**Sorting Algorithms-II**

**VisualAlog:** [*https://visualgo.net/en/sorting*](https://visualgo.net/en/sorting)

**SortingAnimationSite:** [*https://www.toptal.com/developers/sorting-algorithms*](https://www.toptal.com/developers/sorting-algorithms)

**(Merge Sort, Quick Sort, Radix Sort)**

**What is sorting?**

Sorting is the process of rearranging the items in a collection (e.g. an array) so that the items are in some kind of order.

The sorting algorithms we've learned so far don't scale well.   
There is a family of sorting algorithms that can improve time complexity from **O(n2)** to **O(n log n)**.

Here we’re going to learn Intermediate sorting algorithms:-

**Merge** Sort

**Quick** Sort

**Radix** Sort

**Merge Sort**

* It's a combination of two things - merging and sorting!
* Exploits the fact that arrays of 0 or 1 element (Empty or Array having Single Element) are always sorted.
* Works by decomposing an array into smaller arrays of 0 or 1 elements, then building up a newly sorted array. So that’s something knows as a divide and conquer algorithm, a divide and conquer approach.

**Merging Arrays Pseudocode**

* Create an empty array, take a look at the smallest values in each input array
* While there are still values we haven't looked at...
* If the value in the first array is smaller than the value in the second array, push the value in the first array into our results and move on to the next value in the first array
* If the value in the first array is larger than the value in the second array, push the value in the second array into our results and move on to the next value in the second array
* Once we exhaust one array, push in all remaining values from the other array

**Merging Array Code:**

function mergeSort(arr1, arr2) {

let i=0,  
 j=0,  
 result = [];

while(i<arr1.length && j<arr2.length){  
 if(arr1[i]<arr2[j]){  
 result.push(arr1[i]);  
 i++;

}else{  
 result.push(arr2[j]);  
 j++;  
 }

}

while(i<arr1.length){  
 result.push(arr1[j]);  
 i++;

}

while(j<arr2.length){

result.push(arr2[j]);  
 j++;

}

return result;

}

merge([1,10,50], [2,14,99,100]);

**Output:**

[1, 2, 10, 14, 50, 99, 100]

**mergeSort Pseudocode**

* Create an empty array, take a look at the smallest values in each input array
* While there are still values we haven't looked at...
* If the value in the first array is smaller than the value in the second array, push the value in the first array into our results and move on to the next value in the first array
* If the value in the first array is larger than the value in the second array, push the value in the second array into our results and move on to the next value in the second array
* Once we exhaust one array, push in all remaining values from the other array

**MergeSort Code:**

**//---- Merging Arrays Code ---------**

function merge(arr1, arr2) {

let i=0,

j=0,

result = [];

while(i<arr1.length && j<arr2.length){

if(arr1[i]<arr2[j]){

result.push(arr1[i]);

i++;

}else{

result.push(arr2[j]);

j++;

}

}

while(i<arr1.length){

result.push(arr1[i]);

i++;

}

while(j<arr2.length){

result.push(arr2[j]);

j++;

}

return result;

}

**//---- MergeSort Code ---------**

function mergeSort(arr) {

if(arr.length<=1) return arr;

let mid = Math.floor(arr.length/2),

left = mergeSort(arr.slice(0, mid)),

right = mergeSort(arr.slice(mid));

return merge(left, right);

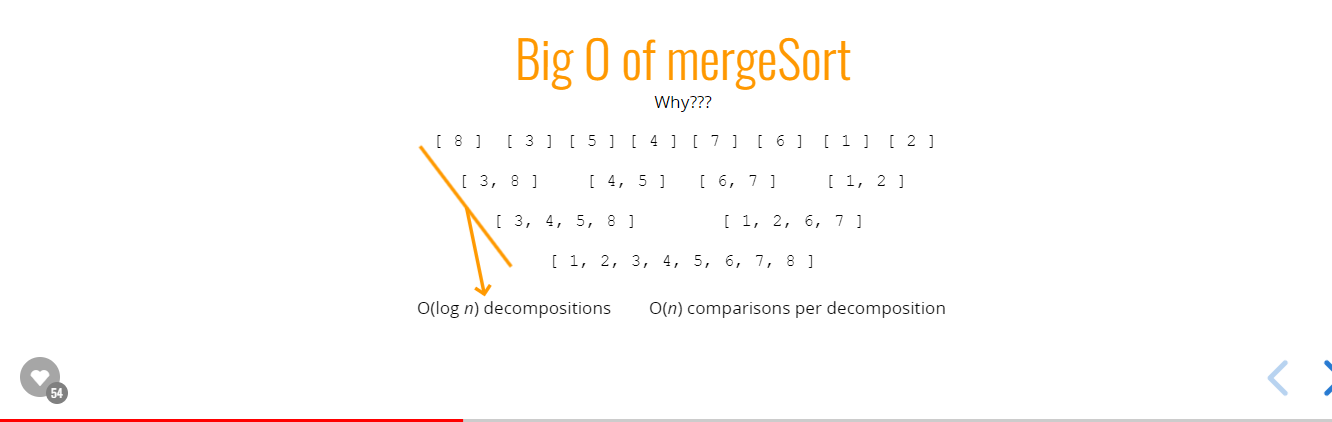
}

mergeSort([10,24,76,73]);

**Output:**[10, 24, 73, 76]

**Note:**

* In order to implement merge sort, it's useful to **first implement** a function responsible arranging elements in an array i.e. (merge helper function).



**Big O of MergeSort Algorithms**

|  |  |  |  |
| --- | --- | --- | --- |
| **Time Complexity (Best)** | **Time Complexity (Average)** | **Time Complexity (Worst)** | **Space Complexity** |
| O(*nlogn*) | O(*nlogn*) | O(*nlogn*) | O(*nlogn*) |

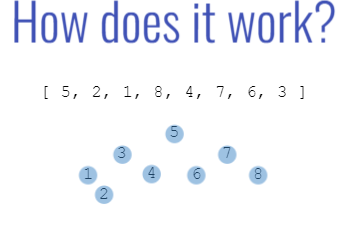
**Private Understanding Note for MergeSort():**

<file:///F:/xampp/htdocs/websites/Learning/JavaScript/Data%20Strucutre%20&%20Algorithm%20(DSA)/Algorithm%20&%20DataStructure%20(Merge)/Theory/Private%20Detail%20Notes%20for%20Understanding/MergeSort().pdf>

**Quick Sort**

Works by selecting one element (called the "pivot") and finding the index where the pivot should end up in the sorted array

Once the pivot is positioned appropriately, quick sort can be applied on either side of the pivot



**Pivot Helper Function:-**

* It is useful to first implement pivot helper function, in order to implement quick sort
* The runtime of quick sort depends in part on how one selects the pivot
* Ideally, the pivot should be chosen so that it's roughly the median value in the data set you're sorting
* For simplicity, we'll always choose the pivot to be the first element (we'll talk about consequences of this later).

**Pivot Pseudo Code:**

* It will help to accept three arguments: an array, a start index, and an end index (these can default to 0 and the array length minus 1, respectively)
* Grab the pivot from the start of the array
* Store the current pivot index in a variable (this will keep track of where the pivot should end up)
* Loop through the array from the start until the end
* If the pivot is greater than the current element, increment the pivot index variable and then swap the current element with the element at the pivot index
* Swap the starting element (i.e. the pivot) with the pivot index
* Return the pivot index

**//Pivot Helper function Code**

function pivot(arr, startIdx = 0){

function swap(array, i, j) {

let temp = array[i];

array[i] = array[j];

array[j] = temp;

}

let pivot = arr[startIdx],

swapIdx = startIdx;

for (let i = 1; i < arr.length; i++) {

if(pivot>arr[i]){

swapIdx++;

swap(arr, swapIdx,i);

}

}

swap(arr, startIdx, swapIdx);

console.log(arr);

return swapIdx;

}

pivot([4,8,2,1,5,7,6,3]);

**Output:**

[3, 2, 1, 4, 5, 7, 6, 8]  
3

**QuickSort Pseudo Code:-**

* Call the pivot helper on the array
* When the helper returns to you the updated pivot index, recursively call the pivot helper on the subarray to the left of that index, and the subarray to the right of that index
* Your base case occurs when you consider a subarray with less than 2 elements

**QuickSort Implementation Code:-**

**//Pivot Helper function**

function pivot(arr, startIdx = 0, endIdx = arr.length-1){

function swap(array, i, j) {

let temp = array[i];   
 array[i] = array[j];   
 array[j] = temp;

}

let pivot = arr[startIdx],   
 swapIdx = startIdx;

for (let i = startIdx+1; i < arr.length; i++) {

if(pivot>arr[i]){   
 swapIdx++;   
 swap(arr, swapIdx,i);

}

}

swap(arr, startIdx, swapIdx);

return swapIdx;

}

**//QuickSort function**

function quickSort(arr, left=0, right=arr.length-1) {

if(left<right){

let pivotIdx = pivot(arr, left, right);

***//Left***

quickSort(arr, left, pivotIdx-1);

***//Right***

quickSort(arr, pivotIdx+1, right);

}

return arr;

}

quickSort([4,6,9,1,2,5]);

**Output:**  
[1, 2, 4, 5, 6, 9]

**Big O of QuickSort Algorithms**

|  |  |  |  |
| --- | --- | --- | --- |
| **Time Complexity (Best)** | **Time Complexity (Average)** | **Time Complexity (Worst)** | **Space Complexity** |
| O(*nlogn*) | O(*nlogn*) | O(*n2*) | O(*logn*) |

**Private Understanding Note for QuickSort():**

<file:///F:/xampp/htdocs/websites/Learning/JavaScript/Data%20Strucutre%20&%20Algorithm%20(DSA)/Algorithm%20&%20DataStructure%20(Merge)/Theory/Detail%20Notes%20for%20Some%20Imp%20Topics%20for%20Understanding/quickSort().pdf>

**Note:**

* QuickSort doesn’t go well with sorted array, because it give time complexity of **O(n2)**. So to avoid this we have to take pivot value as the middle one of the array instead of taking the first element of the array.
* **COMPARISON SORTS:** (Average Time Complexity)Bubble Sort - O(n^2)

Insertion Sort - O(n^2)

Selection Sort - O(n^2)

Quick Sort - O(n log (n))

Merge Sort - O(n log (n))

**Radix Sort**

Radix sort is a special sorting algorithm that works on lists of numbers

It exploits the fact that information about the size of a number is encoded in the number of digits.

It never makes comparisons between elements. So it sorts given list of numbers upon O(nlogn) but not by comparisons between numbers.

**Radix Sort Helper Functions:**

In order to implement radix sort, it's helpful to build a few helper functions first:

1. **getDigit(num, place)** - returns the digit in num at the given place value  
   getDigit(12345, 0); // 5

getDigit(12345, 1); // 4

getDigit(12345, 2); // 3

getDigit(12345, 3); // 2

getDigit(12345, 4); // 1

getDigit(12345, 5); // 0  
  
Code:  
function getDigit(num, i) {

return Math.floor(Math.abs(num) / Math.pow(10, i)) % 10;

}  
**Note:**-- Math.abs() 🡪 Gives the absolute value or the modulus of a Num. Ex: |5|  
-- Math.pow(10, 2) 🡪 Gives the power of a num. Ex: 102 =100  
-- 12345 🡪 Start with 0th place. Ex: (12345/100)%10 == (12345/1)%10 = 5  
-- 1234.5 🡪 Start with 10th place just before decimal.  
 Ex: (12345/101 )%10 == (12345/10 )%10 = 4.5 🡪 Math.floor(4.5) == 4

1. **digitCount(num)** - returns the number of digits in num  
   digitCount(1); // 1

digitCount(25); // 2

digitCount(314); // 3  
  
Code:  
function getCount(num) {  
 if(num===0) return 1;

return Math.floor(Math.log10(Math.abs(num))) + 1;

}  
**Note:**-- Math.log10(|num|) 🡪 Means, **Log10(num)**. Ex: log­10(12345) = 4.09149109  
-- Math.floor(Math.log10(Math.abs(num))): Gives **Length of (+ve num) -1**

1. **mostDigits(nums)** - Given an array of numbers, returns the number of digits of the largest numbers in the list  
   mostDigits([1234, 56, 7]); // 4

mostDigits([1, 1, 11111, 1]); // 5

mostDigits([12, 34, 56, 78]); // 2  
  
Code:  
function mostDigitCount(num){

let mostDigit = 0;

for (let i = 0; i < num.length; i++) {

mostDigit = Math.max(mostDigit, digitCount(num[i]));

}

return mostDigit;

}  
**Note:**-- mostDigitCount function depend upon the above function digitCount.

**Radix Sort Pseudo Code:**

* Define a function that accepts list of numbers
* Figure out how many digits the largest number has
* Loop from k = 0 up to this largest number of digits
* For each iteration of the loop:
* Create buckets for each digit (0 to 9)
* place each number in the corresponding bucket based on its kth digit
* Replace our existing array with values in our buckets, starting with 0 and going up to 9
* return list at the end!

**Radix Sort Code:**

**//getDigit**  
function getDigit(num, i) {

return Math.floor(Math.abs(num)/Math.pow(10, i))%10;

}

**//digitCount**function digitCount(num) {

if(num ===0) return 1;

return Math.floor(Math.log10(Math.abs(num))) + 1;

}

**//mostDigitCount**  
function mostDigitCount(nums){

let maxDigit = 0;

for (let i = 0; i < nums.length; i++) {

maxDigit = Math.max(maxDigit, digitCount(nums[i]));

}

return maxDigit;

}

function radixSort(nums) {

let maxDigitCount = mostDigitCount(nums); ***// largest number of digits***

for (let k = 0; k < maxDigitCount; k++) {

***//Create 10 Empty Array upto k times.***

let digitBucket = Array.from({length:10}, ()=>[]);

for (let i = 0; i < nums.length; i++) {

***//Getting Digit of Num at Kth position. For Ex: 2 at 0th position of 9852***

let digit = getDigit(nums[i], k);

digitBucket[digit].push(nums[i]);

}

// console.log(digitBucket);

***//Get the sorted numbers from the digitBucket for every Iteration.***

nums = [].concat(...digitBucket);

// console.log(nums);

}

return nums;

}

radixSort([23,345,5476,12,2345,9852]);

**Output:**  
[12, 23, 345, 2345, 5476, 9852]

**Big O of RadixSort Algorithms**

|  |  |  |  |
| --- | --- | --- | --- |
| **Time Complexity (Best)** | **Time Complexity (Average)** | **Time Complexity (Worst)** | **Space Complexity** |
| O(*nk*) | O(*nk*) | O(*nk*) | O(*n+k*) |
| Where, |  |  |  |

**n** - length of array  
**k** - number of digits (average)

Means, Time Complexity would be **O(n) in Radix**, if element of an array of length 1. Ex: arr = [2,6,3,7,2,7,4]

**Big O of Sorting Algorithms**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Time Complexity (Best)** | **Time Complexity (Average)** | **Time Complexity (Worst)** | **Space Complexity** |
| Merge Sort | O(*n logn*) | O(*n log(n)*) | O(*n log(n)*) | O(n) |
| Quick Sort | O(*n log(n)*) | O(*n log(n)*) | O(*n2*) | O(logn) |
| Radix Sort | O(*n*k) | O(*n*k) | O(*n*k) | O(n+k) |

**Conclusion**

* Merge sort and quick sort are standard efficient sorting algorithms
* Quick sort can be slow in the worst case, but is comparable to merge sort on average
* Merge sort takes up more memory because it creates a new array (in-place merge sorts exist, but they are really complex!)
* Radix sort is a fast sorting algorithm for numbers
* Radix sort exploits place value to sort numbers in linear time (for a fixed number of digits)



**SortingAnimationSite:** [*https://www.toptal.com/developers/sorting-algorithms*](https://www.toptal.com/developers/sorting-algorithms)