• **DISCLAIMER:** These notes are not necessarily an accurate representation of what I said during the class. They are mostly what I intend to say, and have not been carefully edited.

• Main topics:
  - Runtime recurrences and how to obtain them from divide-and-conquer algorithms.
  - Guess-and-check process for solving runtime recurrences:
    * Guess an answer,
    * Verify that it fits the recurrence (with explicit constants).
  - Example: Binary search, recurse onto one half each time.
  - Example: Inversion pair counting:
    * Cut into two halves.
    * Sort the second half.
    * For everything in the first half
      * Compute its rank on the second half.
      * This equals to the number of inversions pairs involving it and something in the second half.

We continue with the inversion counting problem introduced last time. The goal is to give a fast algorithm for counting the number of inversion pairs in an array $Arr[1 \ldots n]$. An inversion pair is a pair of numbers that’s in reverse order, aka. $i < j$ s.t. $Arr[i] > Arr[j]$.

The number of inversion pairs is $\Theta(n^2)$, so to do better than the try-all-pairs algorithm, we need to be able to tally multiple inversions in a single step.

This can be done in several ways, all of which involves ‘ranking’ $Arr[i]$ in another list. The **rank** of an element $x$ in a set $S$ is the number of things in $S$ that’s less than it. (for simplicity, we’ll work under the assumption that all elements are distinct). Note that for an element $Arr[i]$, the number of inversion pairs involving it is precisely its rank in the set $Arr[i + 1] \ldots Arr[n]$.

For example, if we have

$$Arr = 8, 7, 12, 3, 5, 11, 2, 9,$$
the number of inversion pairs that \( Arr[2] = 7 \) is involved in is its rank in the elements after it,

\[
12, 3, 5, 11, 2, 9,
\]

and the number of inversion pairs that \( Arr[4] = 3 \) is involved in is its rank in the elements

\[
5, 11, 2, 9.
\]

So if we’re able to compute this rank quickly, we can tally up to \( \Theta(n) \) inversion pairs in a single step.

Given a set \( S \), we can pre-process it to handle rank operations very fast. Suppose \( S \) is sorted,

\[
s_1 \leq s_2 \leq s_3 \leq \ldots \leq s_n.
\]

Then to rank an element \( x \), we first compare it with \( s_{n/2} \):

- If \( x \leq s_{n/2} \), then it suffices to find the rank of \( x \) on \( s_1 \ldots s_{n/2} \).
- Otherwise, we know that there are \( n/2 \) elements less than \( x \). Adding this to the rank of \( x \) in \( s_{n/2+1} \ldots s_n \) gives the answer.

In either case, in a single step, we went from a list of size \( n \) to one of size \( n/2 \). The performance of such a routine can be formalized through a runtime recurrence. Let \( T(n) \) denote the time it takes to rank an element on a list of \( n \) elements. Then we have

\[
T(n) = T(n/2) + O(1).
\]

This recurrence solves to \( O(\log n) \) because each step \( n \) halves. To prove it formally, we can use guess-and-check.

1. Guess: \( T(n) \leq C \log_2 n \).

2. Assume \( T(n/2) \leq C \log_2 (n/2) = C(\log_2 n - 1) \).

3. Plug into recurrence and check:

\[
C(\log_2 n - 1) + O(1) = C \log_2 n - C + O(1) \leq C \log_2 n.
\]

The last inequality is because we can choose \( C \) to be larger than the constant in the \( O(1) \), getting \(-C + O(1) \leq 0\).

This proof is really strong induction: it’s using the fact that all the instances recursed on are smaller than the original.

Binary search allows us to preprocess a set by sorting it (in \( O(n \log n) \) time) so that rank operations on it can be done in \( O(\log n) \) time. Note that this time is much less than the potential \( \Theta(n) \) inversion pairs involving \( Arr[i] \). However, the main difficulty facing this approach is that the set of numbers that we need to perform this ‘rank’ operation is
continuously changing. Sorting it each time would incur a total cost of $\Omega(n^2 \log n)$, more than the brute force algorithm.

This is where divide-and-conquer comes in. We pick a middle $m$ that splits the array into two halves, and recursively count inversions on both. The only inversion pairs not counted are the ones that ‘cross’ the middle:

$$i \leq m < j.$$

Then for every element in $Arr[1 \ldots m]$, we need to find its rank in $Arr[m + 1 \ldots n]$. This leads to $O(n)$ rank operations on a static list of size $O(n)$. Going back to the example earlier, we split the array into

$$Arr[1 \ldots 4] = 8, 7, 12, 3; \quad Arr[5 \ldots 8] = 5, 11, 2, 9,$$

and the number of inversion pairs of $Arr[2] = 7$ and $Arr[4] = 3$ in the second half are both their ranks in the set

$$5, 11, 2, 9.$$

Note that for $Arr[2] = 7$, we ignored the inversion pair with $Arr[4] = 3$. This is fine: this pair is contained in the left half, and is counted by the recursive call on it.

Formally, to compute the number of inversions, we sort $Arr[m + 1 \ldots n]$ in $O(n \log n)$ time. Then for each entry in $Arr[1 \ldots m]$, we perform a single binary search on the sorted list, which takes $O(\log n)$ time. This leads to the runtime recurrence:

$$T(n) = 2T(n/2) + O(n \log n),$$

which we will solve next time.