Chapter 6 : Part 5

**Sorting Algorithms: Merge Sort**

**6.5 Advanced Sorting Algorithms**

Actually these are some *Intermediate Sorting Algorithms* (not quite advanced) so don't be too scared!

* We'll see the Pseudocodes, but we focus on walk-through the algorithm, to understand the intuition behind.
* Objectives
* Understand the ***limitations of the sorting algorithms*** we've learned so far, such as: *Bubble*, *Insertion* and *Selection*.
* Implement Merge Sort
* Implement Quick Sort
* Implement Radix Sort
* WHY LEARN THIS?
* The sorting algorithms we've learned so far don't scale well: In real world where we dealing with 100,000 of data sorting, the **O(n2)** algorithms like **Bubble**, **Selection** or **Insertion** are very slow. For example in JS, sorting, **100000** data takes more than **20secs** using **Bubble** **sort**.
* But if we use Merge Sort, it can only take **2** or **3 secs**.
* Try out bubble sort on an array of 100000 elements, it will take quite some time! So in this case *Time Complexity* for *Bubble Sort* is **O(n2)** where *Time Complexity* for *Merge Sort* is .
* FASTER SORTS
* There is a family of sorting algorithms that can improve time complexity from **O(n2)** to **.**
* There's a tradeoff between efficiency and simplicity. The more *efficient algorithms* are much *less simple*, and generally take longer *to* *understand* .

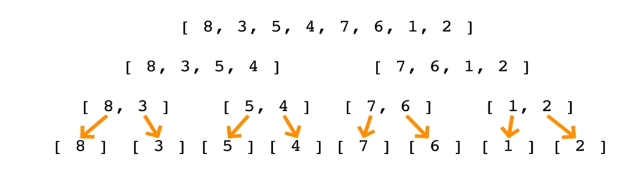
**6.5.1 Merge Sort**

It is created in 1948. By Newman. The idea behind it is:

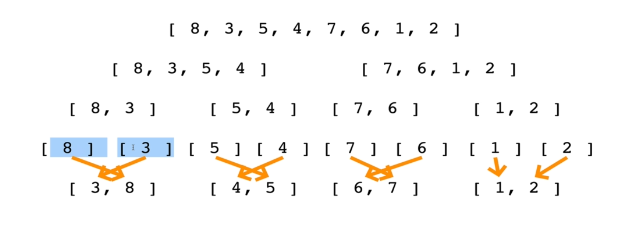
* It's a combination of two things - merging and sorting! We can say 3 things: Splitting, Sorting, Merging.
* Exploits the fact that arrays of 0 or 1 element are always sorted. I.e array of no element or single element is sorted. This *assumption* is *used* in *Merge Sort*.
* Works by decomposing an array into smaller arrays of ***0*** or ***1*** elements, then building up a newly sorted array.
* That is we'll use Divide and Concur strategy. We split-up the larger array into smaller arrays until we get arrays of single/no element.
* For example we split an array of 8-element to 8 sub-array of single elements and then we merge them.

**6.5.2 How it works**

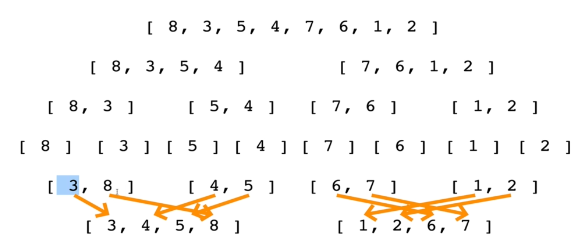
* First we split the array



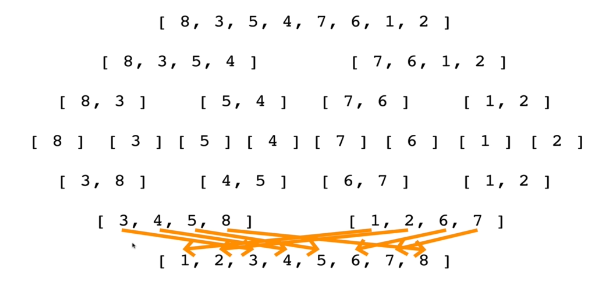
* Now we know, these single element arrays are sorted. The fact is merging two sorted arrays is easier than merging two unsorted arrays.
* During merge, we compare the values, like smaller goes first.



* We Repeat the process: We merge, two 2-elemet-sorted-arrays. And during Merge, we compare each values of both arrays and we sort them.



* In final merge we put them all together:



* Now lets see, **VISUALGO** representation.

|  |  |
| --- | --- |
|  |  |
|  |  |
|  |  |

**6.5.3 Part 1 : Merging functionality**

Now we first focus on Merging functionality of Two arrays (we'll do for two sorted arrays).

1. Assume that we have 2 sorted arrays, and we have to return the combination of those two sorted arrays.
2. Resulting array will be also sorted.

For example, we'll create a function **merge()**. We can use it as:

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

Results:

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

* In order to implement merge sort, it's useful to first implement a function responsible for merging two sorted arrays.
* Given two arrays which are sorted, this helper function should create a new array which is also sorted, and consists of all of the elements in the two input arrays.
* What we'll do if two are not same size?
* This function should run in **O(n + m)** time and **O(n + m)** space and should not modify the parameters passed to it.
* **n** is size of first array, **m** is for second. If **n= 100** and **m = 1000** it should work as **n+m**.
* **O(n)** means you should visit each element of the array only one time. But **O(n+m)** means you should visit each element of each array, only one time
* Pseudocode For Merging two Arrays:

1. Create an empty array, take a look at the smallest values in each input array.

* Define a function takes two inputs, and an empty array to return at the end.
* Start with smallest from each array from the beginning. Use while loop.

1. 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
* Example:

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

We start **i, j** at index 0 of both arrays, i.e. values **1** and **2**.

Since 1<2, **result = [1]**, since firstArray < secondArray, we move to 10, 2nd value of first array.

Since 10>2, **result = [1, 2]**, since firstArray > secondArray, we move to 14, 2nd value of second array.

Since 10<14, **result = [1, 2, 10]**, since firstArray < secondArray, we move to 50, 3rd value of first array.

Since 50>14, **result = [1, 2, 10, 14]**, since firstArray > secondArray, we move to 99, 3rd value of second array.

Since 50<99, **result = [1, 2, 10, 14, 50]**, since firstArray < secondArray, we move to 100, 4th value of second array.

Now we exhausted the first array, since both arrays were sorted already (both same way, either ascending or descending), we now push the remaining values **99** and **100** of the second array to our result. **result = [1, 2, 10, 14, 50, 99, 100]**

// *Merges two already sorted arrays*

**function** **merge**(arr1, arr2){

**let** results=[];

**let** i=0;

**let** j=0;

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

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

            results**.push**(arr1[i]);

            i++;

        } **else** {

            results**.push**(arr2[j])

            j++;

        }

    }

**while**(i **<** arr1**.**length) {

        results**.push**(arr1[i])

        i++;

    }

**while**(j **<** arr2**.**length) {

        results**.push**(arr2[j])

        j++;

    }

**return** results;

}

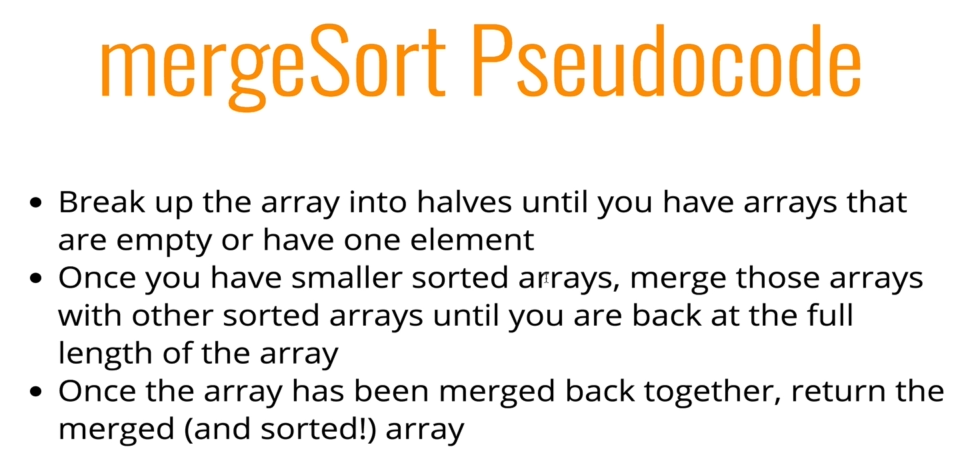
**merge**([100,200], [1,2,3,5,6])

* **while**(i **<** arr1**.**length **&&** j **<** arr2**.**length) works on both arrays until one is exhausted.
* **while**(i **<** arr1**.**length) Works if second array is exhausted and first array is larger.
* **while**(j **<** arr2**.**length) Works if second array is exhausted and first array is larger.
* Notice **i, j** were defined outside of while loops, so when first while loop ends, **i,j** still have changed values.

**6.5.4 Pseudocode Merge Sort**

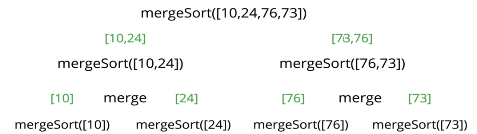
The code is not too big. But it's little difficult to understand. It mostly uses Recursion.

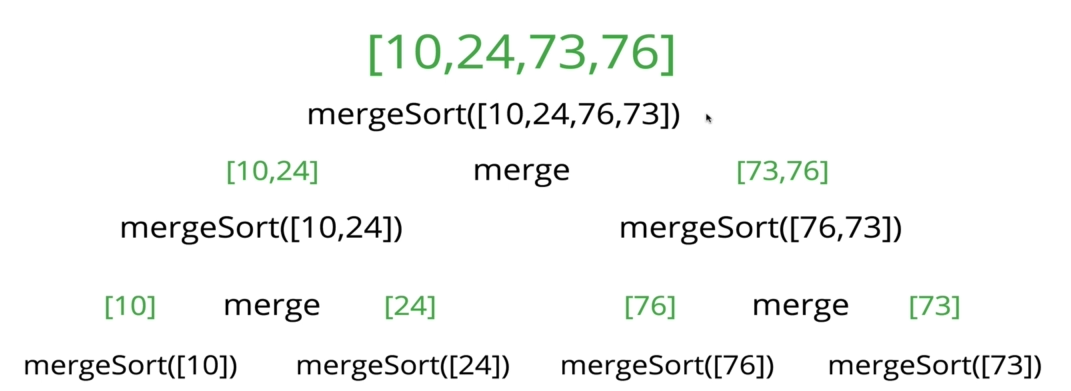
1. Break the array into two half. We cann use here **array.slice()**, index 0 to middle index of the array.
2. Then we call again the ***mergeSort()*** to break these two arrays into their own halfs.
3. We repaeat this process until the base case: where each ***array length*** is ***1*** or ***0***.
4. We then merge back the arrays using **merge()**. During merge the array gets sorted.
5. Finally we get the sorted array.



|  |  |
| --- | --- |
| * First all the left branch will be calculated. | * After base condition's left is done, it calculate the right side. |

* Similarly, top layers right portion will be calculated. Also left portion will calculated first for the following layers, until it hits the base case.





* To view debugging in vs code, install node.js. Then hit f5.

All code at once

// *Merge function from earlier*

**function** **merge**(arr1, arr2){

**let** results=[];

**let** i=0;

**let** j=0;

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

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

            results**.push**(arr1[i]);

            i++;

        } **else** {

            results**.push**(arr2[j])

            j++;

        }

    }

**while**(i **<** arr1**.**length) {

        results**.push**(arr1[i])

        i++;

    }

**while**(j **<** arr2**.**length) {

        results**.push**(arr2[j])

        j++;

    }

**return** results;

}

// *Recrusive Merge Sort*

**function** **mergeSort**(arr){

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

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

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

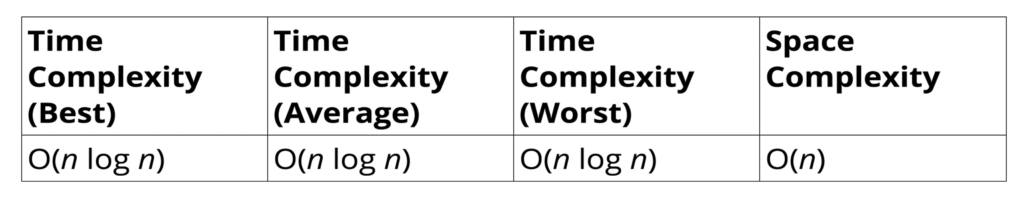
**let** right= **mergeSort**(arr**.slice**(mid));

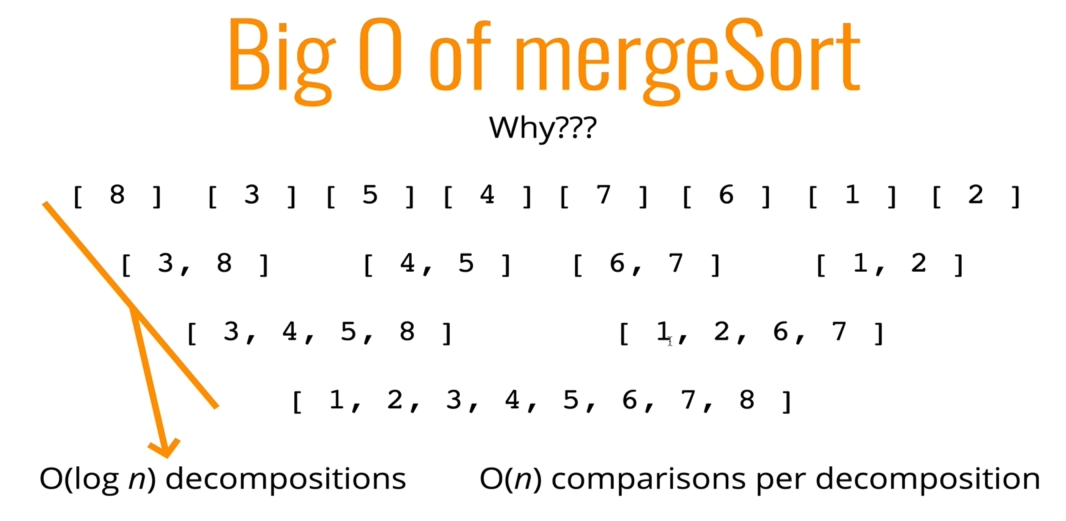
**return** **merge**(left, right);

}

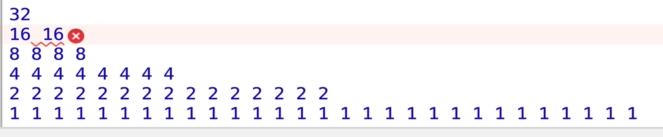
**mergeSort**([10,24,76,73])

**6.5.5 Big-O of MergeSort**

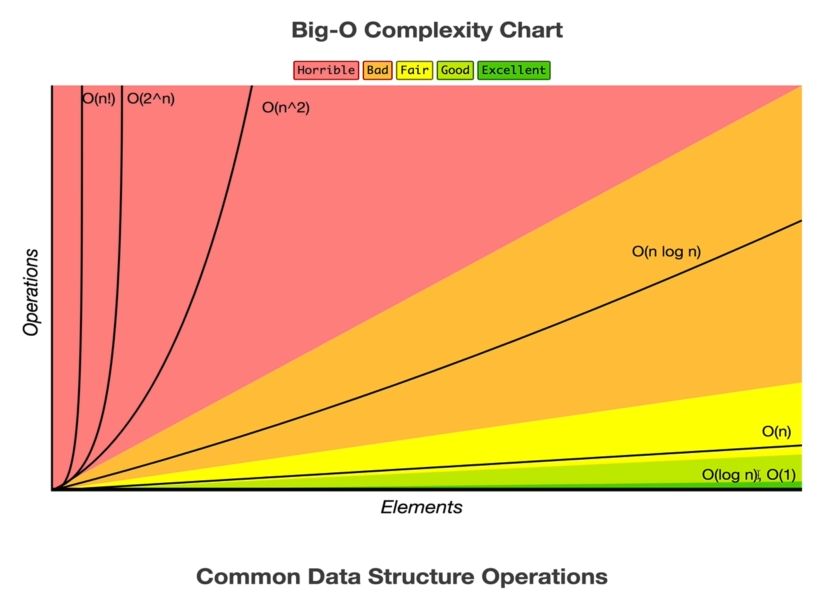




* Means that when the ***length*** of the array ***n*** grows, the ***rate of decomposition*** (split into single element) grows at the rate of **log(n)**
* That is when we have 8 elements, we have to split 3 times, 8 to 4, 4 to 2 and 2 to 1. 3 time split.
* But when when w3 have 32 elements, 32 to 16, 16 to 8, 8 to 4, 4 to 2 and 2 to 1. i.e. 5 times split.



* Note: But note that, in each decomposition, we have **O(n)** comparison. That’s why total Time-Complexity is **O(nlogn)**.



* Space Complexity: Compared to other previous sorting algorithm, Merge-sort takes more space, because we have to create new-arays and that take lot more space. But the good point is it is still **O(n)**.