Sorting Exercises

Download Starter Code <../dsa-sorting.zip>

bubbleSort

Implement a function called bubbleSort. Given an array, bubbleSort will sort the values in the array.

Bubble sort is an O(n^2) algorithm. You can learn about Bubble Sort <https://www.rithmschool.com/courses/javascript-computer-science-fundamentals/basic-sorting-algorithms>

```
bubbleSort([4, 20, 12, 10, 7, 9]); // [4, 7, 9, 10, 12, 20]
bubbleSort([0, -10, 7, 4]); // [-10, 0, 4, 7]
bubbleSort([1, 2, 3]); // [1, 2, 3]
bubbleSort([]);

let nums = [
    4, 3, 5, 3, 43, 232, 4, 34, 232, 32, 4, 35, 34,
    23, 2, 453, 546, 75, 67, 4342, 32
];

bubbleSort(nums); // [2, 3, 3, 4, 4, 4, 5, 23, 32, 32, 34, 34, 35, 43, 67,
                  // 75, 232, 232, 453, 546, 4342]
```

merge

Given two sorted arrays, write a function called merge which accepts two *sorted* arrays and returns a new array with values from both arrays sorted.

This function should run in O(n + m) time and O(n + m) space and should not modify the parameters passed to it.

Also, do not use the built in ***.sort()*** method! We're going to use this function to implement a sort, so the helper itself shouldn't depend on a sort.

```
let arr1 = [1,3,4,5];
let arr2 = [2,4,6,8];
merge(arr1,arr2) // [1,2,3,4,4,5,6,8]

arr1 // [1,3,4,5];
arr2 // [2,4,6,8];

let arr3 = [-2,-1,0,4,5,6];
let arr4 = [-3,-2,-1,2,3,5,7,8];

merge(arr3,arr4); // [-3,-2,-2,-1,-1,0,2,3,4,5,5,6,7,8]

let arr5 = [3,4,5]
let arr6 = [1,2]

merge(arr5,arr6) // [1,2,3,4,5]
```

mergeSort

Implement the merge sort algorithm. Given an array, this algorithm will sort the values in the array. Here's some guidance for how merge sort should work:

- Break up the array into halves until you can compare one value with another
- Once you have smaller sorted arrays, merge those with other sorted pairs until you are back at the full length of the array
- Once the array is merged back together, return the merged (and sorted!) array
- In order to implement this function, you'll also need to implement a merge function that takes in two sorted arrays and returns a new sorted array. You implemented this function in the previous exercise, so use that function!

You can read more about Merge Sort <https://www.rithmschool.com/courses/javascript-computer-science-fundamentals/intermediate-sorting-algorithms>

```
mergeSort([4, 20, 12, 10, 7, 9]); // [4, 7, 9, 10, 12, 20]
mergeSort([0, -10, 7, 4]); // [-10, 0, 4, 7]
```

```
mergeSort([1, 2, 3]); // [1, 2, 3]
mergeSort([]);

let nums = [
    4, 3, 5, 3, 43, 232, 4, 34, 232, 32, 4, 35, 34, 23, 2,
    453, 546, 75, 67, 4342, 32
];

mergeSort(nums); // [2, 3, 3, 4, 4, 4, 5, 23, 32, 32, 34, 34, 35,
                 //  43, 67, 75, 232, 232, 453, 546, 4342]
```

Further Study

insertionSort

Here's some guidance for how insertion sort should work:

- Start by picking the second element in the array (we will assume the first element is the start of the "sorted" portion)
- Now compare the second element with the one before it and swap if necessary.
- Continue to the next element and if it is in the incorrect order, iterate through the sorted portion to place the element in the correct place.
- Repeat until the array is sorted.

```
insertionSort([4, 20, 12, 10, 7, 9]); // [4, 7, 9, 10, 12, 20]
insertionSort([0, -10, 7, 4]); // [-10, 0, 4, 7]
insertionSort([1, 2, 3]); // [1, 2, 3]
insertionSort([]);

let nums = [
    4, 3, 5, 3, 43, 232, 4, 34, 232, 32, 4, 35, 34, 23, 2,
    453, 546, 75, 67, 4342, 32
];

insertionSort(nums); // [2, 3, 3, 4, 4, 4, 5, 23, 32, 32, 34,
                     //  34, 35, 43, 67, 75, 232, 232, 453, 546, 4342]
```

selectionSort

Here's some guidance for how selection sort should work:

- Assign the first element to be the smallest value (this is called the minimum). It does not matter right now if this actually the smallest value in the array.
- Compare this item to the next item in the array until you find a smaller number.
- If a smaller number is found, designate that smaller number to be the new "minimum" and continue until the end of the array.
- If the "minimum" is not the value (index) you initially began with, swap the two values. You will now see that the beginning of the array is in the correct order (similar to how after the first iteration of bubble sort, we know the rightmost element is in its correct place).
- Repeat this with the next element until the array is sorted.

This algorithm has a O(n^2) time complexity. You can read more Selection Sort <https://www.rithmschool.com/courses/javascript-computer-science-fundamentals/basic-sorting-algorithms>.

```
selectionSort([4, 20, 12, 10, 7, 9]); // [4, 7, 9, 10, 12, 20]
selectionSort([0, -10, 7, 4]); // [-10, 0, 4, 7]
selectionSort([1, 2, 3]); // [1, 2, 3]
selectionSort([]);

let nums = [
    4, 3, 5, 3, 43, 232, 4, 34, 232, 32, 4, 35, 34, 23, 2,
    453, 546, 75, 67, 4342, 32
];

selectionSort(nums); // [2, 3, 3, 4, 4, 4, 5, 23, 32, 32, 34, 34,
                     //  35, 43, 67, 75, 232, 232, 453, 546, 4342]
```

pivot

In this exercise, your goal is to implement a function called ***pivot***. This function contains nearly all of the logic you'll need in order to implement Quick Sort in the next exercise.

The ***pivot*** function is responsible for taking an array, setting the pivot value, and mutating the array so that all values less than the pivot wind up to the left of it, and all values greater than the pivot wind up to the right of it. It's also helpful if this helper returns the index of where the pivot value winds up.

For example, if we decide the pivot will always be the first element in the array, it should behave in the following way:

```
let arr = [4, 2, 5, 3, 6];
pivot(arr); // 2
arr; // [3, 2, 4, 5, 6];
```

In this code, the specifics of how the ***arr*** variable gets mutated are not important. All that matters is that 4 winds up at index 2, with 3 and 2 to the left of it (in any order), and with 5 and 6 to the right of it (in any order).

```
let arr1 = [5, 4, 9, 10, 2, 20, 8, 7, 3];
let arr2 = [8, 4, 2, 5, 0, 10, 11, 12, 13, 16];

pivot(arr1); // 3
pivot(arr2); // 4

arr1.slice(0, 3).sort((a, b) => a - b); // [2, 3, 4]
arr1.slice(3).sort((a, b) => a - b); // [5, 7, 8, 9, 10, 20]

arr2.slice(0, 4).sort((a, b) => a - b); // [0, 2, 4, 5]
arr2.slice(4).sort((a, b) => a - b); // [8, 10, 11, 12, 13, 16]
```

quickSort

The next sorting algorithm we'll consider is Quick Sort. Unfortunately, quicksort is not the most intuitive of algorithms and has a wide range of implementations. It may help to check out this great video from Computerphile for a quick introduction to how quicksort works <https://www.youtube.com/watch?v=XE4VP_8Y0BU>.

The algorithm is as follows:

- Pick an element in the array and designate it as the “pivot”. While there are quite a few options for choosing the pivot. We’ll make things simple to start, and will choose the pivot as the first element. This is not an ideal choice, but it makes the algorithm easier to understand for now.
- Next, compare every other element in the array to the pivot.
- If it’s less than the pivot value, move it to the left of the pivot.
- If it’s greater, move it to the right.
- Once you have finished comparing, the pivot will be in the right place.
- Next, recursively call quicksort again with the left and right halves from the pivot until the array is sorted.

It’s easiest to implement Quick Sort with the aid of your pivot helper from the earlier exercise. This function is responsible for taking an array, setting the pivot value, and mutating the array so that all values less than the pivot wind up to the left of it, and all values greater than the pivot wind up to the right of it. It’s also helpful if this helper returns the index of where the pivot value winds up.

```
quickSort([4, 20, 12, 10, 7, 9]); // [4, 7, 9, 10, 12, 20]
quickSort([0, -10, 7, 4]); // [-10, 0, 4, 7]
quickSort([1, 2, 3]); // [1, 2, 3]
quickSort([]);

let nums = [
    4, 3, 5, 3, 43, 232, 4, 34, 232, 32, 4, 35, 34, 23,
    2, 453, 546, 75, 67, 4342, 32
];

quickSort(nums); // [2, 3, 3, 4, 4, 4, 5, 23, 32, 32, 34, 34,
                 //  35, 43, 67, 75, 232, 232, 453, 546, 4342]
```

radixSort

Write a function called radixSort which accepts an array of numbers and sorts them in ascending order.

```
radixSort([8, 6, 1, 12]);
// [1, 6, 8, 12]

radixSort([10, 100, 1, 1000, 10000000]);
// [1, 10, 100, 1000, 10000000]

radixSort([902, 4, 7, 408, 29, 9637, 1556, 3556, 8157, 4386, 86, 593]);
// [4, 7, 29, 86, 408, 593, 902, 1556, 3556, 4386, 8157, 9637]
```

Additional Sorting algorithms

If you are curious, here a few more interesting sorting algorithms to discover

- heap sort
- shell sort
- counting sort

Solution

View our solutions <solution/index.html>