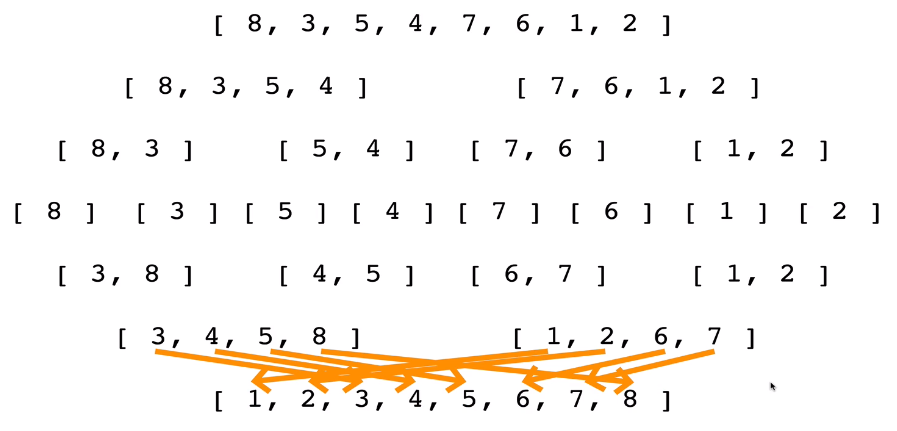
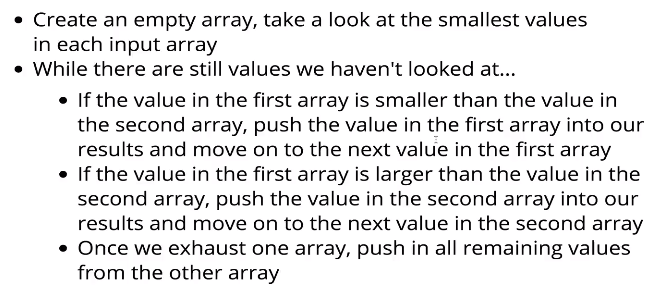
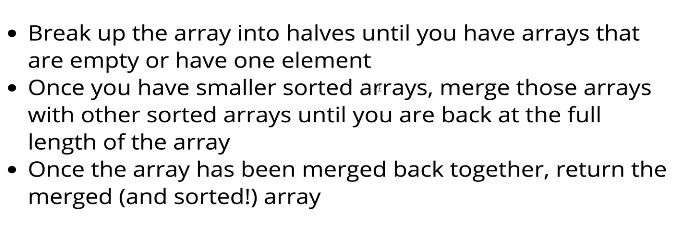
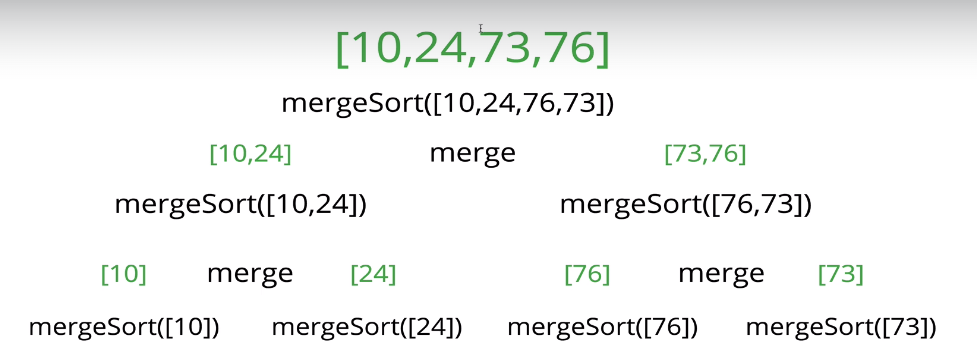
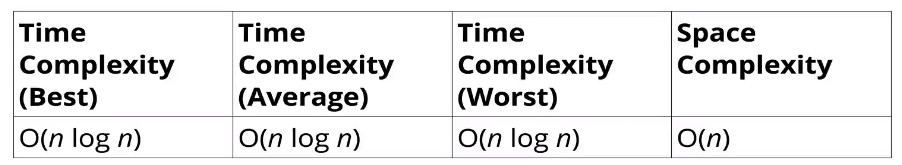
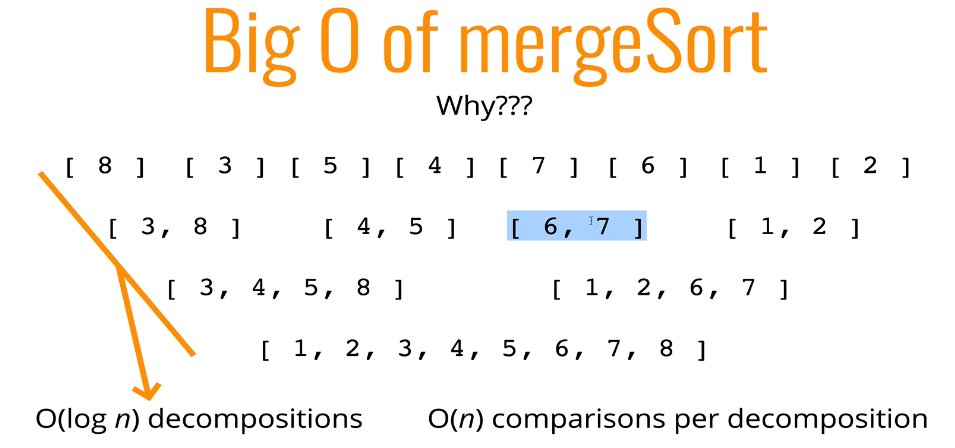
JavaScript Algorithms and Data Structures Masterclass

# Section 15-17: Intro to Intermediate Sorting: Merge Sort, Quick Sort, Radix Sort

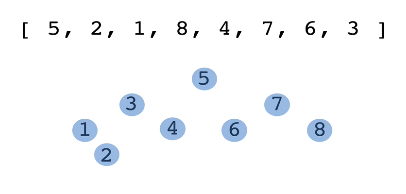
## Limitations of Elementary Sorts vs. Intermediate Sorts

* These new sorts are FASTER, but more COMPLEX
  + Improves time complexity from O(n2) to O(n log n)

## Section 15: Merge Sort

* A combination of merging and sorting
  + Exploits that 0 or 1 Element arrays are already sorted
  + Works by breaking an array into smaller 0/1 element arrays
    - Then building a new sorted array
  + Visualization:
    - 
    - Break apart the array into single element arrays, then re-merge them.
      * As you merge, you order the items
* **Logic**:
  + To implement merge sort, you need 1 function to merge 2 sorted arrays
    - This function should make a new sorted array that consists of the values of the 2 input arrays
    - Time = O(n + m)
    - Space = O(n + m)
      * This means that we are iterating over each element of the 2 arrays (n & m) once
      * And the parameters themselves are not modified (return new array!)
  + **Psuedo-code for MERGE()**:
    - 
  + **Psuedo-code for MERGE-SORT():**
    - 
      * Utilize recursion to break up the array into halves (.slice?) until each array is empty/only 1 element
      * **Recursion Diagram**:
        + 
* **Big O of Merge-Sort**:
  + 
  + 

## Section 16: Quick Sort

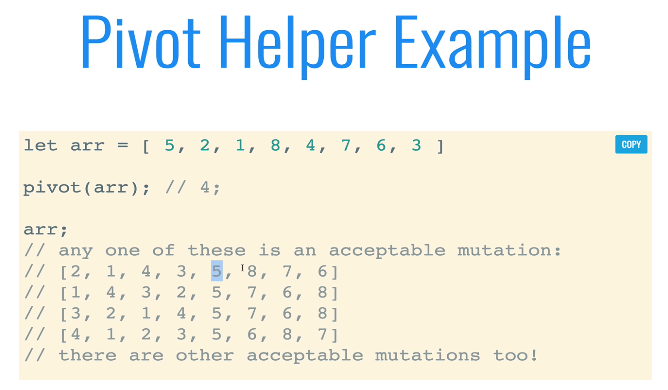
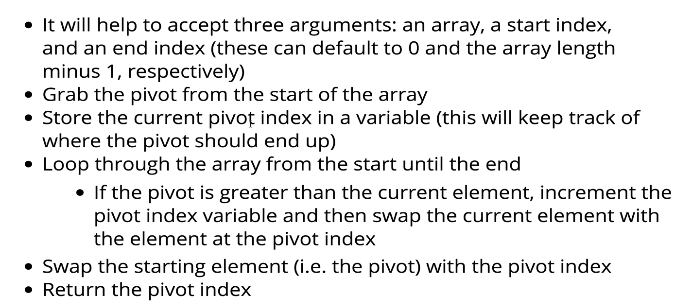
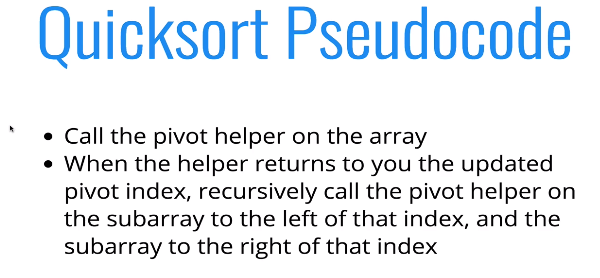
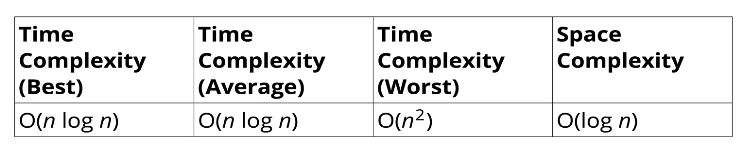
* Similar to Merge Sort (exploits that 1/0 element arrays are always sorted
* **Logic**:
  + Select 1 element (called ‘pivot’) and determine where this element’s index will be in the final sorted array
    - You then place elements (unsorted) above or below the pivot based on if it is bigger/smaller than pivot
    - Repeat this on left and right sides
* **Diagram**:
  + 

### Two Parts of Quick Sort:

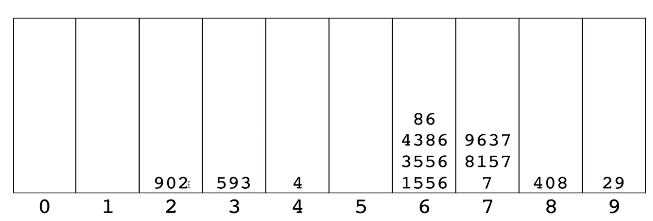
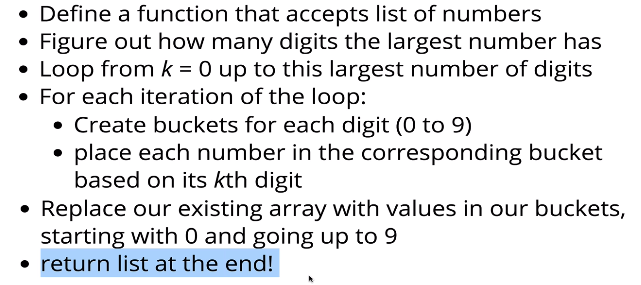
* + 1)
    - Pivot helper (function that arranges elements in an array on either side of a ‘pivot’ element
      * **Inputs**: array
      * **Output**: return the index of the pivot (don’t make a new array)
      * **Logic**:
        + Rearrange elements in the array so that:

All values < pivot == moved to left

All value > pivot == move to right

* + - * + Order on each side, doesn’t matter
    - Picking a Pivot
      * Normally should be the median value in array
      * But for simplicity, choose the pivot as the first element
        + 
    - **Psuedo-code for Pivot():**
      * 
    - **Psuedo-code for QuickSort()**
      * 
* **Quick Sort Big O**
  + 

## Section 17: Radix Sort

* It doesn’t make comparisons between numbers (INTEGER SORT)
  + It utilizes the ‘size’/digits of the number to determine sort
  + Compares the value of each digit of the numbers with each other
    - **Ex**:
      * 
    - Each bucket correlates to the value at that digit (ex. 1556, goes in bucket6 since the 4th digit = 6)
* **Radix Helpers**:
  + 1) getDigit(num, place)
    - This returns the digit in *num* at the given *place* 
      * **ex**. getDigit(12345, 0) => 5
  + 2) digitCount(num)
    - return num’s digits
  + 3) mostDigits(arr)
    - return the number of digits in the largest number in the array
* **Psuedo-code for Radix Sort**:
  + 
* **Radix Sort Big O**:
  + 