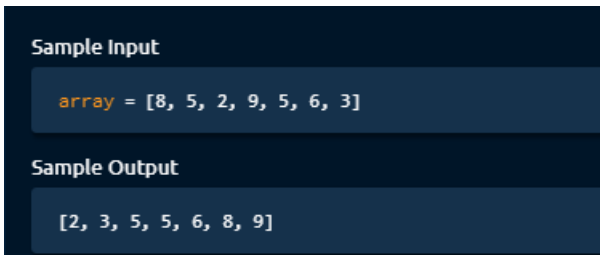
Topic - Sorting

Difficulty Easy: Bubble Sort

Write a function that takes in an array of integers and returns a sorted version of that array. Use the Bubble Sort algorithm to sort the array.



Hint:

Traverse the input array, swapping any two numbers that are out of order and keeping track of any swaps that you make. Once you arrive at the end of the array, check if you have made any swaps; if not, the array is sorted and you are done; otherwise, repeat the steps laid out in this hint until the array is sorted.

Optimal Space & Time Complexity

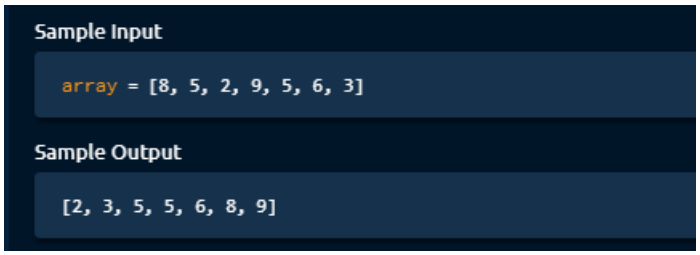
Best: O(n) time | O(1) space - where n is the length of the input array Average: O(n^2) time | O(1) space - where n is the length of the input array Worst: O(n^2) time | O(1) space - where n is the length of the input array

Difficulty Easy: Insertion Sort

Write a function that takes in an array of integers and returns a sorted version of that array. Use the Insertion Sort algorithm to sort the array.

Hint:

Divide the input array into two subarrays in place. The first subarray should be sorted at all times and should start with a length of 1, while the second subarray should be unsorted. Iterate through the unsorted subarray, inserting all of its elements into the sorted subarray in the correct position by swapping them into place. Eventually, the entire array will be sorted.

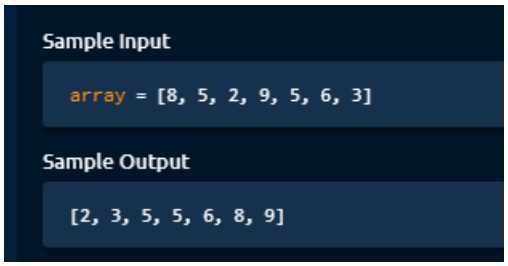


Optimal Space & Time Complexity

Best: O(n) time | O(1) space - where n is the length of the input array Average: O(n^2) time | O(1) space - where n is the length of the input array Worst: O(n^2) time | O(1) space - where n is the length of the input array

Difficulty Easy: Selection Sort

Write a function that takes in an array of integers and returns a sorted version of that array. Use the Selection Sort algorithm to sort the array.



Hint:

Divide the input array into two subarrays in place. The first subarray should be sorted at all times and should start with a length of 0, while the second subarray should be unsorted. Find the smallest (or largest) element in the unsorted subarray and insert it into the sorted subarray with a swap. Repeat this process of finding the smallest (or largest) element in the unsorted subarray and inserting it in its correct position in the sorted subarray with a swap until the entire array is sorted.

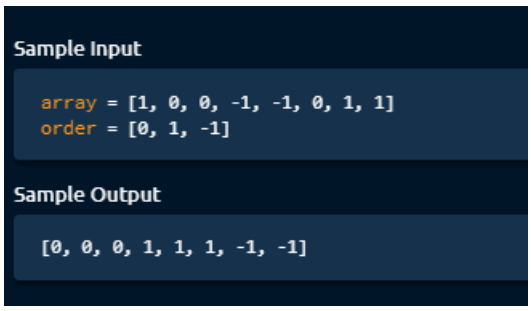
Optimal Space & Time Complexity

Best: O(n^2) time | O(1) space - where n is the length of the input array
Average: O(n^2) time | O(1) space - where n is the length of the input array
Worst: O(n^2) time | O(1) space - where n is the length of the input array

Difficulty Medium : Three Number Sort

You're given an array of integers and another array of three distinct integers. The first array is guaranteed to only contain integers that are in the second array, and the second array represents a desired order for the integers in the first array. For example, a second array of [x, y, z] represents a desired order of [x, x, ..., x, y, y, ..., y, z, z, ..., z] in the first array. Write a function that sorts the first array according to the desired order in the second array. The function should perform this in place (i.e., it should mutate the input array), and it shouldn't use any auxiliary space (i.e., it should run with constant space: O(1) space).

Note that the desired order won't necessarily be ascending or descending and that the first array won't necessarily contain all three integers found in the second array—it might only contain one or two.



Hint:

What advantage does knowing the three values contained in the array give you, and how can you use that to solve this problem in linear time? Try counting how many times each of the three values appears in the input array. Once you have these counts, you can repopulate the input array as need be. Putting aside the first two hints, try conceptually splitting the original array into three subarrays and moving elements of each unique value into the correct subarray. You'll need to keep track of the respective starting indices of these subarrays. You can solve this problem either with two passes through the input array or with a single pass. If you do two passes through the array, you'll specifically be positioning the first ordered element during the first pass and the third ordered element during the second pass. You'll be swapping elements from the left side of the array whenever you encounter the first element, and you'll be swapping elements from the right side of the array whenever you encounter the third element. You'll have to keep track of where you last placed a first element or a third element. With a single pass through the array, you'll have to implement both of these strategies and a little more all at once. . .

Optimal Space & Time Complexity

O(n) time | O(1) space - where n is the length of the array

Difficulty Hard: Quick Sort

Write a function that takes in an array of integers and returns a sorted version of that array. Use the Quick Sort algorithm to sort the array.

Hint:

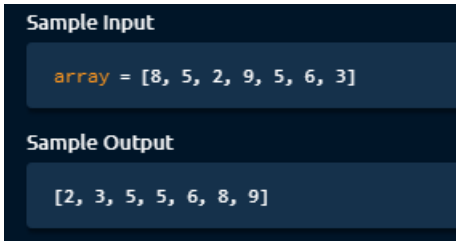
Quick Sort works by picking a "pivot" number from an array, positioning every other number in the array in sorted order with respect to the pivot (all smaller numbers to the pivot's left; all bigger numbers to the pivot's right), and then repeating the same two steps on both sides of the pivot until the entire array is sorted. Pick a random number from the input array (the first number, for instance) and let that number be the pivot. Iterate through the rest of the array using two pointers, one starting at the left extremity of the array and progressively moving to the right, and the other one starting at the right extremity of the array and progressively moving to the left. As you iterate through the array, compare the left and right pointer numbers to the pivot. If the left number is greater than the pivot and the right number is less than the pivot, swap them; this will effectively sort these numbers with respect to the pivot at the end of the iteration. If the left number is ever less than or equal to the pivot, increment the left pointer; similarly, if the right number is ever greater than or equal to the pivot, decrement the right pointer. Do this until the pointers pass each other, at which point swapping the pivot with the right number should position the pivot in its final, sorted position, where every number to its left is smaller and every number to its right is greater. Repeat the process mentioned on the respective subarrays located to the left and right of your pivot, and keep on repeating the process thereafter until the input array is fully sorted.

Optimal Space & Time Complexity

Best: O(nlog(n)) time | O(log(n)) space - where n is the length of the input array Average: O(nlog(n)) time | O(log(n)) space - where n is the length of the input array Worst: O(n^2) time | O(log(n)) space - where n is the length of the input array

Difficulty Hard: Heap Sort

Write a function that takes in an array of integers and returns a sorted version of that array. Use the Heap Sort algorithm to sort the array.



Hint:

Divide the input array into two subarrays in place. The second subarray should be sorted at all times and should start with a length of 0, while the first subarray should be transformed into a max (or min) heap and should satisfy the heap property at all times. Note that the largest (or smallest) value of the heap should be at the very beginning of the newly-built heap. Start by swapping this value with the last value in the heap; the largest (or smallest) value in the array should now be in its correct position in the sorted subarray, which should now have a length of 1; the heap should now be one element smaller, with its first element out of place. Apply the "sift down" method of the heap to re-position this out-of-place value. Repeat the step mentioned above, then the heap is left with only one value, at which point the entire array should be sorted.

Optimal Space & Time Complexity

Best: O(nlog(n)) time | O(1) space - where n is the length of the input array Average: O(nlog(n)) time | O(1) space - where n is the length of the input array Worst: O(nlog(n)) time | O(1) space - where n is the length of the input array

Difficulty Hard: Radix Sort

Write a function that takes in an array of non-negative integers and returns a sorted version of that array. Use the Radix Sort algorithm to sort the array.

Sample Input

```
array = [8762, 654, 3008, 345, 87, 65, 234, 12, 2]
```

Sample Output

```
[2, 12, 65, 87, 234, 345, 654, 3008, 8762]
```

Hint:

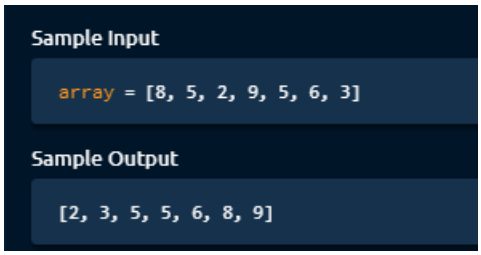
Radix Sort sorts numbers by looking only at one of their digits at a time. It first sorts all of the given numbers by their ones' column, then by their tens' column, then by their hundreds' column, and so on and so forth until they're fully sorted. Radix Sort uses an intermediary sorting algorithm to sort numbers one digits' column at a time. The goal of Radix Sort is to perform a more efficient sort than popular sorting algorithms like Merge Sort or Quick Sort for inputs that are well suited to be sorted by their individual digits' columns. With this in mind, what intermediary sorting algorithm should we use with Radix Sort? Keep in mind that this sorting algorithm will run multiple times, sorting one digits' column at a time. The most popular sorting algorithm to use with Radix Sort is Counting Sort. Counting Sort takes advantage of the fact that we know the range of possible values that we need to sort. When sorting numbers, we know that we only need to sort digits, which will always be in the range 0-9. Therefore, we can count how many times these digits occur and use those counts to populate a new sorted array. We'll perform counting sort multiple times, once for each digits' column that we're sorting, starting with the ones' column. We need to ensure that our counting sort performs a stable sort, so that we don't lose information from previous iterations of sorting. Counting sort runs in O(n) time, which means that we might have a much more efficient sorting algorithm if the largest number in our input contains few digits. See the Conceptual Overview section of this question's video explanation for a more in-depth explanation .

Optimal Space & Time Complexity

O(d * (n + b)) time | O(n + b) space - where n is the length of the input array, d is the max number of digits, and b is the base of the numbering system used

Difficulty Hard: Merge Sort

Write a function that takes in an array of integers and returns a sorted version of that array. Use the Merge Sort algorithm to sort the array.



Hint:

Merge Sort works by cutting an array in two halves, respectively sorting those two halves by performing some special logic, and then merging the two newly-sorted halves into one sorted array. The respective sorting of the two halves is done by reapplying the Merge Sort algorithm / logic on each half until single-element halves are obtained; these single-element arrays are sorted by nature and can very easily be merged back together. Divide the input array in two halves by finding the middle-most index in the array and slicing the two halves around that index. Then, recursively apply Merge Sort to each half, and finally merge them into one single, sorted array by iterating through their values and progressively adding them to the new array in ascending order. Your implementation of Merge Sort almost certainly uses a lot of auxiliary space and likely does not sort the input array in place. What is the space complexity of your algorithm? Can you implement a version of the algorithm using only one additional array of the same length as the input array, and can this version sort the input array in place?

Optimal Space & Time Complexity

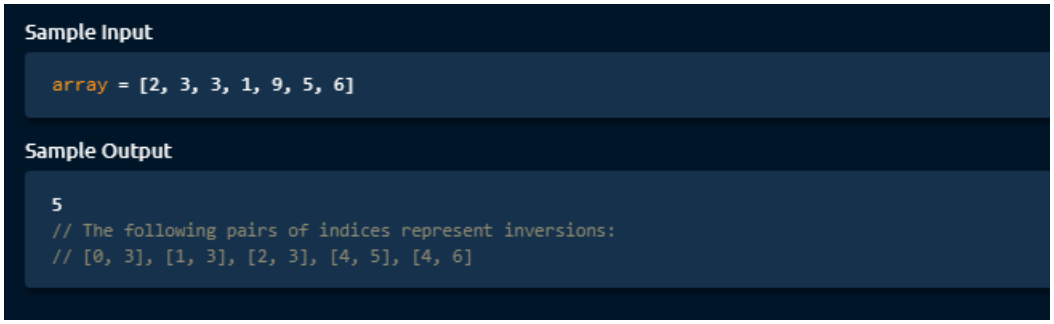
Best: O(nlog(n)) time | O(n) space - where n is the length of the input array
Average: O(nlog(n)) time | O(n) space - where n is the length of the input array
Worst: O(nlog(n)) time | O(n) space - where n is the length of the input array

Difficulty V. Hard : Count Inversions

Write a function that takes in an array of integers and returns the number of inversions in the array. An inversion occurs if for any valid indices i and j, i < j and array[i] > array[j].

For example, given array = [3, 4, 1, 2], there are 4 inversions. The following pairs of indices represent inversions: [0, 2], [0, 3], [1, 2], [1, 3].

Intuitively, the number of inversions is a measure of how unsorted the array is.



Hint:

The brute-force approach to solve this problem is to simply compare every pair of indices in the array and to determine how many of them represent inversions. This approach takes O(n^2) time, where n is the length of the array. Can you do better than this? If the number of inversions is the degree to which the array is unsorted, and it if it takes O(nlog(n)) time to sort an array using an optimal sorting algorithm, can you determine how unsorted the array is with a solution that runs in that time complexity? Try thinking about how you would solve this problem if, instead of being given one array, you were given two separate arrays representing the main array's two halves. You would need to determine the number of inversions in the array created by merging the left array and the right array. The number of inversions in this example is actually equal to the number of inversions in the left array, the number of inversions in the right array, and the number of inversions when you merge the sorted left array and the sorted right array. Recall how Merge Sort works for a hint about how you can solve this problem. Once you understand the information stated above, you can use an algorithm that's very similar to Merge Sort to determine the number of inversions in any array. You'll recursively determine the number of inversions in the left and right halves of an array while sorting both the left and right halves, just like you do in Merge Sort. Once your two

halves are sorted, you'll merge them together and count the number of inversions in the merged array. Take the example of these two sorted arrays: a1 = [1, 3, 4] and a2 = [2, 2, 5]. When you merge these two sorted arrays, you insert elements from the left and right array into one larger array. Whenever you insert an element from the right array before inserting an element from the left array, that means an inversion or multiple inversions have occurred. This is because elements in the right array are positioned after all elements in the left array (if these two arrays were originally left and right halves of another array). The remaining elements to be inserted from the left array when we insert an element from the right array are all inverted with this right-array element. See the Conceptual Overview section of this question's video explanation for a more in-depth explanation.

Optimal Space & Time Complexity:

O(nlogn) time | O(n) space - where n is the length of the array