**Merge Sort**

*Definition:*

* It’s a combination of two things – merging and sorting!
* Exploits the fact that arrays of 0 or 1 elements are always sorted
* Works by decomposing an array into smaller arrays of 0 or 1 element, then building up a newly sorted array

*Graph:*

[8, 3, 5, 4, 7, 6, 1, 2] => [8, 3, 5, 4] and [7, 6, 1, 2] => [8,3] and [5,4] and [7,6] and [1,2]   
 => [8], [3], [5], [4], [7], [6], [1], [2] => [3,8] and [4,5] and [6,7] and [1,2] => [3,4,5,8] and [1,2,6,7]   
 => [1,2,3,4,5,6,7,8]

*Pseudocode for Merge Sort Helper:*

* 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

*Solution for Merge Sort Helper:*

function merge(arr1, arr2) {

let result = [];

let i = 0;

let j = 0;

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

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

result.push(arr1[i]);

i++;

} else {

result.push(arr2[j]);

j++;

}

}

while(i < arr1.length) {

result.push(arr1[i]);

}

while(j < arr2.length) {

result.push(arr2[j]);

}

return result;

}

*Pseudocode for Merge Sort:*

* Break up the array into halves until you have arrays that are empty or have one element (use recursion)
* Once you have smaller sorted arrays, merge those arrays with other sorted arrays until you are back at the full length of the array
* Once the array has been merged back together, return the merged (and sorted) array

*Solution for Merge Sort:*

function mergeSort(arr) {

if (arr.length <= 1) {

return arr;

}

var mid = Math.floor(arr.length / 2);

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

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

merge(left, right);

}

*Big O:* We have O(log n) decompostions and Olog(n) comparisons per decomposition => O(nlog n)

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

**Quick Sort**

*Definition:*

* Like merge sort, exploits the fact that arrays of 0 or 1 elements are always sorted
* 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 to either side of the pivot

*Graph (nên coi lại trên VisuAlgo):*

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

*Pseudocode for Pivot Helper:*

* It will help to accept three arguments: an array, a start index, and an end index (these can be 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 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

*Solution for Pivot:*

function pivot(arr, start=0, end=arr.length - 1) {

function swap(array, i, j) {

var temp = arr[i];

arr[i] = arr[j];

arr[j] = temp;

}

var pivot = arr[start];

var swapIdx = start;

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

if (pivot > arr[i]) {

swapIdx++;

swap(arr, swapIdx, i);

}

}

swap(arr, start, swapIdx);

return swapIdx;

}

*Pseudocode for Quick Sort:*

* 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

*Solution for Quick Sort:*

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

if (left < right) {

var pivotIndx = pivot(arr, left, right);

quicksort(arr, left, pivotIndx-1);

quicksort(arr,pivotIndx+1, right);

}

Return arr;

}

*Big O:*

|  |  |  |  |
| --- | --- | --- | --- |
| **Time Complexity**  **(Best)** | **Time Complexity (Average)** | **Time Complexity (Worst)** | **Space Complexity** |
| O(nlog n) | O(nlogn) | O(n^2) | O(logn) |

**Radix Sort**

*Definition:*

* Radix sort is a special sorting algorithm that works on lists of numbers
* It never makes comparisons between elements
* It exploits the fact that information about the size of a number is encoded in the number of digits
* More digits mean a bigger number

*Graph (look back lecture 99 Udemy)*

*Radix Sort’s Helper:*

getDigit(num, place) – return the digit in num at the given place value

digitCount(num) – return the number of digits in num

mostDigits(nums) –returns the number of digits in the largest numbers in the array

function getDigit(num, i) {

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

}

function digitCount(num) {

if (num === 0) return 0

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

}

function mostDigits(arr) {

var maxDigits = 0;

for (var i = 0; i < arr.length; i++) {

maxDigits = Math.max(maxDigits, digitCount(arr[i]));

}

return maxDigits;

}

*Pseudocode for Radix Sort:*

* 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

*Solution for Radix Sort:*

function radixSort(arr) {

var mostDigits = mostDigits(arr);

for (var k = 0; k < mostDigits; i++) {

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

for (var i = 0; i < arr.length; i++) {

var digit = getDigit(num[i], k);

digitBuckets[digit].push(arr[i]);

}

arr = [].concat(…digitBuckets);

}

return arr;

}

*Big O:* n – length of array, k – number of digits (average)

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