1 Divide & Conquer Algorithms

1.1 $O(n \log n)$ Algorithm for Counting Inversions I

1.1.1 Divide and Conquer Paradigm

- Divide into smaller subproblems
- Conquer via recursive calls
- Combine solutions of subproblems into one for the original problem

The Problem

Input: Array, A, containing the numbers 1, 2, 3, ..., in the some order

Output: number of inversions = number of pairs (i, j) of array indices with $i < j$ and $A[i] > A[j]$.

If the array is not sorted, the number of inversions will be non-zero.

Example [4:00]: (1, 3, 5, 2, 4, 6)

Invversions: (3, 2), (5, 2), (5, 4)

Pictorially, the number of line crossings corresponds to inversions.

Motivation: Numerical similarity measure between two ranked lists.

Used in collaborative filtering.

In general, the maximum number of inversions in an array is $\binom{n}{2} = \frac{n(n-1)}{2}$.

High-level Approach

Brute-force: Setup a double for loop for i and j and check whether each pair is inverted, if so, add to count. This run in $O(n^2)$ time

Divide and Conquer: Classify inversions (i, j) with $[i < j]$ of an array in one of three types:

- **left**: if $i, j \leq \frac{n}{2}$
- **right**: if $i, j > \frac{n}{2}$
- **split**: if $i \leq \frac{n}{2} < j$

The first two can be computed recursively, split needs separate subroutine.

Hign-Level Algorithm

```
Count(array A, length n)
        if n = 1 return 0
        else
                x = Count(1st half of A, n/2)
                y = Count(2nd half of A, n/2)
                z = CountSplitInv(A, n)
        return x+y+z
```

Goal: implement CountSplitInv in linear $O(n)$ time => then Count will run in $O(n \log n)$ time.

1.2 $O(n \log n)$ Algorithm for Counting Inversions II

1.2.1 Piggybacking on Merge Sort

Key Idea #2: have recursive calls both count inversions and sort.

Motivation: Merge subroutine naturally uncovers split inversions.

```
Sort_and_Count(array A, length n)
        if n = 1 return 0
        else
                (B, x) = Sort_and_Count(1st half of A, n/2)
                (C, y) = Sort_and_Count(2nd half of A, n/2)
                (D, z) = Merge_and_CountSplitInv(A, n)
        return x+y+z
```

output sorted A, B, C.

Merging uncovers number of inversions.

Pseudocode for Merge

D = ouput [length=n]

B = 1^{st} sorted array [n/2]

C = 2^{nd} sorted array [n/2]

i = 1

j = 1

Take B, C, traverse D using the k index, maintain pointers i, j to B and C respectively

for k = 1 to n

if A(i) < B(j) C(k) = A(i) i++ else [B(j)<A(i)] C(k)=B(j) j++ end (ignores end cases)

Suppose the input array A has no split inversions. What is the relationship between the sorted subarrays B and C?

All elements of B are less than all elements of C.

On an array with this property, MergeSort will copy everything from B to D before C even gets touched. So copying elements from array C has something to do with split inversions.

Ex.: [7:50]

General Claim: The split inversions involving an element, y, of the 2nd array C are percisely the numbers left in the 1st array B when y is copied to the output D.

Proof: let x be an element of the 1st array B

- if x copied to output D before y, then $x < y$ ⇒no inversion involving x&y
- if y copied to output D before x, then $y < x$ ⇒x&y are a split inversion

1.2.2 Merge_and_CountSplitInv

While merging the two sorted subarrays, keep running total of numbers of split inversions.

When element of 2nd array C gets copied to output D, increment total by number of elements remaining in 1st array.

1.2.3 Run time of subroutine

Linear time in merging, constant for count, sloppy:

$$O(n) + O(n) = O(n)$$

1.3 Strassen's Subcubic Matrix Multiplication Algorithm

$$X \times Y = Z$$

$n \times n$ matrices, where

$$Z_{ij} i^{th} \text{ row of Y} \times i^{th} \text{ column of Y} = \sum_{k=1}^{n} X_{ik} \times Y_{kj}$$

Note: input size $= O(n^n)$

Ex.: $(n = 2)$

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} e & f \\ g & h \end{bmatrix} = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}$$

Running time from definition: $O(n^3)$ relative to matrix dimension, n.

1.3.1 Applying Divide and Conquer

Idea:

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

$$Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

where A through H are all $\frac{n}{2} \times \frac{n}{2}$ matrices.

then:

$$X \times Y = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}$$

1.3.2 Recursive Algorithm #1

Step 1: recursively compute the 8 necessary products

Step 2: do the necessary additions, $O(n^2)$ time

Fact: running time is $O(n^3)$ [fallows from master method]

1.3.3 Strassen's Algorithm (1969)

Step 1: recursively compute only 7 (cleverly chosen) products

Step 2: do the necessary (clever) additions + substractions (still $O(n^2)$ time)

Fact: better than cubic time! [see Master Method lecture for running time]

1.3.4 The Details:

The Seven Products:

$$P_1 = A(F - H)$$
$$P_2 = (A + B)H$$

$$P_3 = (C + D)E$$

$$P_4 = D(G - E)$$

$$P_5 = (A + D)(E + H)$$

$$P_6 = (B - D)(G + H)$$

$$P_7 = (A - C)(E + F)$$

Claim:

$$\begin{aligned} X \times Y &= \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix} \\ &= \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix} \end{aligned}$$

Proof of one term:

$$\begin{aligned} P_5 + P_4 - P_2 + P_6 &= \\ AE + \cancel{AH} + \cancel{DE} + \cancel{DH} + \cancel{DG} - \cancel{DE} &- \\ \cancel{AH} - \cancel{BH} + BG + \cancel{BH} - \cancel{DG} - \cancel{DH} &= AE + BG \end{aligned}$$