Divide and Conquer: Counting Inversions
Collaborative filtering
- matches your preference (books, music, movies, restaurants) with that of others
- finds people with similar tastes
- recommends new things to you based on purchases of these people

The basis of collaborative filtering: compare the similarity of two rankings
What's similar?

Given numbers 1 to n (the things) rank these according to your preference
- You get some permutation of 1..n
- Compare to someone else's permutation

Extreme similarity
- somebody else's ranking is exactly the same

Extreme dissimilarity
- somebody else's ranking is exactly the opposite

In the middle:
- count the number of out of place rankings
Simplify it

Count the number of **inversions** of a ranking
- \( r_1, r_2, \ldots, r_n \)
- count the number of out of order pairs
  - \( i < j \quad r_i > r_j \)
- eg: 2 1 4 3 5 2 inversions: (2,1) (4,3)

Why is this synonymous with comparing two different rankings?

Because we can re-number the things, such that one of the rankings (e.g. my ranking) becomes 1,2,...,n

my ranking: 1,2,...,5 your ranking 2,1,4,3,5
your #1 is my #2, your #2 is my #1
your #3 is my #4, your #4 is my #3
Visualizing inversions

zero inversions

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

one inversion

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 \\
1 & 2 & 3 & 4 & 5 \\
\end{array}
\]

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 \\
\end{array}
\]

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 \\
\end{array}
\]

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 \\
\end{array}
\]

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 \\
\end{array}
\]
Visualizing inversions

how many?  3  2  1  4  5

enumerate them

3: (3,1) (3,2) (1,2)

7: (5,2) (5,3), (5,1) (5,4)
(1,4) (1,3) (1,2)

all lines crossing!

Careful: don't count inversions twice!
Does Bubble sort count inversions?
Bubble sort is $O(n^2)$

Do it on: 4 2 3 5 1 and see what happens
Do bubble sort, show each swap, count inversions

4 ←→ 2  3  5  1
1  2  3  4  5

2  3  4  5 ←→ 1
1  2  3  4  5

2  3 ←→ 1  4  5
1  2  3  4  5

2 ←→ 1  3  4  5
1  2  3  4  5

every swap takes out 1 inversion, and thus 1 line crossing
Can we do better?

Notice: there are potentially $n^2/2$ inversions. Why?
Reverse order, all pairs are out of orders

Bubble sort counts each individual swap = inversion. To do better we must not count each individual inversion.

Think of merge sort

- in merge sort we do not swap consecutive elements that are out of order as in bubble sort, we make larger distance swaps
- if we can merge sort and keep track of the number of inversions we may get an $O(n \log n)$ algorithm
- Key observation: when an element from right is merged in, it “jumps” over all remaining elements of left!!
Eg: [ 4 2 3 5 1 ]

**sort** [4 2 3 5 1]

- **sort LEFT:** [4 2 3]
  - sort left: [4 2] \(\rightarrow\) [2 4]: 1 inversion
  - sort right: [3]
  - merge(left,right) \(\rightarrow\) [2 3 4] 1 inversion (3 jumps over 4)

- **sort RIGHT:** [5 1] \(\rightarrow\) [1 5] 1 inversion

- **merge(LEFT,RIGHT)** \(\rightarrow\) [1 2 3 4 5] 3 inversions (1 jumps over 2,3 & 4)

**Total inversions:** 1+1+1+3=6 (go check the visualization)
The algorithm

While merging in merge sort keep track of the number of inversions.
When merging an element from left: no inversions added
When merging an element from right: how many inversions added?

As many elements as are remaining in left, because the element from the right jumps over all the remaining elements from left
Counting Inversions: Algorithm

count_inversions(list)
    if list has one element
        return 0
    divide list into two halves A and B
    r_A = count_inversions(A)
    r_B = count_inversions(B)
    r_m = merge-and-count(A, B, list)
    return r_A + r_B + r_m

merge-and-count(L, R, list)
    count = 0
    while L and R not empty:
        put smallest of Li and Rj in list
        if Rj smallest
            add number of elements remaining in L to count
        if L or R empty:
            append the other one to list
    return count
Running time

Just like merge sort, the sort and count algorithm running time satisfies:

\[ T(n) = 2 \ T(n / 2) + cn \]

Running time is therefore \( O(n \log n) \)