Divide and Conquer

- Approach:
  - Break problem into smaller sub-problems
  - Solve smaller sub-problems via recursion
  - Combine solutions of sub-problems to get a solution to the original problem
- A special case of recursion
- Reduce a given problem to multiple smaller instances of the original problem
  - constant factor smaller (n -> n/b)

Problem: Inversion counting

- Input: an array A containing numbers 1,2,…,n in arbitrary order
- Output: number of inversions, i.e., the number of pairs (i, j) such that $i < j$ and $A[i] > A[j]$

Example

- Input $(1, 4, 2, 5, 3)$

Why study this problem?

- Consider a set of n movies

You and I can both rank them according to how much we like them

Mine: 1, 2, 3, 4, …n

Yours: 5, 4, 1, …

Measure the difference between two ranked lists --- a fundamental operation behind all the recommender systems (collaborative filtering)

High level idea

- Brute force counting

- Divide and conquer
  - Break the array into two parts
  - Count the left part
  - Count the right part
  - Count in the inversions cross the two parts

High level algorithm

Count(A, n)

if n=1 return 0

else

x=Count$\left(A_l, \frac{n}{2}\right)$

y=Count$\left(A_r, \frac{n}{2}\right)$

z=CountCross(A_l, A_r)

return x+y+z

countcross(A_l, A_r) needs to run in $O(n)$ in order to achieve an overall runtime of $O(n \log n)$

What if A_l and A_r are already sorted?

Can we count the cross inversions in $O(n)$ time?

$A_l = (2,4,5) \qquad A_r = (1,3,6)$

Building on Merge_sort

Sort-and-Count(A, n)

if n=1 return 0

else

(L_s, x)=Sort-and-Count$\left(A_l, \frac{n}{2}\right)$

(R_s, y)=Sort-and-Count$\left(A_r, \frac{n}{2}\right)$

(A_s, z)=Merge-and-CountCross(L_s, R_s)

return $(A_s, x + y + z)$

Pseudo code for merge sort

L: the left-half sorted array

R: the right-half sorted array

A: output sorted array of size n

$i = 1;\ j = 1$

for $k = 1$ to n

    if $L(i) \leq R(j)$

        $A(k) = L(i);\ \ i++$

    else

        $A(k) = R(j);\ j++$

    end if

End for

(Ignore the end case)

Pseudo code for merge sort

L: the left-half sorted array

R: the right-half sorted array

A: output sorted array of size n

$i = 1;\ j = 1$

for $k = 1$ to n

    if $L(i) \leq R(j)$

        $A(k) = L(i);\ \ i++$

    else

        $A(k) = R(j);\ j++$

    end if

End for

(Ignore the end case)

What would happen if there is no inversion between L and R?

How many inversions can we be sure of when the else branch is taken?

Example

- Consider merging $(1, 3, 5)$ and $(2, 4, 6)$

- Claim: when an element y of R is copied into the output array A, the number of inversion y incurs = the number of elements left in L
- Proof:

Let x be an element of L.

1. If x is copied into A before y, we know $x < y$, which does not cause inversion
2. If x is copied into A after y, we know $y < x$, which causes inversion

Pseudo code for Merge-and-CountCross

L: the left-half sorted array

R: the right-half sorted array

A: output sorted array of size n

z: the # of cross inversions

$i = 1;\ j = 1; z = 0$

for $k = 1$ to n

    if $L(i) \leq R(j)$

        $A(k) = L(i);\ \ i++$

    else

        $A(k) = R(j);\ j++$

        $z = z + \left(\frac{n}{2} + 1 - i\right)$

    end if

End for

(Ignore the end case)

Run time of subroutine:

$O(n)$

Run time of the overall algorithm:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

Same as merge_sort:

$O(n \log n)$