Counting Inversions

Definition: Given a sequence of integers \((a_1, a_2, \ldots, a_n)\), an inversion is a pair \((i, j)\) such that:
- \(1 \leq i < j \leq n\)
- \(a_i > a_j\).

Problem: Count \# inversions.

- 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:
\[a_1, a_2, \ldots, a_n \mid b_1, b_2, \ldots, b_n\]

- Count \# inversions in \((a_1, \ldots, a_n)\)
- \(\mid\) in \((b_1, \ldots, b_n)\)
- Count \# \((i, j)\) s.t. \(a_i > b_j\). \(\rightarrow\) \# "cross-inversions"
- Add up.
- How does one count # cross-inversions in $O(n)$-time?
- Not possible in general.
- But possible if $(a_1, \ldots, a_n)$ both sorted $(b_1, \ldots, b_n)$.
- Do sorting as well!

**Algorithm**

- **Input:** Seq. of $n$ integers
- **Output:** # inversions
  - sorted seq.

**Alg**
- Divide seq. of size $2n$ into

  \[a_1, \ldots, a_n | b_1, \ldots, b_n\]

  \[\text{A} \quad \text{B}\]

- Run algo. recursively to output

  \[a_1' \leq a_2' \leq \ldots \leq a_n' \mid b_1' \leq b_2' \ldots \leq b_n'\]

  \[\gamma_A, \gamma_B = \# \text{ inversions in } \text{A, B resp.}\]
- Now count \( r = \# (i, j) \text{ st. } a_i' > b_j' \).

\[
\begin{align*}
a_1' &\leq a_2' \leq a_3' \quad a_4' &\leq a_5' \quad a_6' \quad a_7' &\leq a_8' \\
b_1' &\leq b_2' \leq b_3' \quad b_4' &\leq b_5' \quad b_6' \quad b_7' &\leq b_8'
\end{align*}
\]

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

\[ r = \text{Add up over all } i. \]

- Keep one finger on \( a_i' \)
  another on \( b_j' \)
  keep moving fingers to the right.

- Done in one scan! \( \Rightarrow O(n) \) time.

- Output \( r_A + r_B + r \).
Fast Fourier Transform

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

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

be a complex $n^{th}$ root of unity.

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

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

Note

$$b_0 = a_0 + a_1 + a_2 + \ldots + a_{n-1}$$
$$b_1 = a_0 + a_1 \omega + a_2 \omega^2 + \ldots + a_{n-1} \omega^{n-1}$$
$$b_j = a_0 + a_1 \omega^j + a_2 \omega^{2j} + \ldots + a_{n-1} \omega^{(n-1)j}$$
In matrix form

\[
\begin{bmatrix}
    b_0 \\
    b_1 \\
    \vdots \\
    b_{n-1}
\end{bmatrix} = j \begin{bmatrix}
    \omega^i \\
    \omega^{2i} \\
    \vdots \\
    \omega^{ni}
\end{bmatrix} \begin{bmatrix}
    a_0 \\
    a_1 \\
    \vdots \\
    a_{n-1}
\end{bmatrix}
\]

Alternately, if one defines the polynomial

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

then

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

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

- Trivial \( O(n^2) \).

**Theorem** FT can be computed in \( O(n \log n) \) time!

**Applications**

- 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 $0 \leq j \leq 2n-2$, coefficient of $x^j$ in $P(x) Q(x)$ is

\[ a_0 b_j + a_1 b_{j-1} + a_2 b_{j-2} + \ldots + a_j b_0. \]

Trivial $O(n^2)$-time

Theorem Polynomial multiplication can be done in $O(n \log n)$ time with 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!

FFTs, FFTs, Inverse-FFT.
Inverse FT is same as FT by replacing $\omega$ by $\overline{\omega}$.

\[
\frac{1}{n} \begin{bmatrix}
\frac{1}{n} & \cdots & \frac{1}{n} \\
\overline{\omega} & \cdots & \overline{\omega}^n
\end{bmatrix}
\]

---

**FFT**

$n$ is power of 2, so assume the given sequence is

\[A = (a_0, a_1, a_2, \ldots, a_{2n-1})\]

Divide it into

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

\[C = (a_1, a_3, a_5, a_7, \ldots, a_{2n-1}) = (c_0, c_1, c_2, \ldots, c_{n-1})\]
\[
\begin{align*}
\text{FT}(A)_j &= \sum_{i=0}^{2n-1} a_i \omega^i \\
&= a_0 + a_1 \omega^j + a_2 \omega^{2j} + a_3 \omega^{3j} + \cdots + a_{2n-1} \omega^{(2n-1)j} \\
\omega &= \omega_{2n} = e^{2\pi i / (2n)} \\
&= \left( a_0 + a_2 \omega^2 + a_4 \omega^4 + \cdots + a_{2n-2} \omega^{(2n-2)j} \right) \\
&\quad + \left( a_1 \omega + a_3 \omega^3 + a_5 \omega^5 + \cdots + a_{2n-1} \omega^{(2n-1)j} \right) \\
&= (b_0 + b_1 \omega^j + b_2 \omega^{2j} + \cdots + b_{n-1} \omega^{(n-1)j}) \\
&\quad + \omega \left( c_0 + c_1 \omega^j + c_2 \omega^{2j} + \cdots + c_{n-1} \omega^{(n-1)j} \right) \\
\omega' &= \omega^2 = e^{2\pi i / n} = \omega_n \\
&= \left( \sum_{i=0}^{n-1} b_i \omega^i \right) + \omega^j \left( \sum_{i=0}^{n-1} c_i \omega^i \right) \\
&= \text{FT}(B)_j + \omega^j \text{FT}(C)_j.
\end{align*}
\]
Hence

\[ \text{FT}(A)_j = \text{FT}(B)_j + \omega^j \text{FT}(C)_j \]

\[ j \mod n \quad \text{since} \quad 0 \leq j \leq 2^n - 1 \]

Algorithm

- Compute \( \text{FT}(B) \), \( \text{FT}(C) \) recursively
- Combine them to obtain \( \text{FT}(A) \) 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:

\[
\begin{array}{cccc}
0010 & 1101 & 0110 & 1001 \\
0011 & 0111 & 0001 & \\
\end{array}
\]

most significant bit

\[
\begin{array}{cccc}
0010 & & 1101 & \\
0011 & & 1001 & \\
0111 & & \\
0110 & & \\
0001 & & \\
\end{array}
\]

2nd MSB

\[
\begin{array}{cccc}
0010 & & 0111 & 1001 | 1101 \\
0011 & & 0110 & \\
0001 & & \\
\end{array}
\]

3rd MSB
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.
n = 3  \ a_1, a_2, a_3.

- \ a_1 \leq a_2

  Yes \quad \text{No}

  - \ a_3 \leq a_2

    Yes \quad \text{No}

    - \ a_1 \leq a_3

      Yes \quad \text{No}

      - \ a_1 \leq a_3 \leq a_2

      - \ a_3 \leq a_1 \leq a_2

- \ h = \text{height.}

- \ # \text{nodes} \leq 2^h.

- \ # \text{nodes} = n!

- \therefore \ n! \leq 2^h

- \therefore \ h \geq \log n!

\geq \frac{1}{2} \cdot n \log n.

\text{Note:} \quad \therefore n! \approx \frac{1}{2} n \log n - \Theta(n)