## Buble Sort Algorithm

Bubble sort algorithms is one of the most basic sorting algorithms and maybe the least efficient, it has 2 loops the first loop iterations == n and the number of the iterations of the second loop == (n - i ) where is the index of the iteration in the first loop, for example if the size of the array is 10 , then the first loop does 10 iterations and the second nested loop does n -9 , n-8 , n - 7 , n -6 , n-5 , n-4 , n-3, n-2 and n-1 iterations. In case of the ascending , we start by comparing the first element to the second element and if the second element is smaller we swap elements and then we compare second element to third element then third element to fourth element. This guarantees that the largest element will be at the bottom, and the second iteration the second larget item will be in the second element to the bottom. there is an enhancement to the algorithm called the cocktail party, where we once we reach the end of the array , we do a backward pass ( in addition to the foward ) pass to put the smallest item in the top . Also the algorithm check wether the array is sorted and skips if it is sorted see below the pseudo code a la rough garden style

### The Algorithm

**input :** An array of n numbers , in arbitrary order

**output:** An array of the same numbers sorted from smallest to larget

**assumptions :** None

is\_swapped := true, start = 0, end = number\_of\_elements - 1 ,

**while** is\_swapped **do**

**for** forward\_element := start to end **do**

**if** array(forward\_element) > array(forward\_element + 1)

swap forward\_element with foward\_element + 1

is\_swapped := True

**if** is\_swapped := False

reduce end by 1

**for** backward\_element := end -1 to start -1 **do**

**if** array(forward\_element) > array(forward\_element + 1)

swap forward\_element with foward\_element + 1

is\_swapped := True

increase start by 1

### The example

sort 7,9,5,3,4,8,1

**first iteration**

array is 7,9,5,3,4,8,1

the first index of the array is 0 and the last index is 6

in the forward pass the 9 is pushed to the last index of the array, the array becomes 7,5,3,4,8,1,9

the first index of the array is 0 and the last index is 5

in the backward pass the 1 is pushed to the first index of the array the array becomes 1,7,5,3,4,8,9

**Second Iteration**

array is *1*,7,5,3,4,8,*9 , italic element means sorted*

the first index of the array is 1 and the last index is 5

in the forward pass the 7 is pushed to the last index of the array, the array becomes 1,5,3,4,7,8,9

the first index of the array is 1 and the last index is 4

in the backward pass the 3 is pushed to the first index of the array, the array becomes 1,3,5,4,7,8,9

**Third Iteration**

array is *1,3*,5,4,*7,8,9 , italic element means sorted*

the first index of the array is 2 and the last index is 4

in the forward pass the 5 is pushed to the last index of the array, the array becomes 1,3,4,5,7,8,9

the first index of the array is 2 and the last index is 3

nothing to do here

**fourth Iteration**

array is *1,3,4,5,7,8,9 , italic element means sorted*

the first index of the array is 3 and the last index is 3

nothing to do here

swap is set to false

nothing to do here

**We don't enter the fifth Iteration.**

# Insertion Sort

## The Example

Assume you have the array 7,9,5,3,4,8,1,9

Before the iteration 7 is stored in the sorted array,

Which will be 7

and 9 is grabbed from the unsorted array and saved in a temporary memory location , we compared 9 to 7 and since 9 is bigger than 7, we append 9 to the sorted array

**Second iteration 1st Loop**

Sorted array is 7,9 , we grab 5 from the unsorted array

We compare 9 to 5 and since 5 is smaller than 9 we enter first iteration of the second loop

**First Iteration 2nd Loop**

We store 5 in the place of 9 and 9 in the place of 5, the sorted array will be

7,5,9

We test the entry condition of the 2nd loop and we find that 5 is smaller than 7, then we enter

**Second Iteration 2nd Loop**

We swap 5 and 7, the sorted array is 5,7,9

We test the entry condition of the 2nd loop and we find that the index of 5 is out of bound , we

Exit the 2nd loop

**Third Iteration 1st loop**

Sorted Array is 5,7,9 , we grab the 3 from the unsorted array , we compare the 3 to the 9 and we find that 3 is smaller than 9 , we enter the first iteration of the second loop

**First Iteration 2nd Loop**

We store 3 in the place of 9 and 9 in the place of 3, the sorted array will be

5,7,3,9

We test the entry condition of the 2nd loop and we find that 3 is smaller than 7, then we enter

**Second Iteration 2nd Loop**

We swap 3 and 7, the sorted array is 5,3,7,9

We test the entry condition of the 2nd loop and we find that 3 is smaller than five then we enter

**Third Iteration of 2nd loop**

We swap 3 and 5 , the sorted array is 3,5,7,9, after finishing the loop we test for re-entry but we

Fail because out of bound index

**fourth Iteration 1st loop**

Sorted Array is 3,5,7,9 , we grab the 8 from the unsorted array , we compare the 8 to the 9 and we find that 8 is smaller than 9 , we enter the first iteration of the second loop

**First Iteration 2nd Loop**

We store 8 in the place of 9 and 9 in the place of 8, the sorted array will be

3,5,7,8,9

We test the entry condition of the 2nd loop and we find that 8 is larger than 7, then we exist the

2nd loop

**fifth Iteration 1st loop**

Sorted Array is 3,5,7,8,9 , we grab the 1 from the unsorted array , we compare the 1 to the 9 and we find that 1 is smaller than 9 , we enter the first iteration of the second loop

**First Iteration 2nd Loop**

We store 1 in the place of 9 and 9 in the place of 1, the sorted array will be

3,5,7,8,1,9

We test the entry condition of the 2nd loop and we find that 1 is smaller than 8, then we enter the 2nd loop again

**Second Iteration 2nd Loop**

We swap 1 and 8 array is

3,5,7,1,8,9

We test the entry condition of the 2nd loop and we find that 1 is smaller than 7, then we enter the 2nd loop again

**Third Iteration 2nd Loop**

We swap 1 and 7 array is

3,5,1,7,8,9

We test the entry condition of the 2nd loop and we find that 1 is smaller than 5, then we enter the 2nd loop again

**Third Iteration 2nd Loop**

We swap 1 and 5 array is

3,1,5,7,8,8

We test the entry condition of the 2nd loop and we find that 1 is smaller than 3, then we enter the 2nd loop again

**Fourth Iteration 2nd Loop**

We swap 1 and 3 array is

1,3,5,7,8,9

We test the entry condition of the 2nd loop, and we find that 1 is index out of bound , we exit the loop

We exit the 1st loop because we reached the full size of the unsorted array

## The Description

One of the simplest sorting algorithms and is used to sort arrays with a small number of elements. For simplicity and visualization we will consider an **unsorted array (UA)** static ( fixed size ) and a dynamically growing **sorted array** (SA) that has size zero at initialization and at the end of the algorithm it will have similar size to the **unsorted\_array** .

Before the first iteration We put the first element (index 0) of unsorted array into the sorted array. Sorted array size is one, We started at the second element of unsorted array, the sorted array size is 2 -1 which his one. (Initialization of loop invariant).

**At the first iteration**

we compare the first SA and second element of the UA, if the element of the SA is larger than UA then we put the element of UA to the left of the SA, if not we add the UA to the right, in either cases SA grows to 2 .

**At the second iteration**

We compare the third element of UA to the element of SA, if the last element of SA is larger than the current element of UA , we compare the current element of UA to the element before that, which is the first element , if the first element of SA is larger, we swap , if not we place to the left of first element of SA and to the right of the second element, if the last element of SA is smaller that means the third element of UA goes to the right of the array . in either case SA grows to 3. (Maintenance of loop invariant) , the SA is

**At the third iterations and other iterations**

We get the fourth element of the UA and we compare it to the last element of the SA if it is larger than the last element of the SA , we place it as the last element of the SA and we grow the SA to four and we move to the fifth element of UA. If it is smaller than the last element of the SA , we find it is place in the sorted array.

**Exit**

We compare the last element of UA to the last element of the SA, if the last elment of UA is larger than the last element of the SA , we append to the SA and we exit and if it s not we place it I the SA. (Termination )

At all stages of this sorting algorithm, the SA array is the sorted version of a the i-1 subset array of UA, that what we call by

splits an array into two virtual entities, the sorted array and the unsorted array. In the beginning the unsorted array contains all the unsorted elements. The sorted array starts with one element that is sorted (of course one element is sorted).

The iteration at starts the second element of the unsorted array, We take the second element from the unsorted array and compare it to the first element of the sorted array if the second element from the unsorted array is smaller than the first element of the sorted array then we place the element to the left if not we don’t need to do anything. At this stage the sorted array stays sorted.

## The Pseudocode

**Input :**  An array of n numbers, in arbitrary order

**Output:**  An array of the same numbers sorted from smallest to largest

**Assumptions :** None

number\_of\_elements := array\_size, starting\_index = 1 // in 0 based indexing, 1 is second element

**for** index := starting\_index **to** number\_of\_elements -1 **do** // Forward path

key := array[index]

previous\_index = index – 1

**while** previous\_index >= 0 **and** array[previous\_index] > key **do** // Backward pass,

array[index] = array[previous\_index]

array[previous\_index] = key

previous\_index = previous\_index – 1 / /decrement

returns sorted\_array

## The Analysis

## 

## The implementation

### Python

### Rust

# MergeSort

## The Example

7,9,5,3,4,8,1,9

## Runtime Analysis

### Theorem

For every input array of length n>= 1, the merge sort algorithm performs at most

We assume that the array size, n, is a power of 2, to simplify the numbers whenever using log2.

### Analysis and proof

A computer screen shot of a chart

Description automatically generated