Bars Project

Sorting algorithms

For each sorting algorithm discussed below, there is a step-by-step explanation of how the algorithm works, image representation, and implementation of the algorithm using JavaScript.

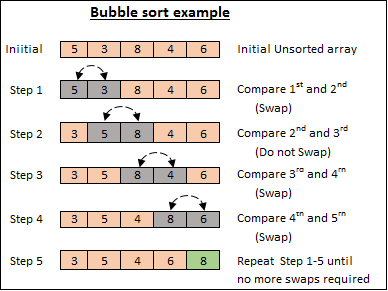
Bubble sort

Bubble sort follows the recursion technique.

Step-by-step guide:

* Start by comparing the first two elements in an array.
* Swap them if required.
* Continue till the end of the array. At this point, you have made a series of inner passes and completed an outer pass.
* Repeat the process until the entire array is sorted.

Image representation



[Image Source](https://medium.com/karuna-sehgal/an-introduction-to-bubble-sort-d85273acfcd8)

JavaScript implementation

function bubbleSort(arr){

//Outer pass

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

//Inner pass

for(let j = 0; j < arr.length - i - 1; j++){

//Value comparison using ascending order

if(arr[j + 1] < arr[j]){

//Swapping

[arr[j + 1],arr[j]] = [arr[j],arr[j + 1]]

}

}

};

return arr;

};

console.log(bubbleSort([5,3,8,4,6]));

Output:

[ 3, 4, 5, 6, 8 ]

Bubble sort has the following performance cases:

* Worst-case time complexity: Big O (n^2).
* Average-case time complexity: Big theta (n^2).
* Best-case time complexity: Big omega (n).
* Space complexity: Big O (1).

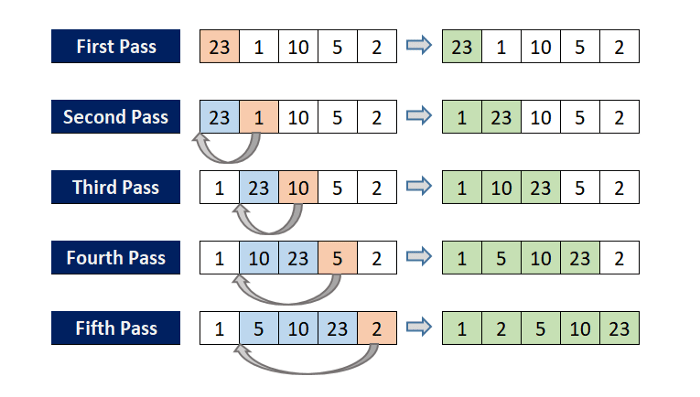
Insertion sort

Insertion sort uses the recursion technique. There is a portion of the array that is sorted and the other that is unsorted. So you have to compare the elements from the unsorted portion one by one and insert them into the sorted portion in the correct order. In the guide below we are using ascending order.

Step-by-step guide

* Start by comparing the second element of the array with the first element assuming the first element is the sorted portion. Swap if the second element is smaller than the first element.
* Iterate through comparing the first element with each element of the unsorted portion. If the element from the unsorted portion is smaller than the first element, swap.
* After iterating through the entire array, increment the sorted portion to the next index and recursively compare the value of the last index of the sorted portion with each value of the unsorted portion. Swap where the value of the unsorted portion is smaller.
* The sorted portion shall increase until it covers the entire array yielding a sorted array.

Image representation



[Image Source](https://medium.com/@rajat_m/implement-5-sorting-algorithms-using-javascript-63c5a917e811)

JavaScript Implementation

function insertionSort(arr){

//Start from the second element.

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

//Go through the elements behind it.

for(let j = i - 1; j > -1; j--){

//value comparison using ascending order.

if(arr[j + 1] < arr[j]){

//swap

[arr[j+1],arr[j]] = [arr[j],arr[j + 1]];

}

}

};

return arr;

}

console.log(insertionSort([23, 1, 10, 5, 2]));

Output:

[ 1, 2, 5, 10, 23 ]

Insertion sort has the following performance cases:

* Worst-case time complexity: Big O(n^2).
* Average-case time complexity: Big theta (n^2).
* Best-case time complexity: Big omega (n).
* Space complexity: Big O (1).

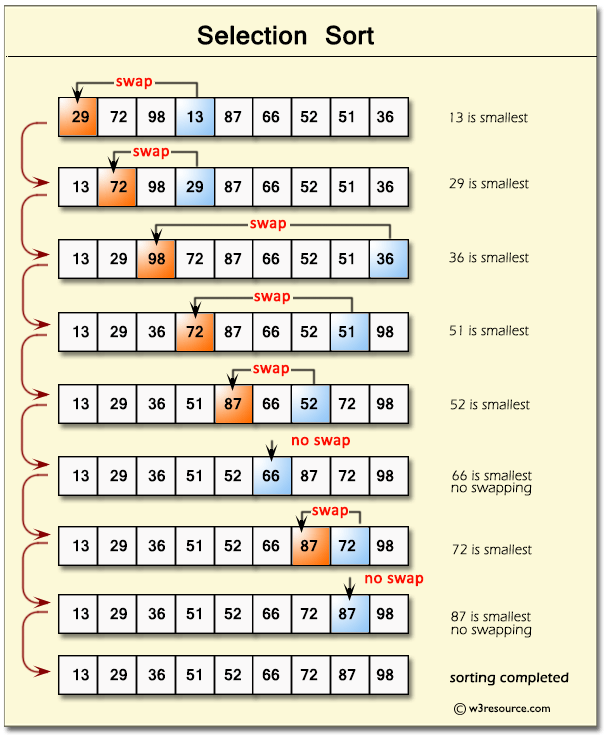
Selection sort

Selection sort uses the recursion technique. In the guide below, we are using ascending order. For descending order, you do the reverse.

Step-by-step guide

* Given an array, assume that the first element in the array is the smallest.
* From the other portion of the array, find the minimum value, and swap it with the first element. At this point, you have completed the first pass.
* Repeat the same procedure with the rest of the array comparing the elements to the right, not the left.

Image representation



[Image Source](https://medium.com/@rajat_m/implement-5-sorting-algorithms-using-javascript-63c5a917e811)

JavaScript Implementation

function selectionSort(arr) {

let min;

//start passes.

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

//index of the smallest element to be the ith element.

min = i;

//Check through the rest of the array for a lesser element

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

if (arr[j] < arr[min]) {

min = j;

}

}

//compare the indexes

if (min !== i) {

//swap

[arr[i], arr[min]] = [arr[min], arr[i]];

}

}

return arr;

}

console.log(selectionSort([29, 72, 98, 13, 87, 66, 52, 51, 36]));

Output

[ 13, 29, 36, 51, 52, 66, 72, 87, 98 ]

Selection sort has the following performance cases:

* Worst-case time complexity: Big O (n^2).
* Average-case time complexity: Big theta (n^2).
* Best-case time complexity: Big omega (n).
* Space complexity: Big O(1).

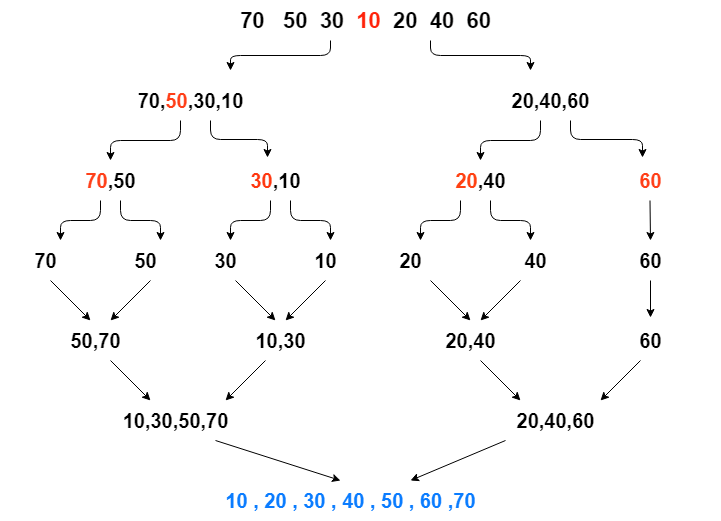
Merge sort

Merge sort uses the divide and conquer technique. The main concept of merge sort is that an array of length 1 is sorted. The task, therefore, lies in splitting the array into subarrays of size 1 and then merge them appropriately so that it comes up with the sorted array.

Step-by-step guide

* Split the array elements into individual elements.
* Compare the individual elements and arrange them in order. This results in subarrays of length 1 or 2.
* Make an empty array.
* Compare the elements of the subarrays and push the smaller of the values to the empty array.
* If you had extracted all the values from one of the arrays, push the remaining array to the new array.
* Continue until all subarrays have been covered and you have one sorted array.

Image representation



[Image Source](https://medium.com/@rajat_m/implement-5-sorting-algorithms-using-javascript-63c5a917e811)

JavaScript implementation

//merging two arrays appropriately.

function merge(arr1, arr2) {

//make a new array and have two value pointers

let res = [],

i = 0,

j = 0;

//sorting the first array.

if (arr1.length > 1) {

let min = 0;

for (let i = 0; i < arr1.length; i++) {

if (i !== min) {

if (arr1[i] < arr1[min]) {

//also swap the elements

[arr1[i], arr1[min]] = [arr1[min], arr1[i]];

//change the minimum

min = i;

}

}

}

}

//sorting the second array.

if (arr2.length > 1) {

let min = 0;

for (let i = 0; i < arr2.length; i++) {

if (i !== min) {

if (arr2[i] < arr2[min]) {

//also swap the elements

[arr2[i], arr2[min]] = [arr2[min], arr2[i]];

//change the minimum

min = i;

}

}

}

}

//Value comparison.

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

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

res.push(arr1[i]);

i++;

} else {

res.push(arr2[j]);

j++;

}

}

//pushing the rest of arr1.

while (i < arr1.length) {

res.push(arr1[i]);

i++;

}

//pushing the rest of arr2.

while (j < arr2.length) {

res.push(arr2[j]);

j++;

}

return res;

}

//merge sort

function mergeSort(arr) {

//Best case

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

//splitting into halves

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

let arr1 = arr.slice(0, mid);

let arr2 = arr.slice(mid);

let arr1\_subarrays = [],

sorted\_arr1\_subarrays = [];

let arr2\_subarrays = [],

sorted\_arr2\_subarrays = [];

//loop through array 1 making subarrays of two elements

for (let i = 0; i < arr1.length; i += 2) {

arr1\_subarrays.push(arr1.slice(i, i + 2));

}

//loop through array 2 making subarrays of two elements.

for (let i = 0; i < arr2.length; i += 2) {

arr2\_subarrays.push(arr2.slice(i, i + 2));

}

// sorting each subarray of arr1.

for (let i = 0; i < arr1\_subarrays.length; i += 2) {

let result = merge(arr1\_subarrays[i], arr1\_subarrays[i + 1]);

result.forEach((value) => sorted\_arr1\_subarrays.push(value));

}

// sorting each subarray of arr2.

for (let i = 0; i < arr2\_subarrays.length; i += 2) {

let result = merge(arr2\_subarrays[i], arr2\_subarrays[i + 1]);

result.forEach((value) => sorted\_arr2\_subarrays.push(value));

}

let result = merge(sorted\_arr1\_subarrays, sorted\_arr2\_subarrays);

return result;

}

console.log(mergeSort([70, 50, 30, 10, 20, 40, 60]));

Output:

[ 10, 20, 30, 40, 50, 60, 70 ]

Merge sort has the following performance cases:

* Worst-case time complexity: Big O (n \* log n).
* Average-case time complexity: Big theta (n \* log n).
* Best-case time complexity: Big omega (n \* log n).
* Space complexity: Big O (n).

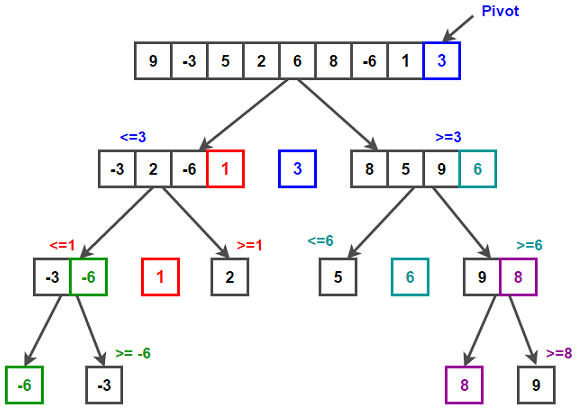
Quicksort

Quicksort applies the divide and conquer technique as well. It works by having a pivot element such that the elements to the left of it are less than it and those to the right are greater than it. The pivot element can be any element in the array.

Step-by-step guide

* Select a pivot element.
* Split the array into two arrays with those less than the pivot element on the left and those greater than the pivot element to the right.
* Carry out the above steps recursively until we have subarrays of length 1. Combine the subarrays to yield a sorted array.

Image representation



[Image Source](https://www.techiedelight.com/quicksort/)

JavaScript implementation

function partition(items, left, right) {

//rem that left and right are pointers.

let pivot = items[Math.floor((right + left) / 2)],

i = left, //left pointer

j = right; //right pointer

while (i <= j) {

//increment left pointer if the value is less than the pivot

while (items[i] < pivot) {

i++;

}

//decrement right pointer if the value is more than the pivot

while (items[j] > pivot) {

j--;

}

//else we swap.

if (i <= j) {

[items[i], items[j]] = [items[j], items[i]];

i++;

j--;

}

}

//return the left pointer

return i;

}

function quickSort(items, left, right) {

let index;

if (items.length > 1) {

index = partition(items, left, right); //get the left pointer returned

if (left < index - 1) {

//more elements on the left side

quickSort(items, left, index - 1);

}

if (index < right) {

//more elements on the right side

quickSort(items, index, right);

}

}

return items; //return the sorted array

}

let items = [5, 3, 7, 6, 2, 9];

console.log(quickSort(items, 0, items.length - 1));

Output:

[ 2, 3, 5, 6, 7, 9 ]

The following steps are followed while implementing quicksort:

* Start with the left pointer pointing at index 0 and the right pointer pointing at index 0.
* Compare the element at the left pointer with the pivot element. If it is less than, increment the left pointer. Recursively do this until the value at a pointer is not less than the pivot. At this point, stop the iteration.
* Compare the element at the right pointer with the pivot element. If it is greater than, decrement the right pointer. Recursively do this until the value at a pointer is not greater than the pivot. At this point, stop the iteration.
* When you have stopped the iteration of the left and the right pointer, swap the values being pointed by the right and left pointer.
* Increment the right and the left pointer. At this point, a further increment shall cause the left pointer to be greater than the right pointer which shall break the loop returning the left pointer.
* Recursively do the steps above until the entire array is sorted.

Quicksort has the following performance cases:

* Worst-case time complexity: Big O (n^2).
* Average-case time complexity: Big theta (n \* log n).
* Best-case time complexity: Big omega (n \* log n).
* Space complexity: Big O (n \* log n).