**IMPLEMENTATION AND COMPARISON OF SORTING ALGORITHMS**

**1) BUBBLE SORT:**

Bubble sort starts from the left element and compares the adjacent elements and keeps “BUBBLING” the larger element to the right till the end element is the largest one (right end), then continues for the N-1, N-2 and so on elements till the whole list is sorted.

Best case complexity: O(n)

Worst case complexity :O(N2)

function bubbleSort(arr){

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

let noSwaps = true

for(let j=0 ;j<i;j++){

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

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

noSwaps = false

}

}

if(noSwaps) break;

}

return arr

}

**2) SELECTION SORT:**

Selection sort selects the smallest element and swaps it with position of the first item and carries on for the next N-1,N-1 and so on elements till the entire list is sorted.

Best and worst case complexity: O(N2)

function selectionSort(arr) {

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

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

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

let lowest = i;

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

if (arr[lowest] > arr[j]) {

lowest = j;

}

}

if (i !== lowest) swap(arr, i, lowest);

}

return arr;

}

**3) INSERTION SORT:**

Insertion sort compares the current element with the adjacent one. The position of the element is found and it is inserted there and this goes on for the rest of the elements till the list is sorted.

Best case complexity: O(n)

Worst case complexity :O(N2)

function insertionSort(arr){

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

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

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

let temp = arr[j]

arr[j]= arr[j-1]

arr[j-1]=temp

}else{

break

}

}

}

return arr

}

**4) MERGE SORT:**

Merge sort works on the divide and conquer approach. It first keeps on dividing the list into two halves and then the halves created into other two halves, till there is just one element in the list and then joins the arrays in sorted manner till it contains all sorted elements merged into the original length of the array.

Best case complexity: O(N log N)

Average case complexity: O(N log N)

Worst case complexity: O(Nlog N)

function merge(arr1, arr2) {

let newArr = []

let j = 0;

let i = 0;

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

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

newArr.push(arr1[i])

i++

} else {

newArr.push(arr2[j])

j++

}

}

while (i < arr1.length) {

newArr.push(arr1[i])

i++

}

while (j < arr2.length) {

newArr.push(arr2[j])

j++

}

return newArr

}

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)

}

**5) QUICK SORT:**

Quick sort also works on divide and conquer approach. It selects a pivot element from the array and partitions the arrays into two sub-arrays and continues to divide the sub-arrays based on the found pivot till they are the smallest possible and then sorts them by continuing to combine them and moving back up till the entire list is sorted.

Best case complexity: O(N log N)

Average case complexity: O(N log N)

Worst case complexity: O(N2)

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

function swap(array, i, j) {

let temp = array[i];

array[i] = array[j];

array[j] = temp

}

let pivot = arr[start]

let swapidx = start;

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

if (pivot > arr[i]) {

swapidx++

swap(arr, swapidx, i)

}

}

swap(arr, start, swapidx)

return swapidx

}

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

if (left < right) {

let pivotIndex = pivot(arr, left, right)//index of 4 ie 3

//left

quickSort(arr, left, pivotIndex - 1)

//right

quickSort(arr, pivotIndex + 1, right)

}

return arr

}

**6) Heap Sort:**

Heap sort first converts the list into a max heap using heapify function. Then it repeatedly swaps the first value(max) of the heap to the last and then the second largest and so on till the list is in sorted order.

Best case complexity: O(N log N)

Average case complexity: O(N log N)

Worst case complexity: O(Nlog N)

function heapSort(arr) {

for (let i = Math.floor(arr.length / 2) - 1; i >= 0; i--){

heapify(arr, arr.length, i);

}

for (let i = arr.length - 1; i > 0; i--) {

let temp = arr[0];

arr[0] = arr[i];

arr[i] = temp;

heapify(arr, i, 0);

}

return arr

}

function heapify(arr, n, i){

let left = 2 \* i + 1;

let right = 2 \* i + 2;

let largest = i;

if (arr[left] > arr[largest] && left < n) largest = left;

if (arr[right] > arr[largest] && right < n) largest = right;

if (largest != i) {

var swap = arr[i];

arr[i] = arr[largest];

arr[largest] = swap;

heapify(arr, n, largest);

}

}

**7) 3 MEDIAN QUICK SORT:**

In 3 median quick sort 3 elements that is left, middle and right elements of the array are chosen as pivots. It is done to prevent the worst case of quick sort( O(n2) ). These 3 elements are then sorted and the middle one is swapped with arr[right-1] and then the partitioning algorithm is applied to arr[i+1],….,arr[right-2].

Best , average and worst case complexity: O(N log N)

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

let swaps = 0;

let comparisons = 0;

var len = arr.length - 1;

mQuickSort(arr, 0, len);

function medianPivot(arr,lowest,highest) {

let mid = (highest) / 2;

let pointersArr = [arr[lowest], arr[mid], arr[highest]];

pointersArr.sort();

let midVal = pointersArr[1];

let temp = arr[highest];

arr[highest] = midVal;

if (midVal == arr[lowest]) {

arr[lowest] = temp;

} else if (midVal == arr[mid]) {

arr[mid] = temp;

}

return partition(arr, lowest, highest);

}

function mQuickSort( arr, lowest, highest) {

if (lowest >= highest)

return;

if (lowest < highest) {

var piv = medianPivot(arr, lowest, highest);

QuickSort(arr, lowest, highest);

}

}

function QuickSort(arr, lowest, highest) {

if (lowest < highest) {

var piv = partition(arr, lowest, highest);

QuickSort(arr, lowest, piv - 1);

QuickSort(arr, piv + 1, highest);

}

}

function partition(arr, lowest, highest) {

let pivot = arr[highest];

let i = (lowest - 1);

for (let j = lowest; j < highest; j++) {

if (arr[j] <= pivot) {

i++;

[arr[i], arr[j]] = [arr[j], arr[i]]

swaps++;

}

comparisons++;

}

[arr[i+1], arr[highest]] = [arr[highest], arr[i+1]]

swaps++;

return i + 1;

}

console.log(comparisons, swaps)

return arr

}

**GRAPHIC USER INTERFACE**

**1) Comparison of outputs and graphical representation(from GUI) based on input size of random order**

1. Input size: 100

Chart

Description automatically generated with medium confidence

1. Input size: 1,000

Chart

Description automatically generated

1. Input Size: 10,000

Chart, bar chart

Description automatically generated

1. Input Size: 50,000

Chart, bar chart

Description automatically generated

**2) Input size vs time comparison and improvement for running time**

As the input size increases algorithms like Bubble, selection, insertion become useless as they take much more time (O(N2)) than the efficient ones like merge heap and 3 median quick sort approach (O(N log N)). Even Quick sort appears to be useless as the input size gets too large. So, as a replacement 3 median quick sort is used so as to fix the problem of worst case time complexity of quick sort.

Merge sort, heap sort, and 3 median quick sort appear to be the fastest ones as the input size grows so they are more reliable as the input size grows as they have average time complexity of O(N log N).

When the input size gets really large then Radix sort can be used whose time complexity is O(k\*n) k being the number of digits in the largest number. It depends on how many iterations it has to go through due to the number of digits. It is better for more randomized data.

**3) GUI Functions and files:**

The **index.html** file contains the html code for the graph and various buttons and random generators for the input and the referenced to the functions they call in order for the sorting methods to work.

The **index.css** file is used to enhance the physical appearance of the html like aligning elements and for the graphical representation of the output and hence make the web page more presentable.

**Functions.js** contains the basic functions that control the functionality of the User Interface and hence binds it to the html features like various buttons like random number generator(generateRandom), range generator(setRange), sorting algorithm comparison function(sort and compareAll) which runs all the sorting algos at once and hence compares them and graphically represents them.

**sortMethods.js** contains all the sorting methods used to sort the input arrays like bubble sort, insertion sort, merge sort, quick sort, heap sort and 3 median quick sort.

**4) Technologies Used:**

1. HTML
2. CSS
3. Javascript

**5) Files:**

1. index.html
2. index.css
3. functions.js
4. sortingMethods.js
5. README.md