Key recurrences for divide-and-conquer algorithms:

\[ T(n) = T\left(\frac{n}{2}\right) + O(1) = O(\log n) \] (binary search)

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

\[ T(n) = 2T\left(\frac{n}{2}\right) + O(1) = O(n) \]

\[ T(n) = 2T\left(\frac{n}{2}\right) + O(n) = O(n \log n) \] (Merge Sort)

From the desired recurrence we get a high level idea of the algorithm.

Divide & Conquer approach:

1) Break into subproblems of the same type.
   Typically problems of half the size.

2) Recursively solve these subproblems

3) Combine/merge solutions to subproblems to get solution to whole problem.
Counting inversions:

Given 2 ordered lists of \( n \) items, find # of inversions = pairs in different order in the 2 lists.

Example: Songs A, B, C, D, E

Person 1: A > B > C > D > E

Person 2: A > C > D > B > E

2 inversions: \((B, D), (B, C)\)

Applications:

Collaborative filtering - Given a person's rankings (e.g., for books, movies, or restaurants), find a person with similar rankings so can offer recommendations.

MetaSearch - Look at a query on several search engines to try to find similarities & generate a merged mega-list.
Can assume the items are labelled 1, 2, ..., n
& the 1st list is 1 > 2 > 3 > ... > n.
(can always renumber to achieve this).

Then the other list is given as

\[ a_1, \ldots, a_n \]

where \( a_i \) = position of item \( i \) in 2nd list.

Earlier example:

\[
\begin{array}{cccccc}
\text{items} & A & B & C & D & E \\
\text{Person 1} & 1 & 2 & 3 & 4 & 5 \\
\text{Person 2} & 1 & 4 & 3 & 2 & 5 \\
\end{array}
\]

\# of inversions

= \# of pairs \( i, j \) where \( i < j \) & \( a_i > a_j \)

= 2 since \( (i=2, j=4) \) has \( a_i = 4 > a_3 = 3 \)
& \( (i=2, j=3) \) has \( a_i = 4 > a_j = 2 \)
Naive approach:

Look at all \( ij \) pairs where \( i < j \)
\[
\Rightarrow O(n^2) \text{ time.}
\]

Try for \( O(n \log n) \) algorithm:

So aim for recurrence: 
\[
T(n) = 2T(\frac{n}{2}) + O(n) \\
= O(n \log n)
\]

Idea: Break input \( A = [a_1, \ldots, a_n] \) into \( A_L = [a_1, \ldots, a_{\frac{n}{2}}] \) 
\& \( A_R = [a_{\frac{n}{2}+1}, \ldots, a_n] \)

1) Find \# of inversions in \( A_L \)
2) Find \# of inversions in \( A_R \)
3) In \( O(n) \) time, find \# of inversions of pairs \( ij \) with \( i \in A_L \) \& \( j \in A_R \)
4) Return sum of these 3 quantities.

How to do step (3) in \( O(n) \) time?
Example:

\[ A = [1, 5, 4, 8, 10, 2, 6, 9, 12, 11, 3, 7] \]
\[ A_L = [1, 5, 4, 8, 10, 2] \quad A_R = [6, 9, 12, 11, 3, 7] \]

5 inversions in \( A_L \):

\[(5, 4), (5, 2), (4, 2), (8, 2), (10, 2)\]

8 inversions in \( A_R \):

\[(6, 3), (9, 3), (9, 7), (12, 11), (12, 3), (12, 7), (11, 3), (11, 7)\]

9 inversions in \( A_L \leftrightarrow A_R \):

\[(5, 3), (4, 3), (8, 6), (8, 7), (8, 3), (10, 6), (10, 9), (10, 3), (10, 7)\]

How to compute this in \( O(n) \) time?
Take element of $A_z$:

Say for 5, need to compute # of elements of $A_k$ that are < 5.

In general for $a_i \in A_z$, compute # of elements of $A_k$ that are < $a_i$.

If $A_k$ was sorted then this is easy to compute.

And if $A_z$ was sorted, using a pair of Pointers, we could compute it for all elements of $A_z$ in $O(n)$ total time.

Key: Sorting $A_z$ & Sorting $A_k$ does not effect # of inversions $A_z \leftrightarrow A_k$.

But we don't have time to sort.

Since our algorithm "looks" like MergeSort when counting inversions in $A_z$ (or $A_k$) then sort it too!
Change problem to count-and-sort

Input: \( A = [a_1, \ldots, a_7] \)

Output: # of inversions & sorted \( A \).

High level algorithm:

1) Break \( A \) into \( A_L = \{1, \ldots, \frac{n}{2}\} \) items
   \( A_R = \{\frac{n}{2} + 1, \ldots, n\} \) items

2) Recursively find # of inversions within \( A_L \)
   & sort \( A_L \)

3) Recursively find # of inversions within \( A_R \)
   & sort \( A_R \)

4) Scan sorted \( A_L \) & \( A_R \) to:
   a) find # of inversions between \( A_L \) & \( A_R \)
   b) sort \( A_L \) & \( A_R \)
Idea of step (4):

\[ A_L = \{2, 4, 5, 8, 10\} \quad A_R = \{3, 6, 7, 9, 11, 12\} \]

Pointers to current smallest in each take smaller of 2:

a) if from \( A_R \) then increase count by \# \text{remaining in} \( A_L \)

b) paste it onto merged sorted list & move pointer right by one.

**Count-and-Sort(A)**

input: \( A = [a_1, \ldots, a_n] \) where \( n \) is a power of 2

output: \# of inversions & sorted \( A \)

Let \( A_L = [a_1, \ldots, a_{n/2}] \) & \( A_R = [a_{n/2+1}, \ldots, a_n] \)

\[(\text{count}_1, B) = \text{count-and-sort}(A_L)\]

\[(\text{count}_2, C) = \text{count-and-sort}(A_R)\]

\[(\text{count}_3, D) = \text{count-and-merge}(B, C)\]

return \((\text{count}_1 + \text{count}_2 + \text{count}_3, D)\)
Count-and-Merge$(B,C)$:

input: sorted $B = [b_1, \ldots, b_k]$ & $C = [c_1, \ldots, c_l]$
output: # of inversions between $B$ & $C$
and sorted $B \cup C$

if $k = 0$, return $(0,C)$
if $l = 0$, return $(0,B)$
if $b_i < c_j$, then
\[ \text{count}_D = \text{count-and-merge}([b_1, \ldots, b_k], C) \]
\[ \text{return} (\text{count}, [b_1, D]) \]
else
\[ \text{count}_D = \text{count-and-merge}(B, [c_1, \ldots, c_l]) \]
\[ \text{return} (k + \text{count}, [c_1, D]) \]

Running time is just like MergeSort:
\[ T(n) = 2T\left(\frac{n}{2}\right) + O(n) = O(n \log n). \]