Introduction to Algorithms: Lecture 3

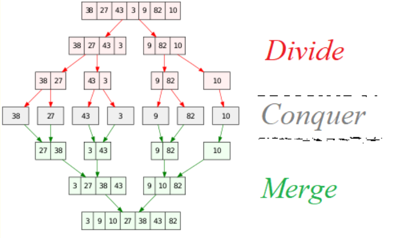
Xue Chen
xuechen1989@ustc.edu.cn
2025 spring in

Homework 2

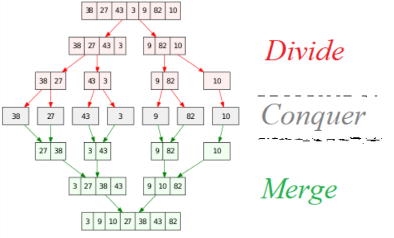
1. Experiment 1 will be out next week
2. TA office hours are in 3A103 in week 3

Outline

Intro: Recall MERGESORT



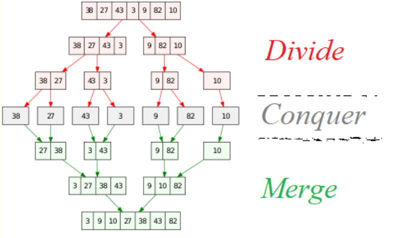
Intro: Recall MERGESORT



Divide & Conquer

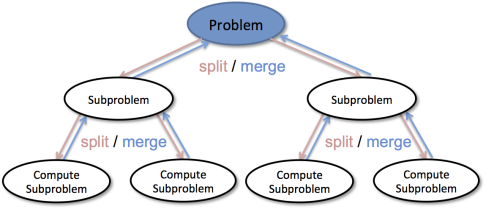
An old idea from 300 BC, now becomes an important algorithm design paradigm.

Intro: Recall MergeSort



Divide & Conquer

An old idea from 300 BC, now becomes an important algorithm design paradigm.

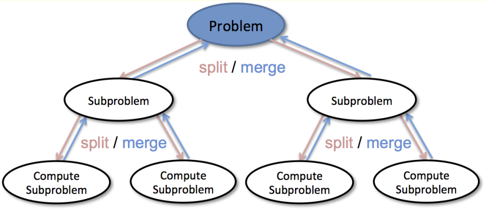


Paradigm

Divide & Conquer

3 steps:

1. Divide the problem into subproblems that are smaller instances.
2. Conquer subproblems by solving them recursively.
3. Merge solutions of subproblems into a solution of the original one.



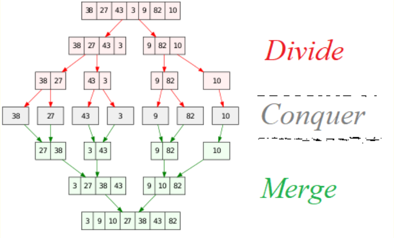
Overview

Recurrences

Let the algorithm divide a problem of size n into a subproblems of size $\lceil n/b \rceil$ and merge solutions in time $f(n)$.

$$\text{Running time } T(n) = a \cdot T(\lceil n/b \rceil) + f(n).$$

$a = b = 2$ and $f(n) = O(n)$ in MERGESORT



Overview

Next: Given $T(n) = a \cdot T(\lceil n/b \rceil) + f(n)$ where a and b are not necessarily 1 or 2, how to bound $T(n)$?

Applications

1. Strassen's algorithm for matrix multiplication
2. Fast Fourier transform
3. Closest pair
4. Binary search, Fibonacci number

Recurrences relation

Question

Given the relation $T(n) = a \cdot T(\lceil n/b \rceil) + f(n)$, how to compute $T(n)$?

Three methods:

1. Substitution method
2. Recursion tree method
3. Master method

Substitution Method

Example 1

$T(n) = 2T([n/2]) + n$.

Substitution Method

Example 1

$T(n) = 2T([n/2]) + n$.

Guess $T(n) \leqslant cn \log_2 n$ for $c = O(1)$ and apply induction.

Substitution Method

Example 1

$T(n) = 2T([n/2]) + n.$

Guess $T(n) \leqslant cn\log_2 n$ for $c = O(1)$ and apply induction.

1. Base case: $n = 1$, $T(1)$ is a constant.
2. Induction step:

$$\begin{aligned} T(n) &= 2c[n/2]\log_2[n/2] + n \\ &\leqslant 2c \cdot \frac{n}{2} \cdot (\log_2 n - 1) + n \\ &\leqslant cn\log_2 n. \end{aligned}$$

More about Substitution

Example 2

Question: What is $T(n) = 2T([n/2]) + 1$?

Remark

While it is simple to apply, guess the tight bound may be non-trivial.

More about Substitution

Example 2

Question: What is $T(n) = 2T([n/2]) + 1$?

If we guess $T(n) \leqslant cn$, $2 \cdot c \cdot \frac{n}{2} + 1 > cn$ for any even n.

Remark

While it is simple to apply, guess the tight bound may be non-trivial.

More about Substitution

Example 2

Question: What is $T(n) = 2T([n/2]) + 1$?

If we guess $T(n) \leqslant cn$, $2 \cdot c \cdot \frac{n}{2} + 1 > cn$ for any even n.
Solution: Strengthen the hypothesis to $T(n) \leqslant cn - 1$.

Remark

While it is simple to apply, guess the tight bound may be non-trivial.

Recursion tree method

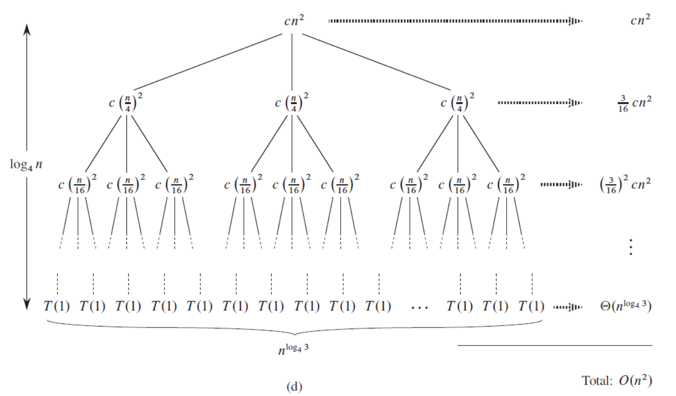
Example 3

$T(n) = 3T(n/4) + n^2$.

Recursion tree method

Example 3

$T(n) = 3T(n/4) + n^2$.

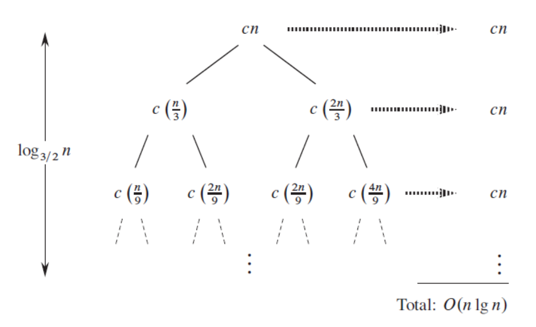


Example 4

$T(n) = T(n/3) + T(2n/3) + n.$

Example 4

$T(n) = T(n/3) + T(2n/3) + n$.



Master Method

Master Theorem

Let $a > 1$ and $b > 1$ be constants and $f(n)$ be a function.
$T(n) = a \cdot T(n/b) + f(n)$ has the following asymptotic bounds:

Master Method

Master Theorem

Let $a > 1$ and $b > 1$ be constants and $f(n)$ be a function.
$T(n) = a \cdot T(n/b) + f(n)$ has the following asymptotic bounds:

1. If $f(n) = \Theta(n^{\log_b a})$, $T(n) = \Theta(n^{\log_b a} \log n)$.

Master Method

Master Theorem

Let $a > 1$ and $b > 1$ be constants and $f(n)$ be a function.
$T(n) = a \cdot T(n/b) + f(n)$ has the following asymptotic bounds:

1. If $f(n) = \Theta(n^{\log_b a})$, $T(n) = \Theta(n^{\log_b a} \log n)$.
2. If $f(n) = O(n^{\log_b a - \Omega(1)})$, $T(n) = \Theta(n^{\log_b a})$.

Master Method

Master Theorem

Let $a > 1$ and $b > 1$ be constants and $f(n)$ be a function.
$T(n) = a \cdot T(n/b) + f(n)$ has the following asymptotic bounds:

1. If $f(n) = \Theta(n^{\log_b a})$, $T(n) = \Theta(n^{\log_b a} \log n)$.
2. If $f(n) = O(n^{\log_b a - \Omega(1)})$, $T(n) = \Theta(n^{\log_b a})$.
3. If $f(n) = \Omega(n^{\log_b a + \Omega(1)})$ and $a \cdot f(n/b) < (1 - \epsilon) \cdot f(n)$ for a constant $\epsilon > 0$, $T(n) = \Theta(f(n))$.

Master Method

Master Theorem

Let $a > 1$ and $b > 1$ be constants and $f(n)$ be a function.
$T(n) = a \cdot T(n/b) + f(n)$ has the following asymptotic bounds:

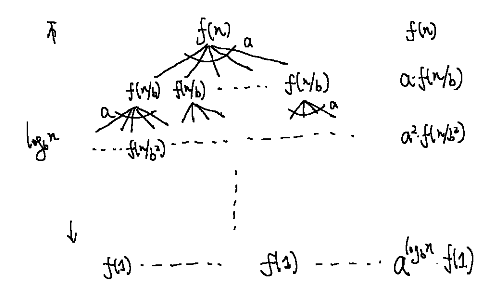
1. If $f(n) = \Theta(n^{\log_b a})$, $T(n) = \Theta(n^{\log_b a} \log n)$.
2. If $f(n) = O(n^{\log_b a - \Omega(1)})$, $T(n) = \Theta(n^{\log_b a})$.
3. If $f(n) = \Omega(n^{\log_b a + \Omega(1)})$ and $a \cdot f(n/b) < (1 - \epsilon) \cdot f(n)$ for a constant $\epsilon > 0$, $T(n) = \Theta(f(n))$.

All conditions $\Omega(1)$ and $a \cdot f(n/b) < f(n)$ are necessary.

Proof of Master THM

Proof by the recursion tree method:

- Case 1: each level contributes $n^{\log_b a}$.
- Case 2: the bottom level dominates.
- Case 3: the top node dominates.



Outline

Matrix Multiplication

Problem

Given two $n \times n$ matrices A and B, compute $C = AB$ where

$$C[i,j] = \sum_{k=1}^{n} A[i,k] \cdot B[k,j]. \tag{1}$$

Most fundamental problem in CS and math:

Matrix Multiplication

Problem

Given two $n \times n$ matrices A and B, compute $C = AB$ where

$$C[i,j] = \sum_{k=1}^{n} A[i,k] \cdot B[k,j]. \tag{1}$$

Most fundamental problem in CS and math:

1. $\omega :=$ the matrix multiplication constant *s.t.* time is $O(n^{\omega \pm o(1)})$.
2. rectangular matrices, matrix Inversion —- computing A^{-1} in $O(n^{\omega})$.

Matrix Multiplication

Problem

Given two $n \times n$ matrices A and B, compute $C = AB$ where

$$C[i,j] = \sum_{k=1}^{n} A[i,k] \cdot B[k,j]. \tag{1}$$

Most fundamental problem in CS and math:

1. $\omega :=$ the matrix multiplication constant *s.t.* time is $O(n^{\omega \pm o(1)})$.
2. rectangular matrices, matrix Inversion —- computing A^{-1} in $O(n^{\omega})$.
3. Solving linear equations and linear programming in time $O(n^{\omega \pm o(1)})$.
4. Graph problem: Max-flow, ...
5. Machine learning and data science: Linear regression, SVD & PCA, ...

First try

Motivating Question

It takes $O(n^3)$ time to compute C directly. Can we design faster algorithms?

On the other hand, it takes $\Omega(n^2)$ time to output C. So $\omega \in [2, 3]$.

First try

Motivating Question

It takes $O(n^3)$ time to compute C directly. Can we design faster algorithms?

On the other hand, it takes $\Omega(n^2)$ time to output C. So $\omega \in [2, 3]$.

Try divide and conquer

Say $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$ for four $\frac{n}{2} \times \frac{n}{2}$ matrices $A_{11}, A_{12}, A_{21}, A_{22}$ and $B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$ vice versa.

First try

Motivating Question

It takes $O(n^3)$ time to compute C directly. Can we design faster algorithms?

On the other hand, it takes $\Omega(n^2)$ time to output C. So $\omega \in [2, 3]$.

Try divide and conquer

Say $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$ for four $\frac{n}{2} \times \frac{n}{2}$ matrices $A_{11}, A_{12}, A_{21}, A_{22}$ and $B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$ vice versa.

$$\text{So } \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}.$$

Naive Method

$$\text{Consider } \begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \\ C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \\ C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22}. \end{aligned}$$

There are 8 submatrix-multiplications of size $\frac{n}{2} \times \frac{n}{2}$. The naive method will have

$$T(n) = 8T(n/2) + O(n^2) = O(n^3).$$

Naive Method

$$\text{Consider } \begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21} \\ C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22} \\ C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21} \\ C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22}. \end{aligned}$$

There are 8 submatrix-multiplications of size $\frac{n}{2} \times \frac{n}{2}$. The naive method will have

$$T(n) = 8T(n/2) + O(n^2) = O(n^3).$$

Key Question

Can we save some submatrix-multiplications?

Strassen's idea

$$\begin{vmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{vmatrix} = \begin{vmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{vmatrix} \begin{vmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{vmatrix}$$

$$\begin{aligned}
P_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
P_2 &= (A_{21} + A_{22}) * B_{11} \\
P_3 &= A_{11} * (B_{12} - B_{22}) \\
P_4 &= A_{22} * (B_{21} - B_{11}) \\
P_5 &= (A_{11} + A_{12}) * B_{22} \\
P_6 &= (A_{21} - A_{11}) * (B_{11} + B_{12}) \\
P_7 &= (A_{12} - A_{22}) * (B_{21} + B_{22})
\end{aligned}$$

$$\begin{aligned}
C_{11} &= P_1 + P_4 - P_5 + P_7 \\
C_{12} &= P_3 + P_5 \\
C_{21} &= P_2 + P_4 \\
C_{22} &= P_1 + P_3 - P_2 + P_6
\end{aligned}$$

Running Time

$T(n) = 7 \cdot T(n/2) + O(n^2) = O(n^{\log_2 7})$.

Discussion

1. While the idea is simple — reducing $8 \cdot T(n/2)$ to $7 \cdot T(n/2)$, highly non-trivial to find the solution.

Discussion

1. While the idea is simple — reducing $8 \cdot T(n/2)$ to $7 \cdot T(n/2)$, highly non-trivial to find the solution.
2. Fastest algorithm is in $O(n^{2.38})$, based on tensor products.
3. Matrix multiplication constant $\omega \in [2, 2.38]$ — a bold conjecture is $\omega = 2$ while many researchers believe $\omega > 2$.

Discussion

1. While the idea is simple — reducing $8 \cdot T(n/2)$ to $7 \cdot T(n/2)$, highly non-trivial to find the solution.
2. Fastest algorithm is in $O(n^{2.38})$, based on tensor products.
3. Matrix multiplication constant $\omega \in [2, 2.38]$ — a bold conjecture is $\omega = 2$ while many researchers believe $\omega > 2$.
4. O-constant is relatively large compared to the naive $O(n^3)$.
5. Matrix multiplication $\equiv$ matrix inversion [CLRS].

Discussion

1. While the idea is simple — reducing $8 \cdot T(n/2)$ to $7 \cdot T(n/2)$, highly non-trivial to find the solution.
2. Fastest algorithm is in $O(n^{2.38})$, based on tensor products.
3. Matrix multiplication constant $\omega \in [2, 2.38]$ — a bold conjecture is $\omega = 2$ while many researchers believe $\omega > 2$.
4. O-constant is relatively large compared to the naive $O(n^3)$.
5. Matrix multiplication $\equiv$ matrix inversion [CLRS].
6. For structured matrices, faster algorithms via Fast Fourier transform (FFT).

Outline

Introduction

Several ways to understand FFT:

1. Fast matrix multiplication (with a vector) for structured matrices V.

$$\begin{bmatrix} V_{1,1} & V_{1,2} & \dots & V_{1,n} \\ V_{2,1} & V_{2,2} & \dots & V_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ V_{n,1} & V_{n,2} & \dots & V_{n,n} \end{bmatrix} \cdot \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{bmatrix}$$

Introduction

Several ways to understand FFT:

1. Fast matrix multiplication (with a vector) for structured matrices V.

$$\begin{bmatrix} V_{1,1} & V_{1,2} & \dots & V_{1,n} \\ V_{2,1} & V_{2,2} & \dots & V_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ V_{n,1} & V_{n,2} & \dots & V_{n,n} \end{bmatrix} \cdot \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{bmatrix}$$

2. Fast polynomial interpolation.
3. Calculating the Fourier transform $\hat{\alpha}$.

Introduction

Several ways to understand FFT:

1. Fast matrix multiplication (with a vector) for structured matrices V.

$$\begin{bmatrix} V_{1,1} & V_{1,2} & \dots & V_{1,n} \\ V_{2,1} & V_{2,2} & \dots & V_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ V_{n,1} & V_{n,2} & \dots & V_{n,n} \end{bmatrix} \cdot \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \vdots \\ \alpha_n \end{bmatrix}$$

2. Fast polynomial interpolation.
3. Calculating the Fourier transform $\hat{\alpha}$.

Lots of applications: Algorithm design, data science, machine learning, signal processing, numerical computation, ...

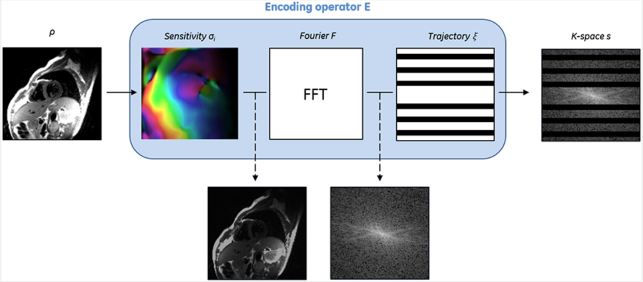
History

- Dates back to Gauss in 1805.
- Cooley and Tukey rediscovered it in 1965.
- Most important numerical algorithm of our life time — Gilbert Strang
- Top 10 Algorithms of 20th Century by IEEE.

History

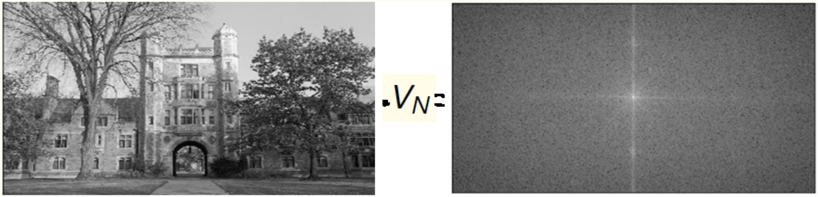
- Dates back to Gauss in 1805.
- Cooley and Tukey rediscovered it in 1965.
- Most important numerical algorithm of our life time — Gilbert Strang
- Top 10 Algorithms of 20th Century by IEEE.

Applications in MRI

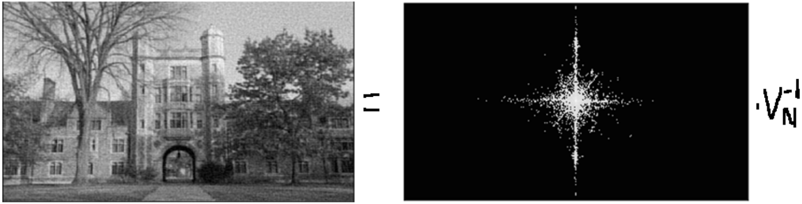


Applications in Data Compression

Many natural data sets are sparse on the frequency domain:



original image and its FT.

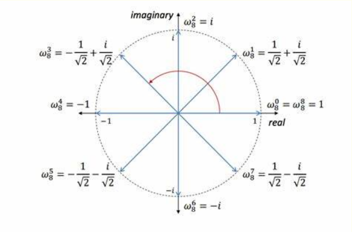


After sparsification, ...

Formal Definition

Complex roots of unity

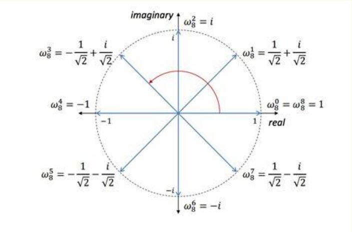
Given a 2-power N, let $\omega_N := e^{2\pi \mathbf{i}/N}$ s.t. $\omega_N^0, \omega_N, \ldots, \omega_N^{N-1}$ are roots of $x^N - 1 = 0$, called Nth roots of unity.



Formal Definition

Complex roots of unity

Given a 2-power N, let $\omega_N := e^{2\pi \mathbf{i}/N}$ s.t. $\omega_N^0, \omega_N, \ldots, \omega_N^{N-1}$ are roots of $x^N - 1 = 0$, called Nth roots of unity.



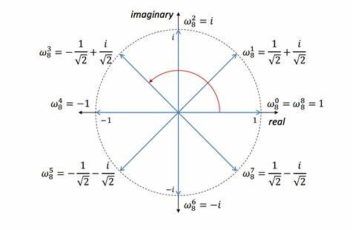
Define their Vandermonde matrix

$$V_N := \begin{bmatrix} \omega_N^{0\cdot 0} & \omega_N^{0\cdot 1} & \ldots & \omega_N^{0\cdot(N-1)} \\ \omega_N^{1\cdot 0} & \omega_N^{1\cdot 1} & \ldots & \omega_N^{1\cdot(N-1)} \\ \vdots & \vdots & \vdots & \vdots \\ \omega_N^{(N-1)\cdot 0} & \omega_N^{(N-1)\cdot 1} & \ldots & \omega_N^{(N-1)\cdot(N-1)} \end{bmatrix}$$

Formal Definition

Complex roots of unity

Given a 2-power N, let $\omega_N := e^{2\pi\mathbf{i}/N}$ s.t. $\omega_N^0, \omega_N, \ldots, \omega_N^{N-1}$ are roots of $x^N - 1 = 0$, called Nth roots of unity.



Define their Vandermonde matrix

$$V_N := \begin{bmatrix} \omega_N^{0\cdot 0} & \omega_N^{0\cdot 1} & \ldots & \omega_N^{0\cdot(N-1)} \\ \omega_N^{1\cdot 0} & \omega_N^{1\cdot 1} & \ldots & \omega_N^{1\cdot(N-1)} \\ \vdots & \vdots & \vdots & \vdots \\ \omega_N^{(N-1)\cdot 0} & \omega_N^{(N-1)\cdot 1} & \ldots & \omega_N^{(N-1)\cdot(N-1)} \end{bmatrix}$$

Problem

Given N and vector $\vec{\alpha} = (\alpha_1, \ldots, \alpha_N)$, how to compute $V_N \cdot \vec{\alpha}$ efficiently?

Overview

Main Result of FFT

FFT algorithm computes $V_N \cdot \vec{\alpha}$ in time $O(N \log N)$.

Overview

Main Result of FFT

FFT algorithm computes $V_N \cdot \vec{\alpha}$ in time $O(N \log N)$.

1. The trivial algorithm takes $O(N^2)$ time.
2. For any matrix $B \in \mathbb{R}^{N \times N}$, FFT takes $O(N^2 \log N)$ time to compute $V_N \cdot B$ or $B \cdot V_N$ (faster than general matrix multiplication).

Overview

Main Result of FFT

FFT algorithm computes $V_N \cdot \vec{\alpha}$ in time $O(N \log N)$.

1. The trivial algorithm takes $O(N^2)$ time.
2. For any matrix $B \in \mathbb{R}^{N \times N}$, FFT takes $O(N^2 \log N)$ time to compute $V_N \cdot B$ or $B \cdot V_N$ (faster than general matrix multiplication).
3. A fundamental question: Is this the fastest method?
4. Many extensions: Other Vandermonde matrices, Toeplitz matrices, convolution, Hadamard & cosine transformation, . . .

Overview

Main Result of FFT

FFT algorithm computes $V_N \cdot \vec{\alpha}$ in time $O(N \log N)$.

1. The trivial algorithm takes $O(N^2)$ time.
2. For any matrix $B \in \mathbb{R}^{N \times N}$, FFT takes $O(N^2 \log N)$ time to compute $V_N \cdot B$ or $B \cdot V_N$ (faster than general matrix multiplication).
3. A fundamental question: Is this the fastest method?
4. Many extensions: Other Vandermonde matrices, Toeplitz matrices, convolution, Hadamard & cosine transformation, ...

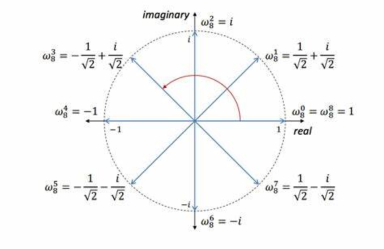
Before we describe FFT, review basic facts about the roots of unity $\omega_N := e^{2\pi \mathbf{i}/N}$

Properties of ω

Property 1

Let $\omega_N := e^{2\pi \mathbf{i}/N}$ for a 2-power N.

1. $\omega_N^N = 1$. So $\omega_N^j = \omega_N^{j \bmod N}$ and focus on remainder.

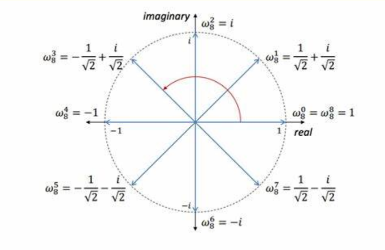


Properties of ω

Property 1

Let $\omega_N := e^{2\pi\mathbf{i}/N}$ for a 2-power N.

1. $\omega_N^N = 1$. So $\omega_N^j = \omega_N^{j \bmod N}$ and focus on remainder.
2. $\omega_N^{N/2} = -1$.
3. ω_N^2 is the $\frac{N}{2}$-root of unity.



Properties of ω (II)

Plan: compute $y = V_N \cdot \alpha$ by divide & conquer — consider ω_N and V_N in terms of $\omega_{N/2}$ and $V_{N/2}$.

1. $\omega_{N/2} = \omega_N^2$.

Properties of ω (II)

Plan: compute $y = V_N \cdot \alpha$ by divide & conquer — consider ω_N and V_N in terms of $\omega_{N/2}$ and $V_{N/2}$.

1. $\omega_{N/2} = \omega_N^2$.
2. V_N is constituted by 4 submatrices $\approx V_{N/2}$.

$$V_N = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 & 1 & \cdots & 1 \\ 1 & \omega_N^{1\cdot 1} & \omega_N^{1\cdot 2} & \cdots & \omega_N^{1\cdot(\frac{N}{2}-1)} & \omega_N^{1\cdot\frac{N}{2}} & \cdots & \omega_N^{1\cdot(N-1)} \\ 1 & \omega_N^{2\cdot 1} & \omega_N^{2\cdot 2} & \cdots & \omega_N^{2\cdot(\frac{N}{2}-1)} & \omega_N^{2\cdot\frac{N}{2}} & \cdots & \omega_N^{2\cdot(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \\ 1 & \omega_N^{(N-2)\cdot 1} & \omega_N^{(N-2)\cdot 2} & \cdots & \omega_N^{(N-2)\cdot(\frac{N}{2}-1)} & \omega_N^{(N-2)\cdot\frac{N}{2}} & \cdots & \omega_N^{(N-2)\cdot(N-1)} \\ 1 & \omega_N^{(N-1)\cdot 1} & \omega_N^{(N-1)\cdot 2} & \cdots & \omega_N^{(N-1)\cdot(\frac{N}{2}-1)} & \omega_N^{(N-1)\cdot\frac{N}{2}} & \cdots & \omega_N^{(N-1)\cdot(N-1)} \end{bmatrix}$$

Properties of ω (II)

Plan: compute $y = V_N \cdot \alpha$ by divide & conquer — consider ω_N and V_N in terms of $\omega_{N/2}$ and $V_{N/2}$.

1. $\omega_{N/2} = \omega_N^2$.
2. V_N is constituted by 4 submatrices $\approx V_{N/2}$.

$$V_N = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 & 1 & \cdots & 1 \\ 1 & \omega_N^{1\cdot 1} & \omega_N^{1\cdot 2} & \cdots & \omega_N^{1\cdot(\frac{N}{2}-1)} & \omega_N^{1\cdot\frac{N}{2}} & \cdots & \omega_N^{1\cdot(N-1)} \\ 1 & \omega_N^{2\cdot 1} & \omega_N^{2\cdot 2} & \cdots & \omega_N^{2\cdot(\frac{N}{2}-1)} & \omega_N^{2\cdot\frac{N}{2}} & \cdots & \omega_N^{2\cdot(N-1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \\ 1 & \omega_N^{(N-2)\cdot 1} & \omega_N^{(N-2)\cdot 2} & \cdots & \omega_N^{(N-2)\cdot(\frac{N}{2}-1)} & \omega_N^{(N-2)\cdot\frac{N}{2}} & \cdots & \omega_N^{(N-2)\cdot(N-1)} \\ 1 & \omega_N^{(N-1)\cdot 1} & \omega_N^{(N-1)\cdot 2} & \cdots & \omega_N^{(N-1)\cdot(\frac{N}{2}-1)} & \omega_N^{(N-1)\cdot\frac{N}{2}} & \cdots & \omega_N^{(N-1)\cdot(N-1)} \end{bmatrix}$$

Observation:

(1) Red submatrix $V_N(\text{odd}, [0, \ldots, \frac{N}{2}-1])$ is $V_{N/2}$, same for $V_N(\text{odd}, [\frac{N}{2}, \ldots, N-1])$.
(2) Blue submatrix $V_N(\text{even}, [0, \ldots, \frac{N}{2}-1])$ is $V_{N/2} \cdot \mathrm{diag}[1, \omega_N, \omega_N^2, \ldots, \omega_N^{N/2-1}]$.

We shall exploit its symmetric properties to speed-up $V_N \cdot \vec{\alpha}$.

Idea (I):

Since $V[i,j] = \omega^{i\cdot j}$, rewrite the calculation $y = V \cdot \vec{\alpha}$ as

$$y[i] = \sum_{j=0}^{N-1} \omega^{i\cdot j} \cdot \alpha[j] \text{ for } i = 0, 1, \ldots, N-1. \qquad (*)$$

Consider the toy examples of $y[0]$ and $y[N/2]$: By the definition,

$$y[0] = \sum_{j=0}^{N-1} \omega^{0\cdot j} \cdot \alpha[j] = \sum_{j=0}^{N-1} \alpha[j].$$

$$y[N/2] = \sum_{j=0}^{N-1} \omega^{N/2\cdot j} \cdot \alpha[j] = \sum_{j=0}^{N-1} (-1)^j \cdot \alpha[j].$$

Idea (I):

Since $V[i,j] = \omega^{i \cdot j}$, rewrite the calculation $y = V \cdot \vec{\alpha}$ as

$$y[i] = \sum_{j=0}^{N-1} \omega^{i \cdot j} \cdot \alpha[j] \text{ for } i = 0, 1, \ldots, N-1. \tag{*}$$

Consider the toy examples of $y[0]$ and $y[N/2]$: By the definition,

$$y[0] = \sum_{j=0}^{N-1} \omega^{0 \cdot j} \cdot \alpha[j] = \sum_{j=0}^{N-1} \alpha[j].$$

$$y[N/2] = \sum_{j=0}^{N-1} \omega^{N/2 \cdot j} \cdot \alpha[j] = \sum_{j=0}^{N-1} (-1)^j \cdot \alpha[j].$$

Question: Do we need $2N$ steps to compute them?

Idea (II):

For general $i < N/2$,

$$y[i] = \sum_{j=0}^{N-1} \omega^{i \cdot j} \cdot \alpha[j]$$

$$= \underbrace{\sum_{\text{even } j} \omega^{i \cdot j} \cdot \alpha[j]}_{S_0(i)} + \underbrace{\sum_{\text{odd } j} \omega^{i \cdot j} \cdot \alpha[j]}_{S_1(i)}.$$

Idea (II):

For general $i < N/2$,

$$y[i] = \sum_{j=0}^{N-1} \omega^{i \cdot j} \cdot \alpha[j]$$

$$= \underbrace{\sum_{\text{even } j} \omega^{i \cdot j} \cdot \alpha[j]}_{S_0(i)} + \underbrace{\sum_{\text{odd } j} \omega^{i \cdot j} \cdot \alpha[j]}_{S_1(i)}.$$

$$y[N/2 + i] = \sum_{j=0}^{N-1} \omega^{(i+N/2) \cdot j} \cdot \alpha[j]$$

$$= \sum_{\text{even } j} \omega^{(i+N/2) \cdot j} \cdot \alpha[j] + \sum_{\text{odd } j} \omega^{(i+N/2) \cdot j} \cdot \alpha[j]$$

Idea (II):

For general $i < N/2$,

$$y[i] = \sum_{j=0}^{N-1} \omega^{i \cdot j} \cdot \alpha[j]$$
$$= \underbrace{\sum_{\textcolor{red}{even}\ j} \omega^{i \cdot j} \cdot \alpha[j]}_{S_0(i)} + \underbrace{\sum_{\textcolor{red}{odd}\ j} \omega^{i \cdot j} \cdot \alpha[j]}_{S_1(i)}.$$

$$y[N/2 + i] = \sum_{j=0}^{N-1} \omega^{(i+N/2) \cdot j} \cdot \alpha[j]$$
$$= \sum_{\textcolor{red}{even}\ j} \omega^{(i+N/2) \cdot j} \cdot \alpha[j] + \sum_{\textcolor{red}{odd}\ j} \omega^{(i+N/2) \cdot j} \cdot \alpha[j]$$
$$= S_0(i) + S_1(i) \cdot \omega^{N/2} = S_0(i) - S_1(i).$$

This saves the whole calculation by a factor of 2.

Idea (III):

Question

Can we push this idea more?

Idea (III):

Question

Can we push this idea more?

Back to $y[i] = \underbrace{\sum_{\text{even } j} \omega^{i \cdot j} \cdot \alpha[j]}_{S_0(i)} + \underbrace{\sum_{\text{odd } j} \omega^{i \cdot j} \cdot \alpha[j]}_{S_1(i)},$

$$S_0(i) = \sum_{\text{even } j} \omega^{2 \cdot i \cdot j/2} \cdot \alpha[j].$$

Observation

1. ω^2 is the $\frac{N}{2}$th root of unity
   s.t. $S_0 = V_{N/2} \cdot (\alpha[0], \alpha[2], \ldots, \alpha[N-2]) \in \mathbb{R}^{N/2}$.

Idea (III):

Question

Can we push this idea more?

Back to $y[i] = \underbrace{\sum_{\text{even } j} \omega^{i \cdot j} \cdot \alpha[j]}_{S_0(i)} + \underbrace{\sum_{\text{odd } j} \omega^{i \cdot j} \cdot \alpha[j]}_{S_1(i)}$,

$$S_0(i) = \sum_{\text{even } j} \omega^{2 \cdot i \cdot j/2} \cdot \alpha[j].$$

Observation

1. ω^2 is the $\frac{N}{2}$th root of unity s.t. $S_0 = V_{N/2} \cdot (\alpha[0], \alpha[2], \ldots, \alpha[N-2]) \in \mathbb{R}^{N/2}$.
2. $S_1(i) = \omega^i \cdot \sum_{\text{odd } j} \omega^{i \cdot 2 \cdot (j-1)/2} \cdot \alpha[j]$ where sum is in $V_{N/2} \cdot (\alpha[1], \alpha[3], \ldots, \alpha[N-1])$.

Algorithm Description

Algorithm FFT($N, \alpha[0], \ldots, \alpha[N-1]$)

if $N = 1$ **then**
    Return $\alpha[0]$
$S_0 \leftarrow FFT(N/2, \alpha[0], \alpha[2], \ldots, \alpha[N-2])$
$S_1' \leftarrow FFT(N/2, \alpha[1], \alpha[3], \ldots, \alpha[N-1])$ $//S_1(i) = S_1'(i) \cdot \omega^i$
$\omega = e^{2\pi\mathbf{i}/N}$
for $i = 0$ to $N/2 - 1$ **do**
    $y[i] = S_0[i] + S_1'[i] \cdot \omega^i$
    $y[i + N/2] = S_0[i] - S_1'[i] \cdot \omega^i$
Return y

Analysis

Correctness

Follows from the above discussion.

Analysis

Correctness

Follows from the above discussion.

Time Complexity

Let $T(N)$ denote the time to compute $\alpha \in \mathbb{R}^N$ s.t.

$$T(N) = 2T(N/2) + O(N)$$

Analysis

Correctness

Follows from the above discussion.

Time Complexity

Let $T(N)$ denote the time to compute $\alpha \in \mathbb{R}^N$ s.t.

$$T(N) = 2T(N/2) + O(N) = O(N \log N).$$

Next: Extensions.

Polynomial Interpolation

$$V_N \cdot \vec{\alpha} \text{ is } \begin{bmatrix} \omega_N^{0\cdot 0} & \omega_N^{0\cdot 1} & \dots & \omega_N^{0\cdot(N-1)} \\ \omega_N^{1\cdot 0} & \omega_N^{1\cdot 1} & \dots & \omega_N^{1\cdot(N-1)} \\ \vdots & \vdots & \vdots & \vdots \\ \omega_N^{(N-1)\cdot 0} & \omega_N^{(N-1)\cdot 1} & \dots & \omega_N^{(N-1)\cdot(N-1)} \end{bmatrix} \cdot \begin{bmatrix} \alpha[0] \\ \alpha[1] \\ \vdots \\ \alpha[N-1] \end{bmatrix}$$

Fast polynomial evaluation

$y = V_N \cdot \vec{\alpha}$ is the evaluation of degree-$(N-1)$ polynomial $p(x) = \sum_{j=0}^{N-1} \alpha[j] \cdot x^j$ at roots of unity $\omega_N^0, \dots, \omega_N^{N-1}$.

Polynomial Interpolation

$$V_N \cdot \vec{\alpha} \text{ is } \begin{bmatrix} \omega_N^{0\cdot 0} & \omega_N^{0\cdot 1} & \dots & \omega_N^{0\cdot(N-1)} \\ \omega_N^{1\cdot 0} & \omega_N^{1\cdot 1} & \dots & \omega_N^{1\cdot(N-1)} \\ \vdots & \vdots & \vdots & \vdots \\ \omega_N^{(N-1)\cdot 0} & \omega_N^{(N-1)\cdot 1} & \dots & \omega_N^{(N-1)\cdot(N-1)} \end{bmatrix} \cdot \begin{bmatrix} \alpha[0] \\ \alpha[1] \\ \vdots \\ \alpha[N-1] \end{bmatrix}$$

Fast polynomial evaluation

$y = V_N \cdot \vec{\alpha}$ is the evaluation of degree-$(N-1)$ polynomial $p(x) = \sum_{j=0}^{N-1} \alpha[j] \cdot x^j$ at roots of unity $\omega_N^0, \dots, \omega_N^{N-1}$.

Question: Discrete Fourier transform on $\mathbb{Z}_N$?

Inverse FFT

Another useful fact: $V_N^{-1} = \frac{1}{N} \cdot \overline{V_N}$.

$$V_N := \begin{bmatrix} \omega_N^{0\cdot 0} & \omega_N^{0\cdot 1} & \dots & \omega_N^{0\cdot(N-1)} \\ \omega_N^{1\cdot 0} & \omega_N^{1\cdot 1} & \dots & \omega_N^{1\cdot(N-1)} \\ \vdots & \vdots & \vdots & \vdots \\ \omega_N^{(N-1)\cdot 0} & \omega_N^{(N-1)\cdot 1} & \dots & \omega_N^{(N-1)\cdot(N-1)} \end{bmatrix}$$

Inverse FFT

Another useful fact: $V_N^{-1} = \frac{1}{N} \cdot \overline{V_N}$.

$$V_N := \begin{bmatrix} \omega_N^{0\cdot 0} & \omega_N^{0\cdot 1} & \dots & \omega_N^{0\cdot(N-1)} \\ \omega_N^{1\cdot 0} & \omega_N^{1\cdot 1} & \dots & \omega_N^{1\cdot(N-1)} \\ \vdots & \vdots & \vdots & \vdots \\ \omega_N^{(N-1)\cdot 0} & \omega_N^{(N-1)\cdot 1} & \dots & \omega_N^{(N-1)\cdot(N-1)} \end{bmatrix}$$

Property 2

Since $\omega_N^0, \dots, \omega_N^{N-1}$ are symmetric, $\sum_j \omega_N^{j\cdot k} = 0$ for any $k \in [1, \dots, N-1]$; otherwise the sum is N for $k = 0$.

Inverse FFT

Another useful fact: $V_N^{-1} = \frac{1}{N} \cdot \overline{V_N}$.

$$V_N := \begin{bmatrix} \omega_N^{0\cdot 0} & \omega_N^{0\cdot 1} & \dots & \omega_N^{0\cdot (N-1)} \\ \omega_N^{1\cdot 0} & \omega_N^{1\cdot 1} & \dots & \omega_N^{1\cdot (N-1)} \\ \vdots & \vdots & \vdots & \vdots \\ \omega_N^{(N-1)\cdot 0} & \omega_N^{(N-1)\cdot 1} & \dots & \omega_N^{(N-1)\cdot (N-1)} \end{bmatrix}$$

Property 2

Since $\omega_N^0, \dots, \omega_N^{N-1}$ are symmetric, $\sum_j \omega_N^{j\cdot k} = 0$ for any $k \in [1, \dots, N-1]$; otherwise the sum is N for $k = 0$.

Corollary: $V_N^{-1} \cdot \vec{\alpha}$ is in time $O(N \log N)$.

Convolution

Definition

Given two vectors $(\alpha[0], \ldots, \alpha[N-1])$ and $(\beta[0], \ldots, \beta[N-1])$, their convolution is

$$(\alpha * \beta)[k] = \sum_{\ell=0}^{N-1} \alpha[\ell] \cdot \beta\left[(k-\ell) \mod N\right].$$

Convolution

Definition

Given two vectors $(\alpha[0], \ldots, \alpha[N-1])$ and $(\beta[0], \ldots, \beta[N-1])$, their convolution is

$$(\alpha * \beta)[k] = \sum_{\ell=0}^{N-1} \alpha[\ell] \cdot \beta[(k-\ell) \mod N].$$

Key fact: $V_N(\alpha * \beta) = (V_N\alpha) \cdot (V_N\beta)$ where $\cdot$ denotes dot-product $(v \cdot u)[i] = v[i]u[i]$.

So FFT computes convolution in time $O(N \log N)$.

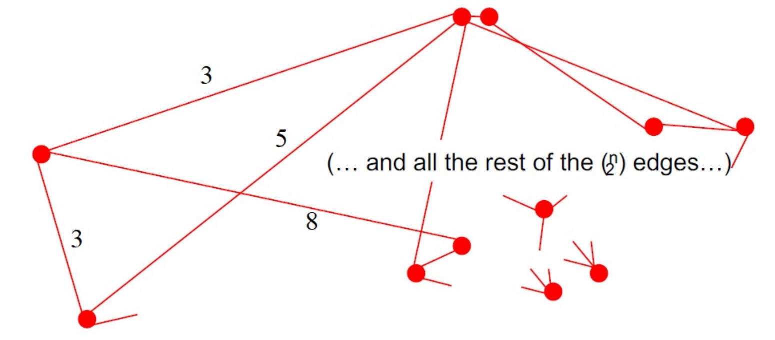
Summary about FFT

1. Elegant and Efficient!
2. Fundamental algorithm in machine learning, data science, signal processing, ...
3. Lots of variations: cryptography, compressed sensing,...

Outline

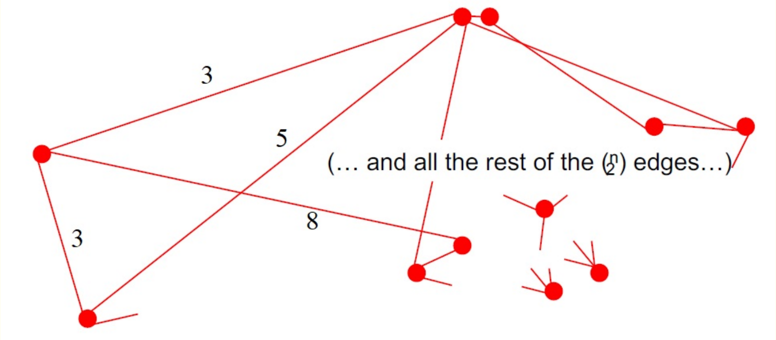
Introduction

Basic question: Given n points and a metric/distance between them, find the closest pair.



Introduction

Basic question: Given n points and a metric/distance between them, find the closest pair.



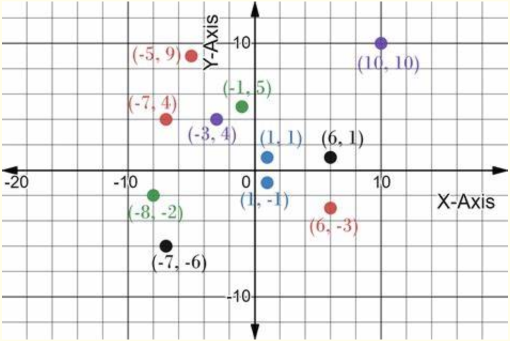
Lower bound

If the metric is arbitrary, $\Omega(n^2)$ because it must look at all pairswise distances.

Our problem in 2-dimension

Description

Given n points in the plane $\mathbb{R}^2$, find a pair with smallest Euclidean distance between them.



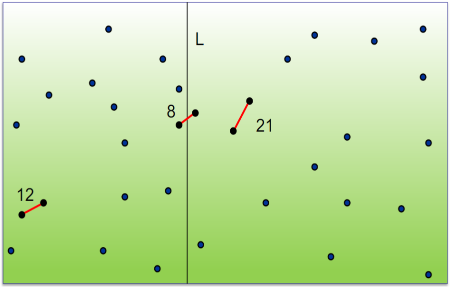
Fundamental geometric primitive: Graphics, computer vision, traffic control, nearest neighbor, Euclidean MST, . . .

Main Result

Theorem

Divide & conquer solves the closest pair problem in 2-dimension in time $O(n \log n)$.

1. The trivial algorithm is in time $O(n^2)$.



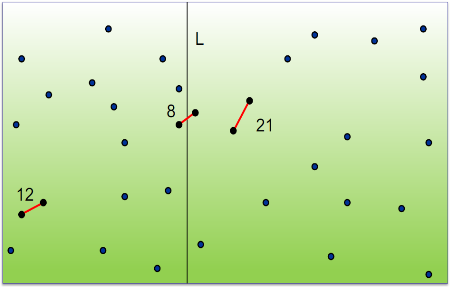
Main Result

Theorem

Divide & conquer solves the closest pair problem in 2-dimension in time $O(n \log n)$.

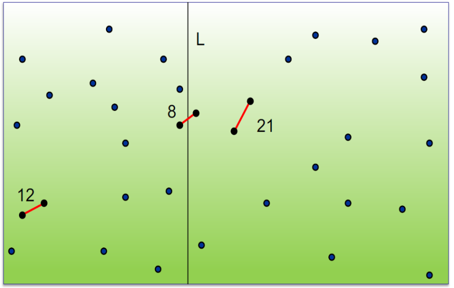
1. The trivial algorithm is in time $O(n^2)$.
2. For ease of exposition, assume no two points have same x or y coordinates and present $O(n \log^2 n)$-time algorithm.

Let us apply the divide & conquer paradigm.



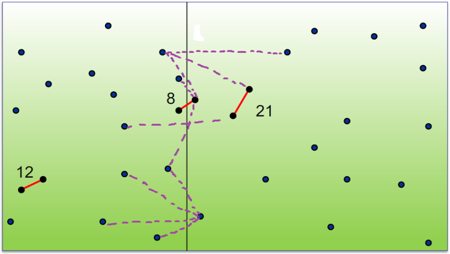
Three Steps

1. Divide: Draw a line with $n/2$ points on each side
2. Conquer: Find closest pair on each side



Three Steps

1. Divide: Draw a line with $n/2$ points on each side
2. Conquer: Find closest pair on each side
3. Merge: Find closest pair overall — check all cross pairs?



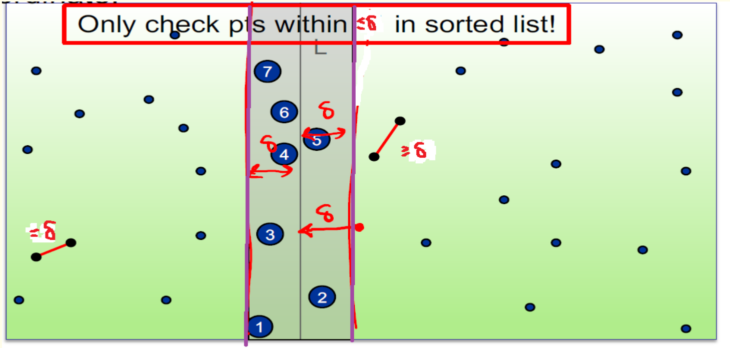
$n^2/4$ cross pairs

Key Observation

No need to check all cross pairs.

Key Idea

OBS: Let δ be the shortest distance from subproblems. Suffices to consider points within δ to L.



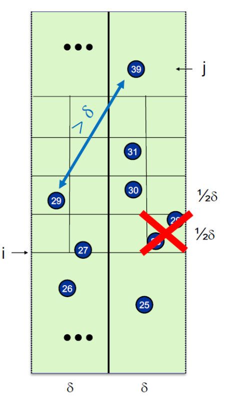
Next question: How many cross pairs in the 2δ-strip?

Almost 1D Problem

Claim:

No two points lie in the same $\frac{\delta}{2} \times \frac{\delta}{2}$ box. So there are at most 11 pairs to check for each point.

Proof: Such pair will have distance $\leqslant \delta/\sqrt{2}$. This contradicts with the definition of δ.



Pseudo-Code

```
Closest-Pair(p₁, …, pₙ) {
   if(n <= ??) return ??

   Compute separation line L such that half the points
   are on one side and half on the other side.

   δ₁ = Closest-Pair(left half)
   δ₂ = Closest-Pair(right half)
   δ  = min(δ₁, δ₂)

   Delete all points further than δ from separation line L

   Sort remaining points p[1]…p[m] by y-coordinate.

   for i = 1..m
      for k = 1…11
       if i+k <= m
          δ = min(δ, distance(p[i], p[i+k]));

   return δ.
}
```

Running Time

Recurrence Relation

$O(n \log^2 n)$: $T(n) = 2T(n/2) + O(n \log n)$ since we need to sort points in the strip.

Running Time

Recurrence Relation

$O(n \log^2 n)$: $T(n) = 2T(n/2) + O(n \log n)$ since we need to sort points in the strip.

$O(n \log n)$-time algorithm: Sort them according to y at the beginning such that we could compute the closest pair in the strip without sorting.

Outline

Binary Search

Recall INSERTIONSORT:

Problem

Given $A[1] \leqslant A[2] \leqslant \cdots \leqslant A[n]$ and k, find # elements in A less than k.

Binary Search

Recall InsertionSort:

Problem

Given $A[1] \leqslant A[2] \leqslant \cdots \leqslant A[n]$ and k, find # elements in A less than k.

Instead of enumerating j from 1 to n, we can apply a binary search.

$$A[1] \leq A[2] \leq A[3] \leq \cdots \leq A\left[\frac{n}{2}\right] \leq A\left[\frac{n}{2}+1\right] \leq \cdots \leq A[n] \qquad k$$

Binary Search

Recall INSERTIONSORT:

Problem

Given $A[1] \leqslant A[2] \leqslant \cdots \leqslant A[n]$ and k, find # elements in A less than k.

Instead of enumerating j from 1 to n, we can apply a binary search.

$$A[1] \leq A[2] \leq A[3] \leq \cdots \leq A\left[\frac{n}{2}\right] \leq A\left[\frac{n}{2}+1\right] \leq \cdots \leq A[n] \qquad k$$

Two cases depending on $k \geqslant A\left[\frac{n}{2}\right]$ or not:

$k > A\left[\frac{n}{2}\right]$

$$A[1] \leq A[2] \leq A[3] \leq \cdots \leq A\left[\frac{n}{2}\right] \leq A\left[\frac{n}{2}+1\right] \leq \cdots \leq A[n] \qquad k$$

$k \leq A\left[\frac{n}{2}\right]$

Pseudo-code

function BinarySearch(ℓ, r, k)
    if $\ell = r$ **then**
        Return $(A[\ell] < k)$
    $j = [(\ell + r)/2]$
    if $A[j] \leqslant k$ **then**
        Return BinarySearch(ℓ, j, k)
    else
        Return $j - \ell + 1$+BinarySearch($j + 1, r, k$)

Pseudo-code

```
function BinarySearch(ℓ, r, k)
    if ℓ = r then
        Return (A[ℓ] < k)
    j = [(ℓ + r)/2]
    if A[j] ⩽ k then
        Return BinarySearch(ℓ, j, k)
    else
        Return j − ℓ + 1+BinarySearch(j + 1, r, k)
```

Running time: $T(n) = T(n/2) + O(1) = O(\log n)$.
Also could be implemented via a while loop.

Extensions

Question

How to compute a^n in time $O(\log n)$?

Assume all integer-operations are in $O(1)$

Extensions

Question

How to compute a^n in time $O(\log n)$?

Assume all integer-operations are in $O(1)$

function SQUARING(a, n)
    if $n = 1$ **then**
        Return a
    else
        Return SQUARING$(a, [n/2])^2 \cdot a^{n \mod 2}$

Fibonacci Numbers

Question

Can we extend this idea to compute $F_n = F_{n-1} + F_{n-2}$?

Fibonacci Numbers

Question

Can we extend this idea to compute $F_n = F_{n-1} + F_{n-2}$?

While $F_n \approx (\frac{1+\sqrt{5}}{2})^n/\sqrt{5}$, it is very unstable.

Matrix

$$\begin{bmatrix} F_n \\ F_{n+1} \end{bmatrix} = \underbrace{\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}}_{A} \cdot \begin{bmatrix} F_{n-1} \\ F_n \end{bmatrix} \text{ s.t. } \begin{bmatrix} F_n \\ F_{n+1} \end{bmatrix} = A^n \cdot \begin{bmatrix} F_0 \\ F_1 \end{bmatrix}$$

Matrix

$$\begin{bmatrix} F_n \\ F_{n+1} \end{bmatrix} = \underbrace{\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}}_{A} \cdot \begin{bmatrix} F_{n-1} \\ F_n \end{bmatrix} \text{ s.t. } \begin{bmatrix} F_n \\ F_{n+1} \end{bmatrix} = A^n \cdot \begin{bmatrix} F_0 \\ F_1 \end{bmatrix}$$

Now we can apply the squaring trick to compute A^n!

Summary: Divide and Conquer

1. Recursion tree method: Works for any recurrent relation.
2. Master Theorem: A handknife to solve most recurrent relations.

Summary: Divide and Conquer

1. Recursion tree method: Works for any recurrent relation.
2. Master Theorem: A handknife to solve most recurrent relations.
3. Matrix Multiplication: An intriguing application of divide and conquer, which is still an active research area.
4. Fast Fourier transform — one of the most important algorithm

Summary: Divide and Conquer

1. Recursion tree method: Works for any recurrent relation.
2. Master Theorem: A handknife to solve most recurrent relations.
3. Matrix Multiplication: An intriguing application of divide and conquer, which is still an active research area.
4. Fast Fourier transform — one of the most important algorithm
5. Nearest neighbor search — how to reduce the time of combining
6. More examples: binary search, counting inversions, ...

Questions?