Chapter 6 : Part 1

**Sorting Algorithms: Bubble sort**

**6.1 We'll Learn**

1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Merge Sort
5. Quick Sort
6. Radix Sort

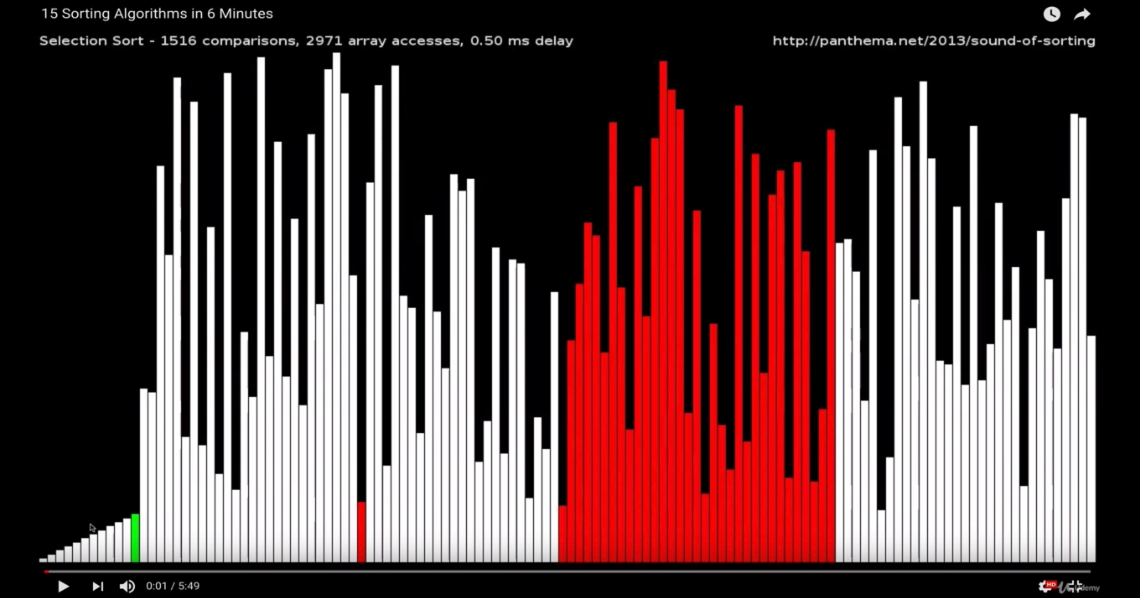
**6.2 Introduction to Sorting**

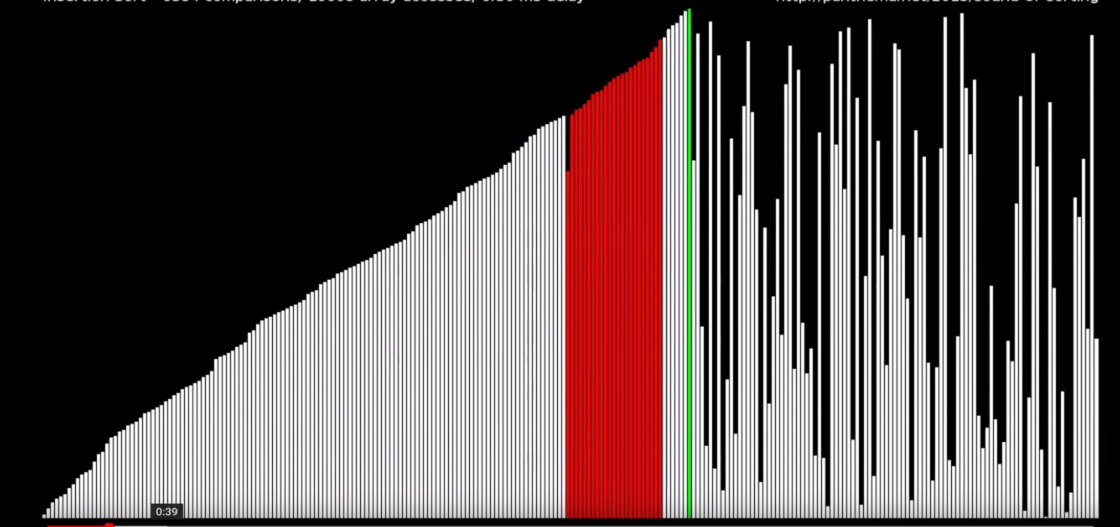
Sorting is most common in programming.

* We can sort numbers, strings. Eg: "*Sorting a number of array ascending order*" or "*Sorting names alphabetically*" or "*sorting individual* *character of a string*".
* Sorting is applied to different kind of Data-Structures, such as: *arrays*, *linked lists* or *trees*.
* JS has *built-in sorting* functions, some cases *built-in functions* are *fast* for *sorting*, in some cases you have to use your *own* *sorting* *function* to make it fast.
* 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.

|  |  |
| --- | --- |
| Examples   * Sorting numbers from smallest to largest * Sorting names alphabetically * Sorting movies based on release year * Sorting movies based on revenue |  |

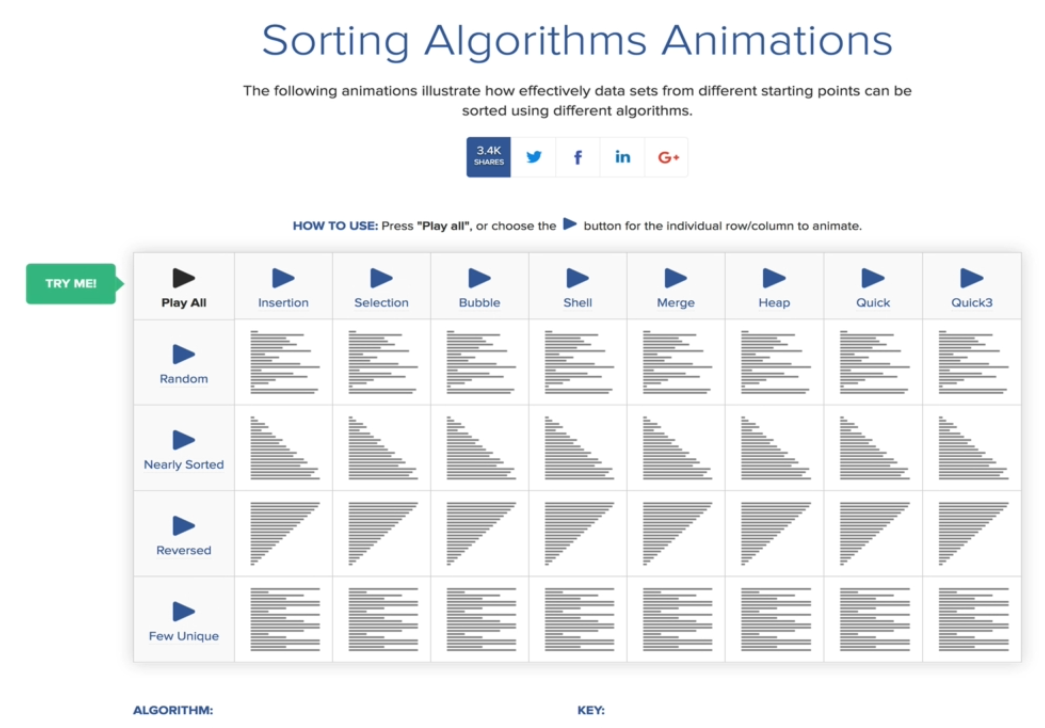
* Data-set Matters: There are various kind of sorting algorithms. Each of them are fast/slow for different kind of dataset.
* For example: A sorting algorithm can be fast for totally random data but can be slow if the data are in reverse order.







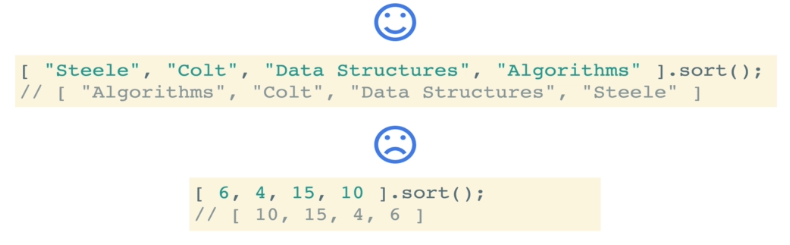
* Why do we need to learn this?
* Sorting is an incredibly common task, so it's good to know how it works
* There are many different ways to sort things, and different techniques have their own advantages and disadvantages
* Different sorting algorithms act differently on different dataset.



* Objectives:
* Implement *bubble sort*
* Implement *selection sort*
* Implement *insertion sort*
* Understand why it is important to learn these simpler sorting algorithms.

**6.3 Built-In JavaScript Sorting**

* Array.Sort(): JS has its built in sorting algorithm: **array.sort()**. But it doesn't always work the way you expect.
* It converts all elements to strings and then apply sorting on string, according to unicode.



* Solution: We have to define another function that tells the built in **array.sort()** the way the array needs to be sorted. This function is called a Comparator Function.
* Telling JavaScript how to sort
* The built-in ***sort*** method accepts an optional **comparator** function
* You can use this *comparator* function to tell *JavaScript* how you want it to *sort*
* The *comparator* looks at *pairs* of elements (***a*** and ***b***), determines their *sort order* based on the return value
* If it returns a ***negative*** number, a should come before **b**
* If it returns a ***positive*** number, a should come after **b**,
* If it returns ***0***, a and b are the same as far as the sort is concerned
* Example: JS ***sort()*** with comparator, that sort the numbers is ascending order.

// JS sort() with comparator

function **numberCompare**(num1, num2) {

  return num1 - num2;

}

[ 6, 4, 15, 10 ].**sort**(numberCompare)

* Example: Following sorts the strings according to their length.

// Sorting w.r. to length of the String

function **compareBylen**(str1, str2) {

  return str2.length - str1.length;

}

[ "Lassi", "Bolt", "Data Structures", "Algorithms" ] .**sort**(compareBylen);

|  |  |
| --- | --- |
| **6.4 BubbleSort**  A sorting algorithm where the **largest** values **bubble** **up** to the top!  Swap in JS: Many sorting algorithms involve some type of swapping functionality (e.g. swapping to numbers to put them in order) |  |

// swapping

// ES5

function **swap**(arr, idx1, idx2) {

   var temp = arr[idx1];

   arr[idx1] = arr[idx2];

   arr[idx2] = temp;

}

// ES2015

const **swap** = (arr, idx1, idx2) => {

[arr[idx1],arr[idx2]] = [arr[idx2],arr[idx1]];

}

* BubbleSort Pseudocode:
* Start **looping** from with a variable called **i** the **end** of the array towards the **beginning**
* Start an **inner loop** with a variable called **j** from the **beginning** until **i -1**
* If **arr[j]** is greater than **arr[j+1]**, swap those two values!
* Return the sorted array

**Practiced version**

Don’t forget to decrease **i**, **i--**

It is **i = arr.length** not **i = arr.length-1**, because **j<i-1**, we are not using **j <= i-1**

// Practiced Bubble-sort

function **bubbleSort\_prctc**(arr){

  for(var i = arr.length; i > 0; i--){

    for(var j = 0; j < i-1; j++){

      var temp = arr[j];

      if (arr[j] > arr[j+1]){

        arr[j]= arr[j+1];

        arr[j+1] = temp;

      }

    }

  }

  return arr;

}

**Instructor solution**

// UNOPTIMIZED VERSION OF BUBBLE SORT

function **bubbleSort**(arr){

  for(var i = arr.length; i > 0; i--){

    for(var j = 0; j < i - 1; j++){

      console.**log**(arr, arr[j], arr[j+1]);

      if(arr[j] > arr[j+1]){

        var temp = arr[j];

        arr[j] = arr[j+1];

        arr[j+1] = temp;

      }

    }

  }

  return arr;

}

// ES2015 Version

function **bubbleSort**(arr) {

  const **swap** = (arr, idx1, idx2) => {

    [arr[idx1], arr[idx2]] = [arr[idx2], arr[idx1]];

  };

  for (let i = arr.length; i > 0; i--) {

    for (let j = 0; j < i - 1; j++) {

      if (arr[j] > arr[j + 1]) {

**swap**(arr, j, j + 1);

      }

    }

  }

  return arr;

}

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

**Naive solution of Bubblesort**

// Naieve solution of Bubble-sort

function **bubbleSort\_naieve**(arr){

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

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

      var temp = arr[j];

      console.**log**(arr, arr[j], arr[j+1]);

      // swap

      if (arr[j] > arr[j+1]){

        arr[j]= arr[j+1];

        arr[j+1] = temp;

      }

    }

  }

  return arr;

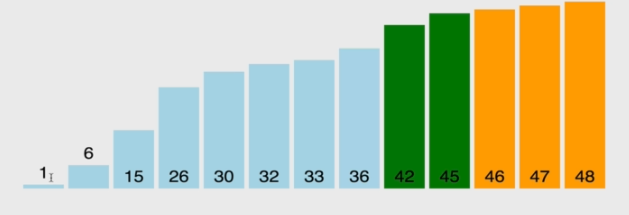
}

**bubbleSort\_naieve**([8,1,2,3,4,5,6,7]);

**6.5 BubbleSort Optimized**

Introducing ***no\_swap variable***.

In case of *nearly sorted data*, we can avoid un-necessary iterations on outer for loop. We can do that by using a *short-circuit break* if no swap is happened inside the inner loop.



|  |  |
| --- | --- |
| Optimized | Un-optimized |
| **bubbleSort([8,1,2,3,4,5,6,7]);**  VM40:7 (8) [8, 1, 2, 3, 4, 5, 6, 7] 8 1  VM40:7 (8) [1, 8, 2, 3, 4, 5, 6, 7] 8 2  VM40:7 (8) [1, 2, 8, 3, 4, 5, 6, 7] 8 3  VM40:7 (8) [1, 2, 3, 8, 4, 5, 6, 7] 8 4  VM40:7 (8) [1, 2, 3, 4, 8, 5, 6, 7] 8 5  VM40:7 (8) [1, 2, 3, 4, 5, 8, 6, 7] 8 6  VM40:7 (8) [1, 2, 3, 4, 5, 6, 8, 7] 8 7  VM40:16 One pass Complete  VM40:7 (8) [1, 2, 3, 4, 5, 6, 7, 8] 1 2  VM40:7 (8) [1, 2, 3, 4, 5, 6, 7, 8] 2 3  VM40:7 (8) [1, 2, 3, 4, 5, 6, 7, 8] 3 4  VM40:7 (8) [1, 2, 3, 4, 5, 6, 7, 8] 4 5  VM40:7 (8) [1, 2, 3, 4, 5, 6, 7, 8] 5 6  VM40:7 (8) [1, 2, 3, 4, 5, 6, 7, 8] 6 7  VM40:16 One pass Complete  (8) [1, 2, 3, 4, 5, 6, 7, 8] | **bubbleSort([8,1,2,3,4,5,6,7]);**  VM91:5 (8) [8, 1, 2, 3, 4, 5, 6, 7] 8 1  VM91:5 (8) [1, 8, 2, 3, 4, 5, 6, 7] 8 2  VM91:5 (8) [1, 2, 8, 3, 4, 5, 6, 7] 8 3  VM91:5 (8) [1, 2, 3, 8, 4, 5, 6, 7] 8 4  VM91:5 (8) [1, 2, 3, 4, 8, 5, 6, 7] 8 5  VM91:5 (8) [1, 2, 3, 4, 5, 8, 6, 7] 8 6  VM91:5 (8) [1, 2, 3, 4, 5, 6, 8, 7] 8 7  VM91:13 One pass Complete  VM91:5 (8) [1, 2, 3, 4, 5, 6, 7, 8] 1 2  VM91:5 (8) [1, 2, 3, 4, 5, 6, 7, 8] 2 3  VM91:5 (8) [1, 2, 3, 4, 5, 6, 7, 8] 3 4  VM91:5 (8) [1, 2, 3, 4, 5, 6, 7, 8] 4 5  VM91:5 (8) [1, 2, 3, 4, 5, 6, 7, 8] 5 6  VM91:5 (8) [1, 2, 3, 4, 5, 6, 7, 8] 6 7  VM91:13 One pass Complete  VM91:5 (8) [1, 2, 3, 4, 5, 6, 7, 8] 1 2  VM91:5 (8) [1, 2, 3, 4, 5, 6, 7, 8] 2 3  VM91:5 (8) [1, 2, 3, 4, 5, 6, 7, 8] 3 4  VM91:5 (8) [1, 2, 3, 4, 5, 6, 7, 8] 4 5  VM91:5 (8) [1, 2, 3, 4, 5, 6, 7, 8] 5 6  VM91:13 One pass Complete  VM91:5 (8) [1, 2, 3, 4, 5, 6, 7, 8] 1 2  VM91:5 (8) [1, 2, 3, 4, 5, 6, 7, 8] 2 3  VM91:5 (8) [1, 2, 3, 4, 5, 6, 7, 8] 3 4  VM91:5 (8) [1, 2, 3, 4, 5, 6, 7, 8] 4 5  VM91:13 One pass Complete  VM91:5 (8) [1, 2, 3, 4, 5, 6, 7, 8] 1 2  VM91:5 (8) [1, 2, 3, 4, 5, 6, 7, 8] 2 3  VM91:5 (8) [1, 2, 3, 4, 5, 6, 7, 8] 3 4  VM91:13 One pass Complete  VM91:5 (8) [1, 2, 3, 4, 5, 6, 7, 8] 1 2  VM91:5 (8) [1, 2, 3, 4, 5, 6, 7, 8] 2 3  VM91:13 One pass Complete  VM91:5 (8) [1, 2, 3, 4, 5, 6, 7, 8] 1 2  2VM91:13 One pass Complete  (8) [1, 2, 3, 4, 5, 6, 7, 8] |

***Practiced version***

// *OPTIMIZED VERSION OF BUBBLE SORT*

**function** **bubbleSort**(arr){

**var no\_Swap**;

**for**(**var** i=arr**.**length; i **>** 0; i--){

**no\_Swap** = **true**;

**for**(**var** j=0; j **<** i - 1; j++){

      console**.log**(arr, arr[j], arr[j+1]);

**if**(arr[j] **>** arr[j+1]){

        // *Swap !*

**var** temp=arr[j];

        arr[j] = arr[j+1];

        arr[j+1] = temp;

**no\_Swap** = **false**;

      }

    }

    console**.log**("One pass Complete");

    // *If no swap happens inside nested-loop, outer loop breaks.*

**if**(**no\_Swap**) **break**;

  }

**return** arr;

}

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

***Instructor version***

// *Optimized BubbleSort with noSwaps*

**function** **bubbleSort**(arr){

**var** noSwaps;

**for**(**var** i=arr**.**length; i **>** 0; i--){

    noSwaps = **true**;

**for**(**var** j=0; j **<** i - 1; j++){

**if**(arr[j] **>** arr[j+1]){

**var** temp=arr[j];

        arr[j] = arr[j+1];

        arr[j+1] = temp;

        noSwaps = **false**;

      }

    }

**if**(noSwaps) **break**;

  }

**return** arr;

}

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

* Time complexity:

1. Non-linear: In general worst case (random, *non-sorted* data), the *Time Complexity* is **O(n2)**. Non-linear.
2. Linear: With optimized bubble-sort for *nearly sorted* data, the *Time Complexity* is **O(n)**. Linear.