Insertion Sort

- Time complexity: O(n^2), Space complexity: O(1)
- Visited every nodes, insert it into the right position of the already sorted array on the left side.

Selection Sort

- Time complexity: O(n^2),   Space complexity: O(1)
- Traversal i for 0 to the last index of the array, find the i-th smallest number in the array and swap if with the i-th element

Heap Sort

- Time complexity: O(logn), Space complexity: O(long)
- Visited every nodes, move the bigger one on the left to the right, insert into the right place

Merge Sort

- Time complexity: O(nlogn), Space complexity: O(n) Because we need a clone of the two subarrays.
- Divid the array into two parts from the middle, and recursively sort both subarrays. And than merge the two sorted array into one array with two pointers.

Quick Sort

- Time complexity: O(nlogn), Space complexity: O(nlogn) (Partition Version) O(n) (Faster Version)
- Randomly choose a pivot, and split the array in to three subarrays, bigger ones, smaller ones, and the others. Recursively sort the subarrays of bigger ones and smaller ones separately.

Discovery

I applyed a traditional quick sort method at first. I randomly chose a pivot, splited the array into two parts, all of the element in one part were smaller than the pivot and the element in the other subarray were equal to or larger than the pivot, which took a really long time.

After several experiments, I found out that this is because **the characteristics of the Input dataset**, which is almost all of the element are repeated. In fact, there are 120k words in the novel but only 7k of them are unique.

According to this feature, I optimized my function. I divided the array into three parts, smaller ones, equal ones, and larger one. In this case, I don't have to choose the repeated word as pivot any more, which makes a huge reduction of sorting time.