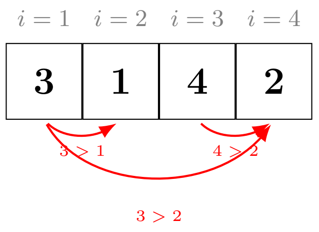
Note: Your TA probably will not cover all the problems. This is totally fine, the discussion worksheets are not designed to be finished in an hour. They are deliberately made long so they can serve as a resource you can use to practice, reinforce, and build upon concepts discussed in lecture, readings, and the homework.

1 Counting Inversions

Let $A = (a_1, a_2, \ldots, a_n)$ be a sequence of n distinct numbers. We say that a pair of indices (i, j) forms an **inversion** if $i < j$ but $a_i > a_j$.

Visualizer: `https://gemini.google.com/share/6805cbd9f9be`

Example: Consider the array $A = [3, 1, 4, 2]$. The inversions are the pairs of values $(3, 1)$, $(3, 2)$, and $(4, 2)$.



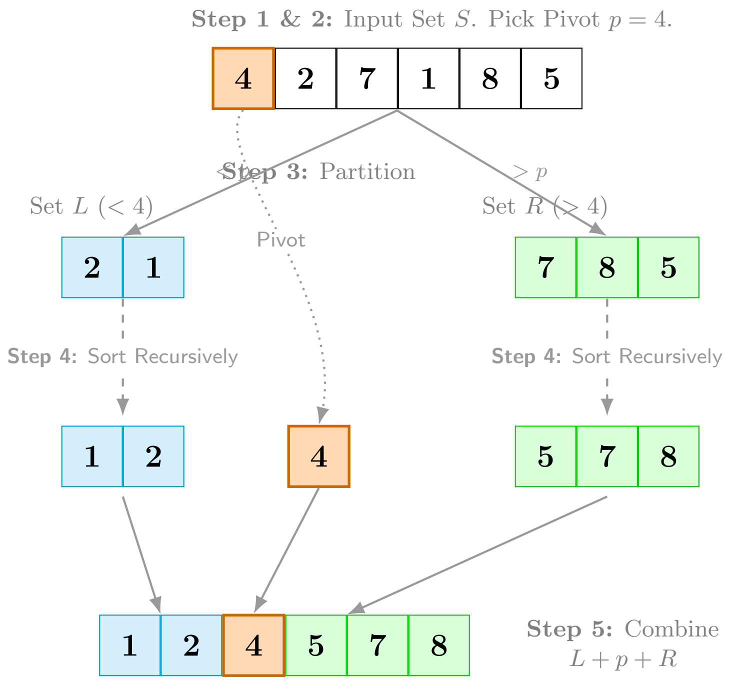
(a) *(Optional Warmup)* Provide an algorithm to determine the number of inversions in the sequence $(a_1, \ldots, a_n)$ in time $\mathcal{O}(n^2)$. Be brief, no need to analyze runtime or proof of correctness.

(b) Provide a divide and conquer algorithm to determine the number of inversions in the sequence $(a_1, \ldots, a_n)$ in time $\mathcal{O}(n \log n)$.

2 Quicksort

Let S be a set of n distinct numbers. We want to sort these numbers in ascending order. Consider the following recursive strategy: [1]

1. Pick a single element p from the set S (we call this the "pivot").
2. Compare every other element in S to p.
3. Place all elements smaller than p into a set L, and all elements larger than p into a set R.
4. Recursively sort L and R.
5. Combine the sorted version of L, the element p, and the sorted version of R to form the result.



Visualizer: `https://gemini.google.com/share/a8709e8eb345`

(a) Using the strategy described above as a subroutine, write the pseudocode for a recursive function `Sort(S)` that returns the sorted list. Clearly state your base case.

[1] Yes, we are learning *yet* another sorting algorithm, but for practical purposes, unless you put in a lot of work, merge sort requires $\mathcal{O}(n)$ auxiliary space. If you are sorting an array that is in the billions of entries (like a database for your next big social media app), quicksort might be more practical.

(b) **Worst-case Analysis:** Suppose that for every recursive call, the pivot p you choose happens to be the **smallest** element in the current subset.

Derive a recurrence relation $T(n)$ for the number of comparisons performed in this specific case, and solve it to find the Big-O time complexity.

Hint: If p is the minimum, what are the sizes of the sets L and R passed to the recursive calls?

(c) *(Optional Warmup)* `Bogosort` is a fun sorting algorithm where you take an array and shuffle it, check if it is sorted, and if not, repeat shuffling and checking until it is sorted. Calculate the best-case and expected runtime for `Bogosort`.

(d) **Probabilistic Analysis:** To avoid the worst-case scenario, we modify the algorithm to pick the pivot p **uniformly at random** from the current set S. Show that the expected runtime is $O(n \log n)$.

3 Agent Meetup

Manhattan has an "amazing" road system where streets form a checkerboard pattern, and all roads are either straight North-South or East-West. We simplify Manhattan's roadmap by assuming that each pair x, y, where x and y are integers, corresponds to an intersection. As a result, the distance between any two intersections can be measured as the *Manhattan distance* between them, i.e. $|x_i - x_j| + |y_i - y_j|$.

You, working as a mission coordinator at the CS 170 Secret Service Agency, have to arrange a meeting between two of n secret agents located at intersections across Manhattan. Hence, your goal is to find the two agents that are the closest to each other, as measured by their Manhattan distance.

Design an efficient algorithm to find the minimum Manhattan distance between any two agents in $O(n \log^2 n)$ time. As mentioned before, you can assume that the coordinates of the agents are integer values, i.e. the ith agent is at location (x_i, y_i) where x_i, y_i are integers.

Note: The optimal runtime is $\Theta(n \log n)$, but it is not required for full credit. This problem is very geometric; we suggest you draw examples when working on it!