

Bubble Sort

Bubble sort, also referred to as comparison sort, is a simple sorting algorithm that repeatedly goes through the list, compares adjacent elements and swaps them if they are in the wrong order. This is the most simple algorithm and inefficient at the same time. Yet, it is very much necessary to learn about it as it represents the basic foundations of sorting. We compare adjacent elements and see if their order is wrong (i.e. $a[i] > a[j]$ for $0 \leq i < j \leq \text{size of array} - 1$; if array is to be in ascending order, and vice-versa). If yes, then swap them.

1. Let us say, we have an array of length n . To sort this array we do the above step (swapping) for $n - 1$ passes.
2. In simple terms, first, the largest element goes at its extreme right place then, second largest to the last by one place, and so on. In the i th pass, the i th largest element goes at its right place in the array by swappings.
3. In mathematical terms, in i th pass, at least one element from $(n - i + 1)$ elements from start will go at its right place. That element will be the i th (for $1 \leq i \leq n - 1$) largest element of the array. Because in the i th pass of the array, in the j th iteration (for $1 \leq j \leq n - i$), we are checking if $a[j] > a[j + 1]$, and $a[j]$ will always be greater than $a[j + 1]$ when it is the largest element in range $[1, n - i + 1]$. Now we will swap it. This will continue until i th largest element is at the $(n - i + 1)$ th position of the array.

An Example

Consider the following array: Arr=14, 33, 27, 35, 10. We need to sort this array using bubble sort algorithm.


0	1	2	3	4
14	33	27	35	10

First Pass: We proceed with the first and second element i.e., Arr[0] and Arr[1]. Check if $14 > 33$ which is false. So, no swapping happens and the array remains the same.

0	1	2	3	4
14	33	27	35	10

We proceed with the second and third element i.e., Arr[1] and Arr[2]. Check if $33 > 27$ which is true. So, we swap Arr[1] and Arr[2].

0	1	2	3	4
14	33	27	35	10



Thus the array becomes:


0	1	2	3	4
14	27	33	35	10

We proceed with the third and fourth element i.e., Arr[2] and Arr[3]. Check if $33 > 35$ which is false. So, no swapping happens and the array remains the same.

0	1	2	3	4
14	27	33	35	10

We proceed with the fourth and fifth element i.e., Arr[3] and Arr[4]. Check if $35 > 10$ which is true. So, we swap Arr[3] and Arr[4].

0	1	2	3	4
14	27	33	35	10



Thus, after swapping the array becomes:

0	1	2	3	4
14	27	33	10	35

Thus, marks the end of the first pass, where the Largest element reaches its final(last) position.


Second Pass: We proceed with the first and second element i.e., Arr[0] and Arr[1]. Check if $14 > 27$ which is false. So, no swapping happens and the array remains the same.

0	1	2	3	4
14	27	33	10	35

We now proceed with the second and third element i.e., Arr[1] and Arr[2]. Check if $27 > 33$ which is false. So, no swapping happens and the array remains the same.

We now proceed with the third and fourth element i.e., Arr[2] and Arr[3]. Check if $33 > 10$ which is true. So, we swap Arr[2] and Arr[3].

0	1	2	3	4
14	27	33	10	35



Now, the array becomes

0	1	2	3	4
14	27	10	33	35

Thus marks the end of second pass where the second largest element in the array has occupied its correct position.

Third Pass: After the third pass, the third largest element will be at the third last position in the array.

0	1	2	3	4
14	10	27	33	35

i-th Pass: After the ith pass, the ith largest element will be at the ith last position in the array.

n-th Pass: After the nth pass, the nth largest element(smallest element) will be at nth last position(1st position) in the array, where 'n' is the size of the array.

After doing all the passes, we can easily see the array will be sorted.

Thus, the sorted array will look like this:

0	1	2	3	4
10	14	27	33	35

```
def bubbleSort(arr):
    n = len(arr)

    for i in range(n-1):
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

```
arr = [64, 34, 25, 12, 22, 11, 90]
```

```
bubbleSort(arr)
```

```
for i in range(len(arr)):
    print (arr[i],end = ' ')
```

Merge Sort

start-index: x

end-index: r

mid = $x + (r-x)/2$

First-Part: x to m

12 14 3 15 17 1
x = 0, r = 5, m = 2

Second-Part: m+1 to r

(1)

12 14 3
x = 0, r = 2, m = 1
(2)

15 17 1
x = 3, r = 5, m = 4
(9)

12 14
x = 0, r = 1
m = 0
(3)

3
x = 2, r = 2
m = 2
(7)

15 17
x = 3, r = 4
m = 3
(10)

1
x = 5, r = 5
m = 5
(14)

12 14
x = 0 x = 1
r = 0 r = 1
m = 0 m = 1
(4) (5)

15 17
x = 3 x = 4
r = 3 r = 4
m = 3 m = 4
(11) (12)

12 14
x = 0, r = 1
m = 0
(6)

15 17
x = 3, r = 4
m = 3
(13)

3 12 14
x = 0
r = 2
m = 1
(8)

1 15 17
x = 3
r = 5
m = 4
(15)

1 3 12 14 15 17
x = 0
r = 5
m = 2
(16)

```
def merge(arr, l, m, r):
```

```
    n1 = m - l + 1
```

```
    n2 = r - m
```

```
    # create temp arrays
```

```
    L = [0] * (n1)
```

```
    R = [0] * (n2)
```

```
    # Copy data to temp arrays L[] and R[]
```

```
    for i in range(0, n1):
```

```
        L[i] = arr[l + i]
```

```
    for j in range(0, n2):
```

```
        R[j] = arr[m + 1 + j]
```

```
    i, j, k = 0, 0, 1
```

```
    while i < n1 and j < n2 :
```

```

        if L[i] <= R[j]:
            arr[k] = L[i]
            i += 1
        else:
            arr[k] = R[j]
            j += 1
        k += 1

    while i < n1:
        arr[k] = L[i]
        i += 1
        k += 1

    while j < n2:
        arr[k] = R[j]
        j += 1
        k += 1

def mergeSort(arr,l,r):
    if l < r:
        m = (l+(r-1))//2
        mergeSort(arr, l, m)
        mergeSort(arr, m+1, r)
        merge(arr, l, m, r)

arr = [12, 11, 13, 5, 6, 7]
n = len(arr)
mergeSort(arr,0,n-1)
print ("\n\nSorted array is")
for i in range(n):
    print ("%d" %arr[i])

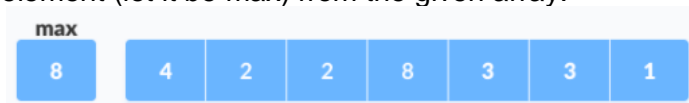
```

Counting sort

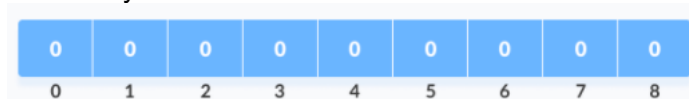
Counting sort is a sorting algorithm that sorts the elements of an array by counting the number of occurrences of each unique element in the array. The count is stored in an auxiliary array and the sorting is done by mapping the count as an index of the auxiliary array.

Working of counting sort

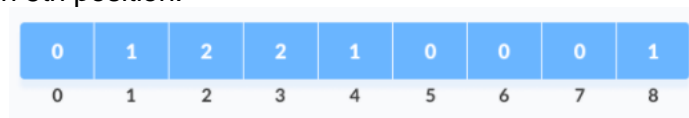
1. Find out the maximum element (let it be max) from the given array.



2. Initialize an array (Say count) of length max+1 with all elements 0. This array is used for storing the count of the elements in the array.



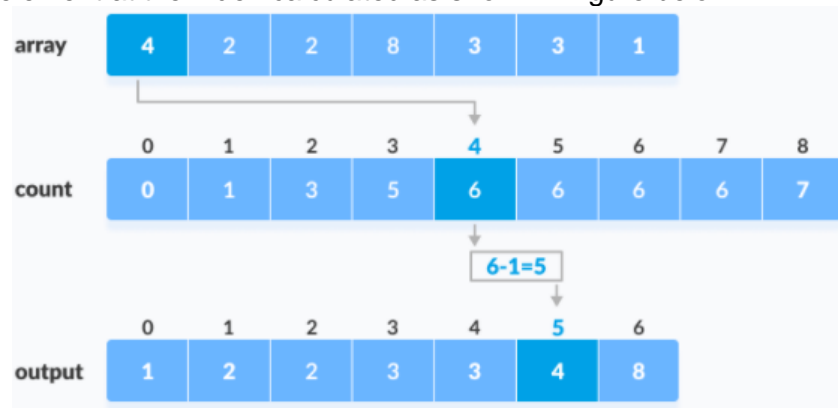
3. Store the count of each element at their respective index in count array. For example: if the count of element 3 is 2 then, 2 is stored in the 3rd position of count array. If element "5" is not present in the array, then 0 is stored in 5th position.



4. Store cumulative sum of the elements of the count array. It helps in placing the elements into the correct index of the sorted array.

0	1	3	5	6	6	6	6	7
0	1	2	3	4	5	6	7	8

5. Find the index of each element of the original array in the count array. This gives the cumulative count. Place the element at the index calculated as shown in figure below.



6. After placing each element at its correct position, decrease its count by one.

```
def countingSort(array):
    size = len(array)
    output = [0] * size

    # Initialize count array
    count = [0] * 10

    # Store the count of each elements in count array
    for i in range(0, size):
        count[array[i]] += 1

    # Store the cumulative count
    for i in range(1, 10):
        count[i] += count[i - 1]

    # Find the index of each element of the original array in count array
    # place the elements in output array
    i = size - 1
    while i >= 0:
        output[count[array[i]] - 1] = array[i]
        count[array[i]] -= 1
        i -= 1

    # Copy the sorted elements into original array
    for i in range(0, size):
        array[i] = output[i]
```

```
data = [4, 2, 2, 8, 3, 3, 1]
countingSort(data)
```