Counting Inversions

Def Given a sequence of integers $(a_1, a_2, \dots, a_n)$, an inversion is a pair (i,j) such that
- $1 \le i < j \le n$
- $a_i > a_j$.

Problem Count # inversions.

App: Measure of similarity bet^n two preference rankings

- Trivial : $O(n^2)$.

Theorem There is $O(n \log n)$ - time algo. to count # inversions.

Idea Given a seq. of size $2n$, divide as

$$a_1 \; a_2 \; \dots \; a_n \;\Big|\; b_1 \; b_2 \; \dots \; b_n$$

→ Count # inversions in $(a_1, \dots, a_n)$

→ " in $(b_1, \dots, b_n)$

→ Count # (i,j) s.t. $a_i > b_j$. → # "cross-inversions"

→ Add up.

- How does one count # cross-inversions in $O(n)$-time?

- Not possible in general.

- But possible if $(a_1, \ldots a_n)$ $(b_1, \ldots, b_n)$ both sorted!

- Do sorting as well!

<u>Algorithm</u>

- <u>Input</u>: Seq. of n integers

- <u>Output</u> - # inversions
  - sorted seq.

<u>Algo</u>:

- Divide seq. of size $2n$ into

$$\underbrace{a_1 \ldots a_n}_{A} \mid \underbrace{b_1 \ldots b_n}_{B}$$

- Run algo. recursively to output

  - $a_1' \leq a_2' \leq \ldots \leq a_n' \mid b_1' \leq b_2' \ldots \leq b_n'$

  - r_A, r_B = # inversions in A, B resp.

- Now count $r = \#(i,j)$ s.t. $a_i' > b_j'$.

$$a_1' \leq a_2' \leq a_3' \quad a_4' \ldots a_5' \quad a_6' \quad a_7' \leq a_8'$$

$$b_1' \leq b_2' \leq b_3' \quad b_4' \quad b_5' \quad b_6' \quad b_7' \leq b_8'$$

- For each a_i' find first/smallest index j s.t. $a_i' \leq b_j'$. $\therefore$ # inversions on a_i' is $j-1$.

  r = Add up over all i.

- Keep one finger on a_i'
  another on b_j'.

  Keep moving fingers to the right.

- Done in one scan! $\Big\}$ $O(n)$ time.

- Output $r_A + r_B + r$.

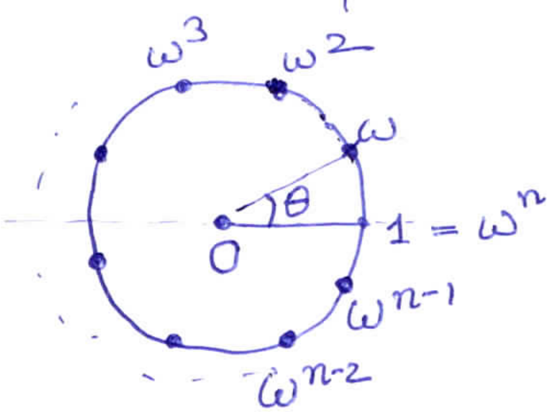
$\blacksquare$

Fast Fourier Transform

Let n be a power of 2, i.e. $n = 2^k$.

Def $\omega = \omega_n = e^{2\pi i/n} = \cos\left(\frac{2\pi}{n}\right) + i\sin\left(\frac{2\pi}{n}\right)$

be complex n^{th} root of unity.



- $\omega^n = 1$
- $e^{i\theta} = \cos\theta + i\sin\theta$
- $i = \sqrt{-1}$.

Def Given a sequence $A = (a_0, a_1, \cdots, a_{n-1})$, its Fourier transform $FT(A) = (b_0, b_1, \cdots, b_{n-1})$ where

$$b_j = \sum_{i=0}^{n-1} a_i \omega^{ji}.$$

Note

$$b_0 = a_0 + a_1 + a_2 + \cdots + a_{n-1}$$

$$b_1 = a_0 + a_1\omega + a_2\omega^2 + \cdots + a_{n-1}\omega^{n-1}$$

$$\vdots$$

$$b_j = a_0 + a_1\omega^j + a_2\omega^{2j} + \cdots + a_{n-1}\omega^{(n-1)j}$$

$$\vdots$$

In matrix form

$$\begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{n-1} \end{bmatrix} = \begin{array}{c c} & i \\ j & \begin{bmatrix} & & \\ & \omega^{ji} & \\ & & \end{bmatrix} \end{array} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}$$

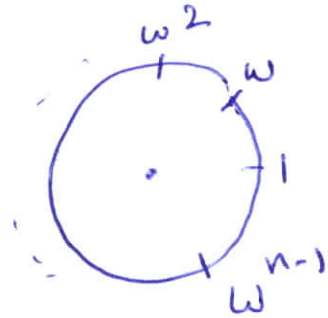
Alternately, if one defines the polynomial

$$P_A(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \cdots + a_{n-1} x^{n-1}$$

then

$$FT(A) = \left(P_A(1), P_A(\omega), P_A(\omega^2), \ldots, P_A(\omega^{n-1}) \right)$$

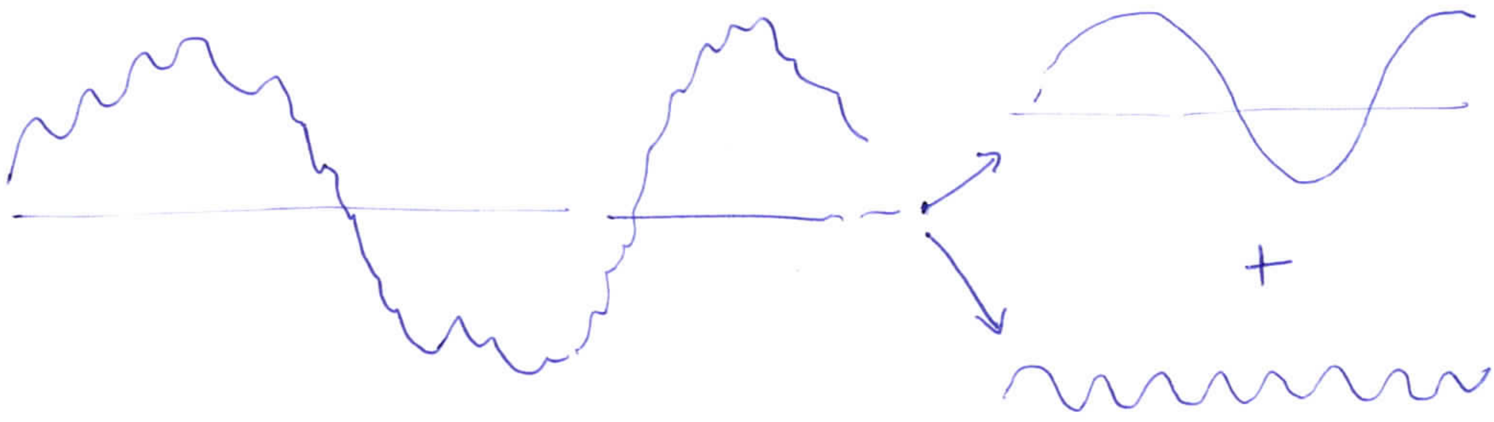
i.e. the polynomial P_A evaluated at n special points $1, \omega, \omega^2, \ldots, \omega^{n-1}$.



- Trivial $O(n^2)$.

<u>Theorem</u> FT can be computed in $O(n \log n)$ time!

<u>Applications</u>
- Many!
- Testing periodicity, signal processing
- Polynomial multiplication



Polynomial Multiplication in $O(n \log n)$ time

Problem Given two polynomials

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

$$Q(x) = \sum_{i=0}^{n-1} b_i x^i$$

Compute $P(x) \cdot Q(x)$. For $0 \le j \le 2n-2$, coefficient of x^j in $P(x)\,Q(x)$ is

$$a_0 b_j + a_1 b_{j-1} + a_2 b_{j-2} + \cdots + a_j b_0 .$$

Trivial $O(n^2)$-time

Theorem Polynomial multiplication can be done in $O(n \log n)$ time w/ FFT as a subroutine.

Fact (Interpolation)

A polynomial of degree $\leq d-1$ is uniquely determined by its values at d distinct points.

Algorithm Given $P(x)$, $Q(x)$ of degree $\leq n-1$

- Think of these as degree $\leq 2n-1$ polys by appending zeroes. Let $\omega = e^{2\pi i/(2n)}$

→ Evaluate $P(x)$ at $x = 1, \omega, \omega^2, \ldots, \omega^{2n-1}$

→ Evaluate $Q(x)$ at $x = 1, \omega, \omega^2, \ldots, \omega^{2n-1}$

Thus we get value of $R(x) = P(x) \cdot Q(x)$ at $x = 1, \omega, \omega^2, \ldots, \omega^{2n-1}$.

→ Obtain $R(x)$ given its values by inverse Fourier Transform!

FFT, FFT, Inverse-FFT.

- Inverse FT is same as FT by replacing ω by $\bar{\omega}$.

$$\begin{array}{c c} & i \\ j & \left[\omega^{ji} \right]^{-1} \end{array} = \frac{1}{n} \left[\bar{\omega}^{ji} \right]$$

$\blacksquare$

— x —

<u>FFT</u> n is power of 2, so assume the given sequence is

$$A = (a_0, a_1, a_2, \cdots, a_{2n-1}).$$

Divide it into

$$B = (a_0, a_2, a_4, a_6, \cdots, a_{2n-2})$$
$$= (b_0, b_1, b_2, \cdots, b_{n-1})$$

$$C = (a_1, a_3, a_5, a_7, \cdots, a_{2n-1})$$
$$= (c_0, c_1, c_2, \cdots, c_{n-1}).$$

$$FT(A)_j$$

$$= \sum_{i=0}^{2n-1} a_i \omega^{ji} \qquad \omega = \omega_{2n} = e^{2\pi i/(2n)}$$

$$= a_0 + a_1 \omega^j + a_2 \omega^{2j} + a_3 \omega^{3j} + \cdots + a_{2n-1} \omega^{(2n-1)j}$$

$$= (a_0 + a_2 \omega^{2j} + a_4 \omega^{4j} + \cdots + a_{2n-2} \omega^{(2n-2)j}) +$$

$$(a_1 \omega^j + a_3 \omega^{3j} + a_5 \omega^{5j} + \cdots + a_{2n-1} \omega^{(2n-1)j})$$

$$= (b_0 + b_1 \omega'^j + b_2 \omega'^{2j} + \cdots + b_{n-1} \omega'^{(n-1)j}) +$$

$$\omega^j (c_0 + c_1 \omega'^j + c_2 \omega'^{2j} + \cdots + c_{n-1} \omega'^{(n-1)j})$$

$$\omega' = \omega^2 = e^{2\pi i/n} = \omega_n$$

$$= \left(\sum_{i=0}^{n-1} b_i \omega_n^{ji}\right) + \omega^j \left(\sum_{i=0}^{n-1} c_i \omega_n^{ji}\right)$$

$$= FT(B)_j + \omega^j \, FT(C)_j.$$

Hence

$$FT(A)_j = FT(B)_j + \omega^j \, FT(C)_j$$

$j \bmod n$ since $0 \leq j \leq 2n-1$

Algorithm

- Compute $FT(B)$, $FT(C)$ recursively
- combine them to obtain $FT(A)$ as above. in $O(n)$-time.
- $\therefore$ overall $O(n \log n)$ time!

Note - FFT implemented in hardware.
- Butterfly network.

Radix Sort

- n integers, b bits each.
- $O(nb)$ - time.

Illustration by example :

0010	1101	0110	1001
0011	0111	0001	

↓ most significant bit

0010	1101
0011	1001
0111	
0110	
0001	

↓ 2^{nd} MSB

0010	0111	1001	1101
0011	0110		
0001			

↓ 3^{rd} MSB

0001 | 0010 0011 || 0111 0110 | 1001 || 1101

$\downarrow$ LSB

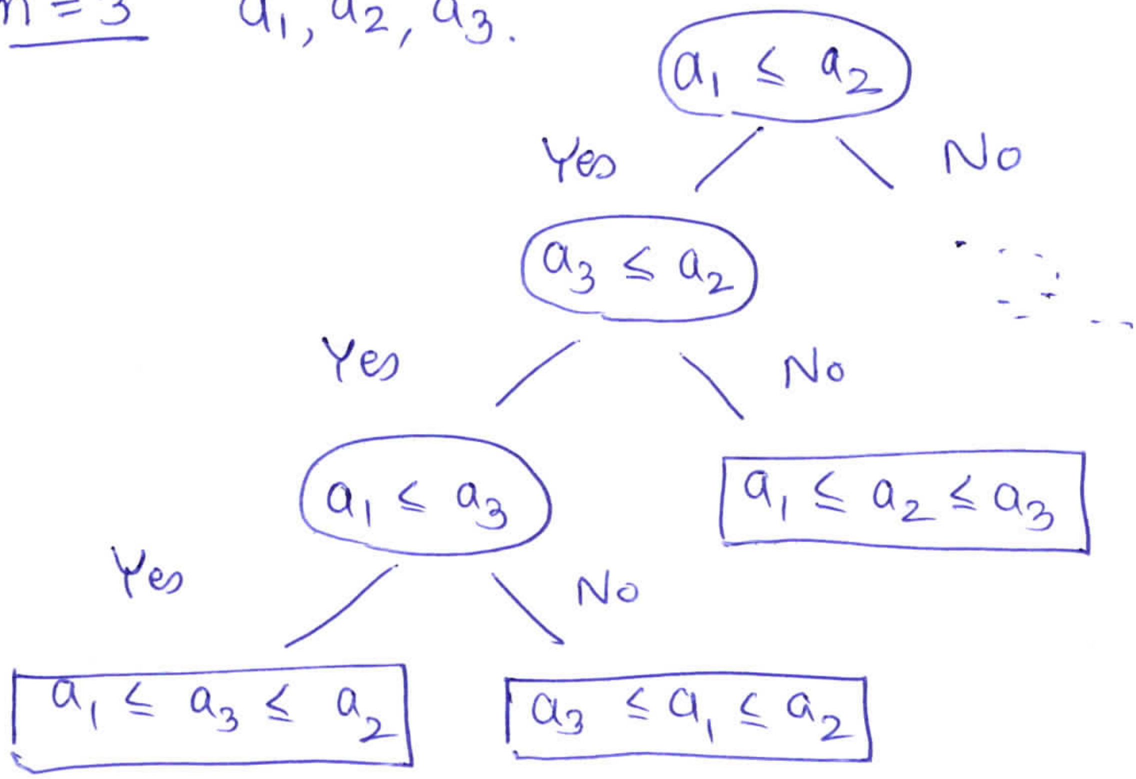
0001 || 0010 | 0011 || 0110 | 0111 | 1001 || 1101

SORTING LOWER BOUND

Theorem Any comparison based sorting algo. must make $\Omega(n \log n)$ comparisons.

- Any such algo. can be represented as decision tree.
  - Every node is a comparison.
  - Branch depending on comparison result.
  - Leaves are sorted orders.
  - Height = (Worst-case) time.

$\underline{n=3}$ $a_1, a_2, a_3.$



- $h =$ height.
- $\#$ nodes $\le 2^h$. $\#$ nodes $= n!$
- $\therefore \quad n! \le 2^h$

$$\therefore \quad h \ge \log n!$$
$$\ge \frac{1}{2} \cdot n \log n.$$

$\underline{\text{Note}}$. $n! \approx 2^{n \log n - \Theta(n)}$