_1 = M_1 + M_4 - M_5 + M_7$$
$$C_2 = M_3 + M_5$$
$$C_3 = M_2 + M_4$$
$$C_4 = M_1 - M_2 + M_3 + M_6$$

Time Complexity

- Addition and Subtraction of two matrices takes $O(n^2)$ time. So time complexity can be written as:

$$T(n) = 7T(n/2) + O(n^2)$$

From Master's Theorem, time complexity of above method is $O(n^{log7})$ which is approximately $O(n^{2.8074})$

- Hopcroft and Kerr have proved(1971) that 7 multiplications are necessary in two 2×2 matrices's multiplication. Therefore, to further improve the time of matrix multiplication complexity, it can no longer be based on the method of calculating the 7-times multiplication of 2×2 matrices. Perhaps a better algorithm for 3×3 or 5×5 matrices should be studied. Currently,the best calculation time upper bound is $O(n^{2.376})$.

Contents

Problem Description

- **Definition:** A **defective chessboad** is a $2^k \times 2^k$ board of squares with exactly one defective square
- **Example:**

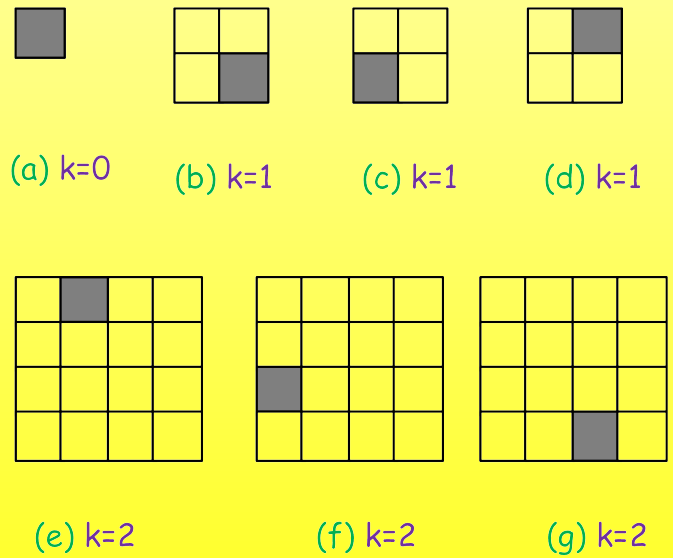


Figure: Defective chessboads

Problem Description

Defective Chessboard Problem

- The problem is to tile (cover) all nondefective cells using a **triomino**.
- A triomino is an L shaped object that can cover three squares of a chessboard.
- A triomino has four orientations:

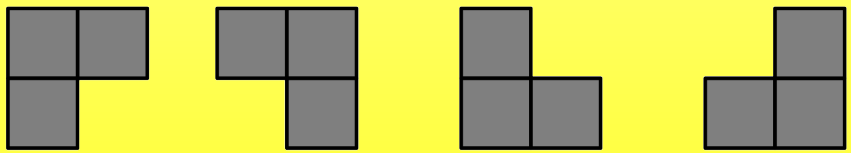
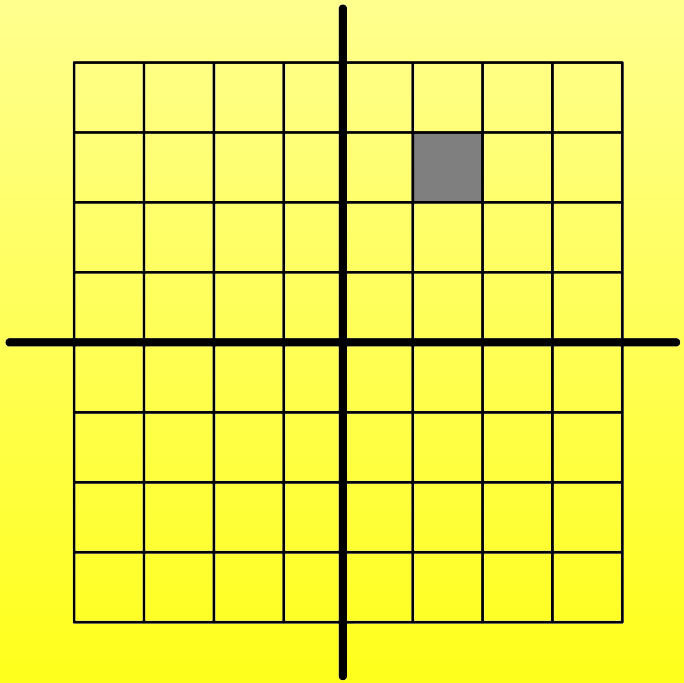


Figure: Trominoes with different orientations

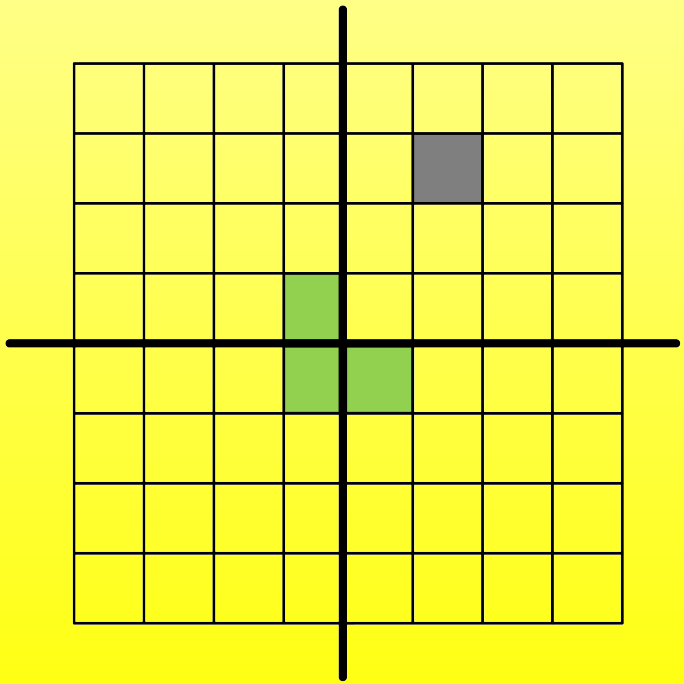
Tiling A Defective Chessboard

- Divide into four smaller chessboards.
- One of these is a defective chessboard.



Tiling A Defective Chessboard

- Make the other three chessboards defective by placing a triomino at their common corner.
- Recursively tile the four defective chessboards.



Time Complexity

- Let $n = 2^k$, d is a constant.
- Let $t(k)$ be the time taken to tile a $2^k \times 2^k$ defective chessboard. Then,

$$t(k) = \begin{cases} d & k = 0 \\ 4t(k-1) + c & k > 0 \end{cases}$$

- Here, c is constant representing time spent on finding the appropriate position for a triomino and to rotate the triomino for a required shape.

Time Complexity

$$
\begin{aligned}
t(k) &= 4t(k-1)+c \\
&= 4[4t(k-2)+c]+c \\
&= 4^2 t(k-2)+4c+c \\
&= \ldots \\
&= 4^k t(0)+4^{k-1}c+4^{k-2}c+\ldots+4^2 c+4c+c \\
&= 4^k d+4^{k-1}c+4^{k-2}c+\ldots+4^2 c+4c+c \\
&= \theta(4^k) \\
&= \theta(\textbf{number of triominoes placed})
\end{aligned}
$$

Since each grid must spend $\theta(1)$ time to place each triomino, it is impossible to get a faster algorithm for this problem.

Contents

Problem Description

Definition 1:

A polynomial in variable x over a field F (such as, R or C) is

$$A(x) = \sum_{j=0}^{n-1} a_j x^j \tag{1}$$

where $a_j \in F$ is called a coefficient.

Problem 1:

The straightforward method of polynomial multiplication is $\theta(n^2)$, where n is the degree of polynomials. While the Fast Fourier Transform (FFT) can reduce the time to $\theta(nlogn)$.

Representation of polynomials

Definition 2:

For the polynomial(1), we have two ways of representing it:

- **Coefficient Representation:**
  $a = (a_0, a_1, ..., a_{n-1})$ is a coefficient representation of (1).
- **Point-value Representation:**
  $P = \{(x_0, y_0), (x_1, y_1), ..., (x_{n-1}, y_{n-1})\}$ is a point-value representation of (1), where all of the x_k are *distinct* and $y_k = A(x_k), k = 0, 1, ..., n-1$.

Coefficient Representation

Property 1: Evaluating the polynomial (1) at a given point x_0 by Horners Rule:

$$A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \ldots + x_0(a_{n-2} + x_0 a_{n-1})))$$

which takes $\theta(n)$ time.

Property 2: Addition of n-dimensional coefficient vectors a and b (i.e., $a+b$) takes $\theta(n)$ time

The convolution: $a \otimes b = (c_0, c_1, \ldots, c_{2n-2})$ is

$$c_j = \sum_{k=0}^{j} a_k b_{j-k} \tag{2}$$

which indicates that it takes $\theta(n^2)$ time for polynomial multiplication by coefficient representation.

Point-value Representation

- By Horners rule, it takes $\theta(n^2)$ time to get a point-value representation of polynomial(1). If we choose x_k cleverly, the complexity reduces to $\theta(nlogn)$.
- **Definition 3:** The process of determining the coefficient form of a polynomial from a point-value representation is called interpolation.
- **Question:** Does the interpolation uniquely determine a polynomial? If not, the concept of interpolation is meaningless.

Uniqueness of Interpolation

Theorem 1:

For any set of n point-value pairs $\{(x_0,y_0),(x_1,y_1),...,(x_{n-1},y_{n-1})\}$, such that all the x_k values are distinct, there is a unique polynomial $A(x)$ of degree-bound n such that $y_k = A(x_k), k = 0,1,...,n-1$.

Proof:

Let $A(x) = \sum_{j=0}^{n-1} a_j x^j$, where a_j is undetermined, then we have

$$\begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{pmatrix} \quad (3)$$

Uniqueness of Interpolation

The left matrix is known as a **Vandermonde matrix**, denoted by $V(x_0, x_1, \ldots, x_{n-1})$, whose determinant is:

$$\prod_{0 \le j < k \le n-1} (x_k - x_j) \neq 0$$

Therefore, the solution of coefficients is:

$$a = V(x_0, x_1, \ldots, x_{n-1})^{-1} y$$

The LU decomposition algorithm takes $O(n^3)$ time to solve these equations.

Lagrange Formula

A fast algorithm for n-point interpolation is

$$A(x) = \sum_{k=0}^{n-1} y_k \frac{\prod_{j \neq k}(x - x_j)}{\prod_{j \neq k}(x_k - x_j)} \tag{4}$$

, which takes $\Theta(n^2)$ time.

Basic Idea of Multiplication

$$\boxed{\begin{matrix} a_0, a_1, \ldots, a_{n-1} \\ b_0, b_1, \ldots, b_{n-1} \end{matrix}} \xrightarrow{\theta(n^2)} \boxed{c_0, c_1, \ldots, c_{2n-2}}$$

$$\downarrow \theta(nlogn) \qquad\qquad \uparrow \theta(nlogn)$$

$$\boxed{\begin{matrix} A(\omega_{2n}^0), B(\omega_{2n}^0) \\ A(\omega_{2n}^1), B(\omega_{2n}^1) \\ \vdots \\ A(\omega_{2n}^{2n-1}), B(\omega_{2n}^{2n-1}) \end{matrix}} \xrightarrow{\theta(n)} \boxed{\begin{matrix} C(\omega_{2n}^0) \\ C(\omega_{2n}^1) \\ \vdots \\ C(\omega_{2n}^{2n-1}) \end{matrix}}$$

Theorem 2

The product of two polynomials of degree-bound n can computed in time $\theta(nlogn)$, with both input and output representations in coefficient form.

Complex Roots of Unity

Definition 4

A complex nth root of unity is a complex ω such that:

$$\omega^n = 1 \tag{5}$$

There are exactly n complex nth roots of unity: $e^{2\pi ik/n}$ for $k = 0, 1, \ldots, n-1$

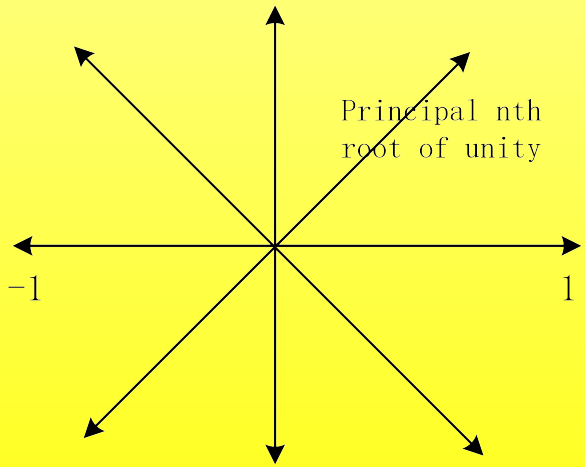
$$e^{iu} = \cos u + i \sin u \tag{6}$$

is called Eulers formula. And, $\omega_n = e^{2\pi i/n}$ is called the principal nth root of unity.

Additive Group

Property 3: The n complex nth roots of unity, form a group under multiplication, which is isomorphic to $(Z_n, +)$.

Example 1: Consider the values of $\omega_8^0, \omega_8^1, \ldots, \omega_8^7$, in the complex plane, where $\omega_8 = e^{2\pi i/8}$ is the principal 8th root of unity.



Properties of Complex Roots

Lemma 1: (Cancellation Lemma) $\forall n, k, d \in N$,we have

$$\omega_{dn}^{dk} = \omega_n^k$$

Corollary 1: For any even integer $n > 0$, we have

$$\omega_n^{n/2} = \omega_2 = -1$$

Lemma 2: (Halving Lemma) If $n > 0$ is even, then

$$(\omega_n^k)^2 = \omega_{n/2}^k, \quad that \ \ is, \quad (\omega_n^{k+n/2})^2 = (\omega_n^k)^2$$

Lemma 3: (Summation Lemma) For any integer $n \geq 1$ and nonnegative integer k not divisible by n,

$$\sum_{j=0}^{n-1} (\omega_n^k)^j = 0$$

Discrete Fourier Transform

Definition 5

Without loss of generality, assume that n is a power of 2, since a given degree-bound can always be raised. Given a polynomial $a = (a_0, a_1, ..., a_{n-1})$, define

$$y_k = A(\omega_n^k) = \sum_{j=0}^{n-1} a_j \omega_n^{kj} \tag{7}$$

Note: A(x)'s values at the n complex n-th roots of unity.

The vector $y = (y_0, y_1, ..., y_{n-1})$ is the Discrete Fourier Transform (DFT) of a, denoted by $y = DFT_n(a)$.

Fast Fourier Transform

Aim: To compute $DFT_n(a)$ in time $\theta(nlogn)$, as opposed to the $\theta(n^2)$ time of the straightforward method.

Idea: There are 3 steps:

1. $A(x) = A''(x^2) + xA'(x^2)$, where
$$\begin{cases} A''(x) = a_0 + a_2x + a_4x^2 + \ldots + a_{n-2}x^{n/2-1} \\ A'(x) = a_1 + a_3x + a_5x^2 + \ldots + a_{n-1}x^{n/2-1} \end{cases}$$
So that the problem of evaluating $A(x)$ at $\omega_n^0, \omega_n^1, \ldots \omega_n^{n-1}$ reduces to step 2.
2. Evaluate the degree-bound $n/2$ polynomials A' and A'' at points $(\omega_n^0)^2, (\omega_n^1)^2, \ldots, (\omega_n^{n-1})^2$.
3. Combine the results according to step 1.

Recursive FFT

Recursive FFT

By Halving Lemma, $\{(\omega_n^0)^2, (\omega_n^1)^2, ..., (\omega_n^{n-1})^2\} = \{\omega_{n/2}^0, \omega_{n/2}^1, ..., \omega_{n/2}^{n/2-1}\}$. This decomposition is the basis for the following recursive FFT algorithm.

RECURSIVE-FFT(a)
1: $n = a.length$
2: **if** $n == 1$ **then**
3: **return** a
4: $\omega_n = e^{2\pi i/n}$
5: $\omega = 1$
6: $\omega = 1$
7: $a'' = (a_0, a_2, ..., a_{n-2})$
8: $a' = (a_1, a_3, ..., a_{n-1})$
9: $y'' =$ RECURSIVE-FFT(a'')
10: $y' =$ RECURSIVE-FFT(a')
11: **for** $k = 0$ to $n/2-1$ **do**
12: $y_k = y_k'' + \omega y_k'$
13: $y_{k+n/2} = y_k'' - \omega y_k'$
14: $\omega = \omega\omega_n$
15: **return** y

Property 4: By divide-and-conquer method, the time cost of FFT is $T(n) = 2T(n/2) + \theta(n) = \theta(nlogn)$.

Interpolation

By (3), DFT can be written by $y = V_n a$, or

$$\begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_n & \omega_n^2 & \omega_n^3 & \cdots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \omega_n^6 & \cdots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \omega_n^9 & \cdots & \omega_n^{3(n-1)} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-2)} & \omega_n^{3(n-1)} & \cdots & \omega_n^{(n-1)(n-1)} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

where (k,j) entry of V_n is ω_n^{kj} for $j,k = 0,1,\ldots,n-1$. For the inverse operation of DFT_n , we write as $a = DFT_n^{-1}(y)$, we proceed by multiplying y by V_n^{-1}.

Interpolation

Theorem 3:

For $j,k = 0,1,\ldots,n-1$, the (j,k) entry of V_n^{-1} is ω_n^{-kj}/n.

Proof :

Consider the (j,j') entry of $V_n^{-1}V_n$:

$$[V_n^{-1}V_n]_{jj'} = \sum_{k=0}^{n-1} (\omega_n^{-kj}/n)(\omega_n^{kj'})$$
$$= \sum_{k=0}^{n-1} \omega_n^{k(j'-j)}/n$$

If $j' = j$, then $[V_n^{-1}V_n]_{jj'} = 1$, otherwise it is 0 by Summation Lemma.

Inverse DFT

Corollary 2 :

$a = DFT_n^{-1}(y)$ is given by

$$a_j = \frac{1}{n}\sum_{k=0}^{n-1} y_k \omega_n^{-kj} \tag{8}$$

DFT_n vs DFT_n^{-1}:

DFT_n	DFT_n^{-1}
$y_k = A(\omega_n^k)$ $= \sum_{j=0}^{n-1} a_j \omega_n^{kj}$	$a_j = \frac{1}{n}\sum_{k=0}^{n-1} y_k \omega_n^{-kj}$

Recall: Basic Idea of Multiplication

$$\boxed{\begin{matrix} a_0, a_1, \ldots, a_{n-1} \\ b_0, b_1, \ldots, b_{n-1} \end{matrix}} \xrightarrow{\theta(n^2)} \boxed{c_0, c_1, \ldots, c_{2n-2}}$$

$$\downarrow \theta(nlogn) \qquad\qquad \uparrow \theta(nlogn)$$

$$\boxed{\begin{matrix} A(\omega_{2n}^0), B(\omega_{2n}^0) \\ A(\omega_{2n}^1), B(\omega_{2n}^1) \\ \vdots \\ A(\omega_{2n}^{2n-1}), B(\omega_{2n}^{2n-1}) \end{matrix}} \xrightarrow{\theta(n)} \boxed{\begin{matrix} C(\omega_{2n}^0) \\ C(\omega_{2n}^1) \\ \vdots \\ C(\omega_{2n}^{2n-1}) \end{matrix}}$$

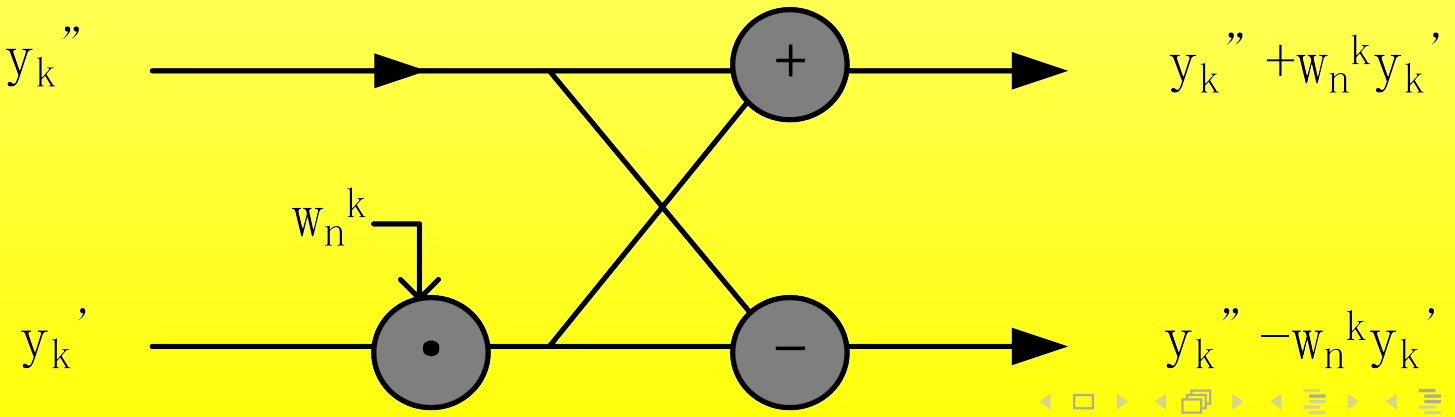
Theorem 2

The product of two polynomials of degree-bound n can computed in time $\theta(nlogn)$, with both input and output representations in coefficient form.

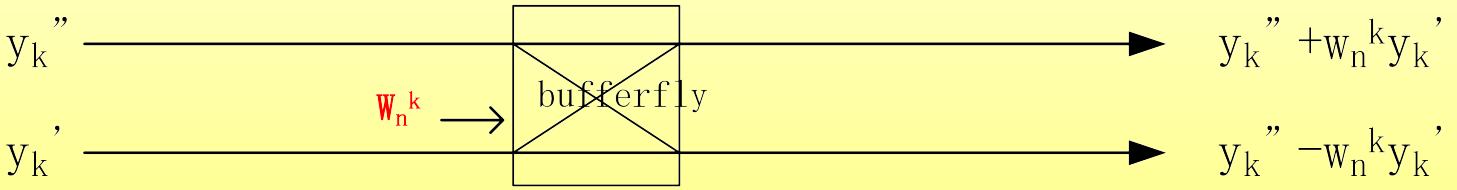
Efficient FFT Implementation

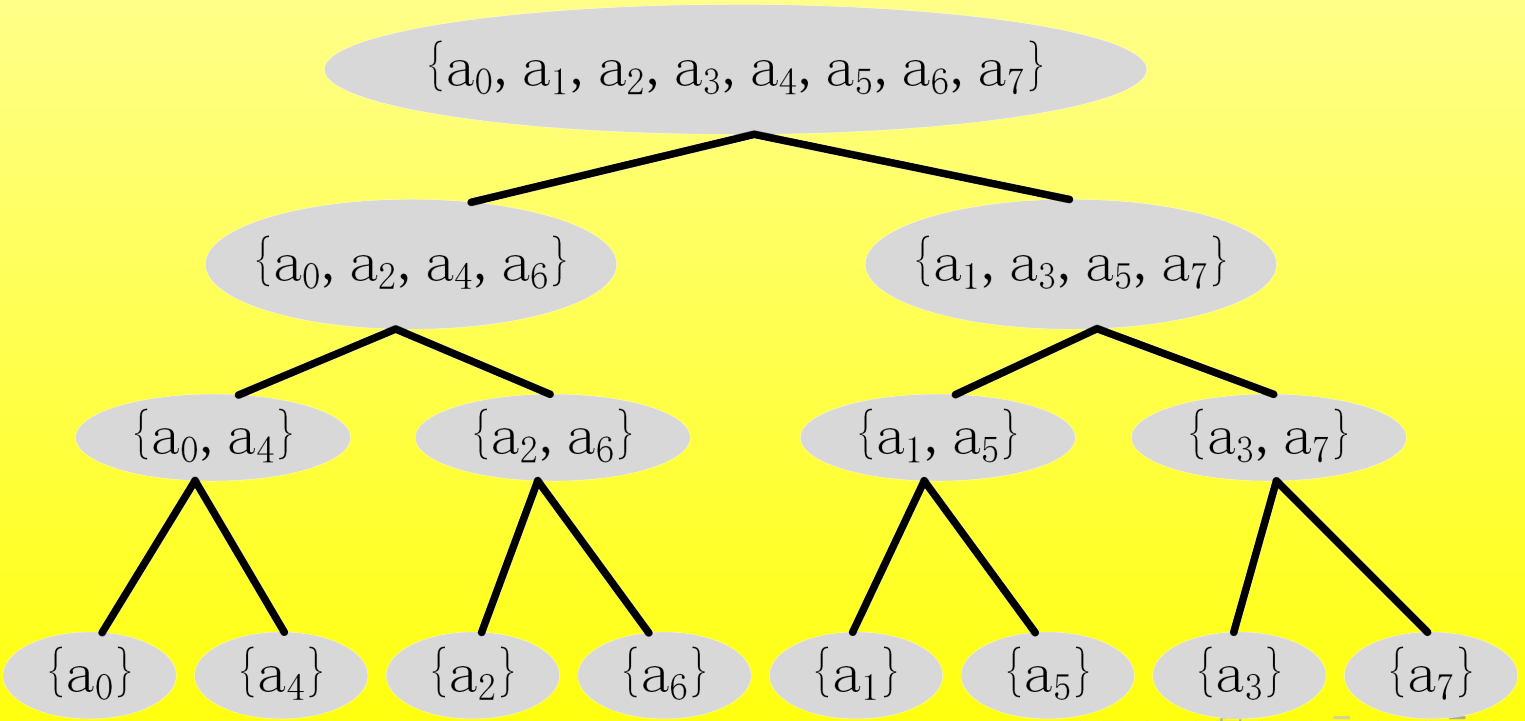
Common subexpression in compiler terminology

Original	Improvement
for $k = 0$ to $n/2 - 1$ do $y_k = y_k'' + \omega y_k'$ $y_{k+n/2} = y_k'' - \omega y_k'$ $\omega = \omega\omega_n$	for $k = 0$ to $n/2 - 1$ do $t = \omega y_k'$ $y_k = y_k'' + t$ $y_{k+n/2} = y_k'' - t$ $\omega = \omega\omega_n$

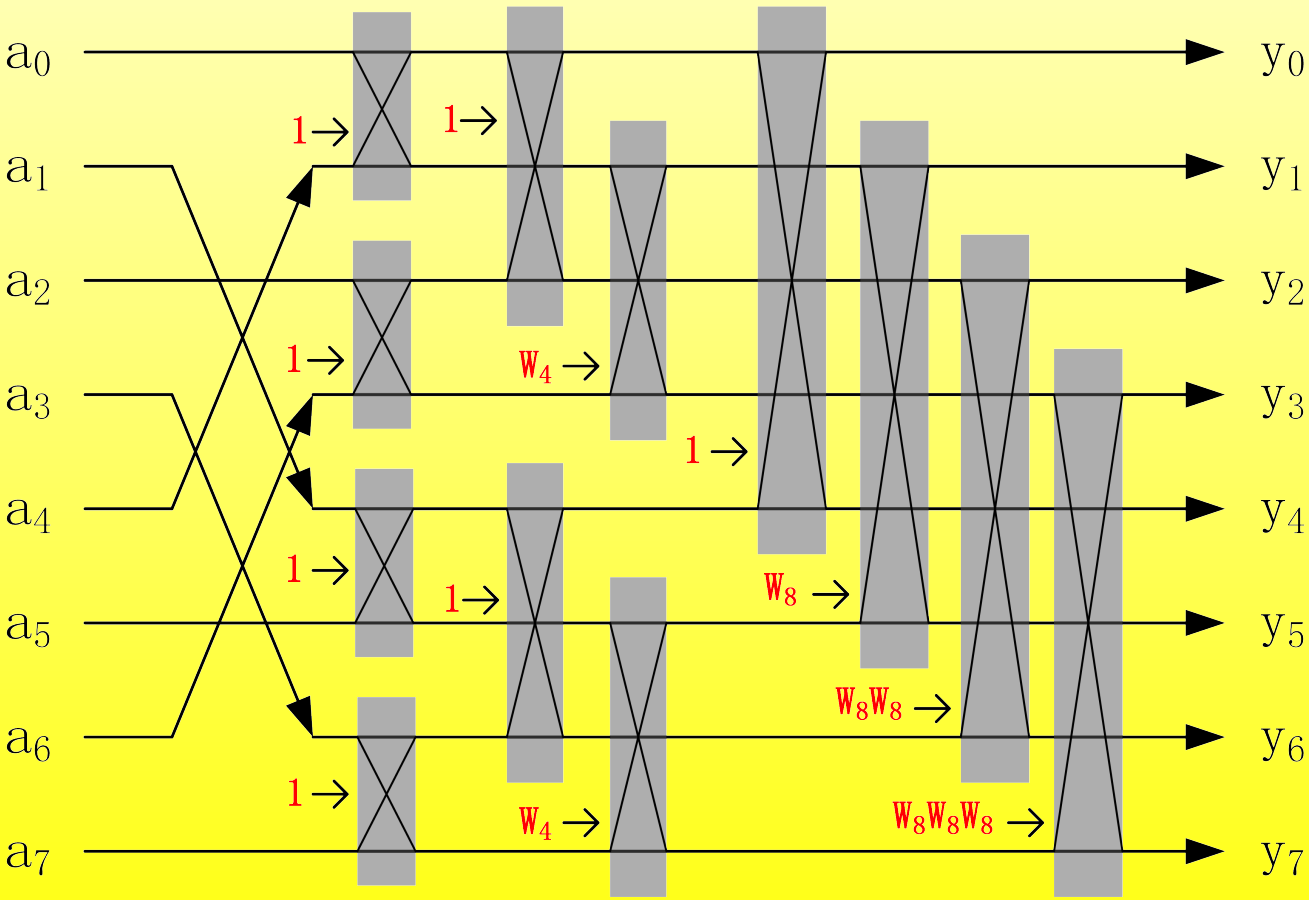


Butterfly Operation





Recursive FFT



Conclusions

- Fourier analysis is not limited to 1-dimensional data. It is widely used in image processing to analyze data in 2 or more dimensions.
- The history of FFT is traced as far back as C. F. Gauss in 1805.
- J. W. Cooley and J. W. Tukey are widely credited with devising the FFT in 1965.