

# Advanced Sorting Algorithm

🕒 Created	@July 30, 2022 1:18 PM
▼ Class	
▼ Type	
🔗 Materials	
☑ Reviewed	<input type="checkbox"/>

We already know about comparison based sorting algorithms (Bubble, selection, insertion, merge, quick). Now there are a set of sorting algorithms that are based on counting.

## Counting Sort

Let's say we have the following array - [2,3,1,1,2,4,56,7,3,2,8,1,2] and we need to sort it. Let's say we don't have knowledge of any previous sorting algorithm. Then how we could have sorted it ?

The most intuitive approach should be, get all occurrences of the minimum element and arrange it to the start of array. Then take the second min element and all of its occurrences and arrange it after the min element and so on and so forth.

So technically, this approach is called as counting sort. We can just count the frequency of all the elements and store it in a frequency map. Then place all the occurrences of the smallest element in the start, then second smallest , then third smallest and so on and so forth.

```
[2, 3, 1, 1, 2, 4, 56, 7, 3, 2, 8, 1, 2]
frequency map - [0, 3, 4, 2, 0, 0, 1, 1, 0, 0, 0, 0, 0, ..... 1, 0, 0, 0, , , ,]
1-3
2-4
3-2
4-1
7-1
8-1
56-1

[1, 1, 1, 2, 2, 2, 2, 3, 3, 4, 7, 8, 56]
```

## How to prepare frequency map here ?

Let's assume we have only positive elements, then we can use an array to prepare the frequency map ?

`arr[i]` → will store frequency of `i`

```
function countingSort(arr) {
  let maxElement = Math.max(...arr); // O(n)
  let freq = Array(maxElement + 1).fill(0); // O(1)
  for(let i = 0; i < arr.length; i++) { //O(n)
    freq[arr[i]]++;
  }
  let k = 0;
  for(let i = 0; i < freq.length; i++) { // O(maxElement + n)
    while(freq[i] > 0) {
      arr[k] = i;
      k++;
      freq[i]--;
    }
  }
}
```

```
freq = [0,0,0,0,0,0,0,1,1,0,0,0,0,0, ..... 1, 0,0,0,,,,]
arr = [1,1,1,2,2,2,2,3,3,2,8,1,2]
i = 5, k = 9

[2,1,9999999];
```

## Disadvantages

- If the max element is huge, counting sort will perform very bad for even very small arrays
- Space is also extra
- Unstable → this current implementation is unstable. We can modify the implementation without hampering the time and space and make counting sort stable.

## How to make it stable ?

```

arr = [2,1,3,2,1,4,5,3,2,6]
freq = [0,2,3,2,1,1,1] // convert this freq array to prefix sum array
pre-freq = [0,2,4,6,8,9,9]
final output = [0,0,0,0,2,0,3,0,0,6]
indexes -      1,2,3,4,5,6,7,8,9,10

```

Assume 1 based indexing in the final output array. If we carefully observe, then `pre-freq[i]` it is denoting the last index in the final output array, where `i` should be placed.

Example: `pre-freq[1] = 2` → and in the final output array the last occurrence of 1 is at position `2`

Example: `pre-freq[2] = 5` → and in the final output array the last occurrence of 2 is at position `5`

Example: `pre-freq[5] = 9` → and in the final output array the last occurrence of 5 is at position `9`

Now because we know where the last occurrence of any element should be placed, can we start reading the array from behind, so that for any element we first encounter the last occurrence and then with the position of the last occurrence we calculated, we can place it.

```

function countingSortStable(arr) {
// Time: O(n+k) Space: O(n+k)
  let maxElement = Math.max(...arr); // O(n)
  let freq = Array(maxElement + 1).fill(0); // O(1)
  for(let i = 0; i < arr.length; i++) { //O(n)
    freq[arr[i]]++;
  }
  // freq -> prefix sum array
  for(let i = 1; i < freq.length; i++) { // O(k)
    freq[i] = freq[i] + freq[i-1];
  }
  // console.log(freq);
  let output = Array(arr.length);
  for(let i = arr.length - 1; i >= 0; i--) { //O(n)
    let currelement = arr[i];
    let index = freq[currelement]; // index- 1 based
    output[index - 1] = currelement; // stored based on 0 based indexing
    freq[currelement]--;
  }
}

```

```
    return output;
}
```

## In the current implementation, why negatives won't be handled ?

In the current implementation, we are storing the frequency of elements inside an array, every index represents the element and the value at that index represents the frequency. But if the element is negative, we don't have negative indexes in the array.

## How to handle this ?

In the current implementation we are assuming elements to be from 0 to some positive value. And we want to modify this behaviour to some negative element to some positive element.

In the current implementation the mapping of index to element is :

Index	Element
0	0
1	1
2	2
3	3
4	4
.	.
.	.
.	.
X	X

If now we want to change the behaviour of the algorithm so that instead of  $[0, X]$  we can support  $[-Y, X]$  the mapping should look like:

Index	Element
0	-Y

1	$-Y + 1$
2	$-Y + 2$
3	$-Y + 3$
4	$-Y + 4$
5	$-Y + 5$
.	.
.	.
.	.
Y	0
Y + 1	1
Y + 2	2
Y + 3	3
Y + 4	4
Y + 5	5
Y + 6	6
.	.
.	.

For Example: We want to support  $[-3, 4]$

Index	Element
0	-3
1	-2
2	-1
3	0
4	1
5	2
6	3
7	4

This process looks like **Shifting of origin**

So if we have an element **A** then the index of the element will be  **$A - \text{Minimum\_Element}$**

Ex:  $A = 3$ , index of  $A = 3 - (-3) \Rightarrow 6$

So, all we have to do to start the support of negative is while considering any element  $A$  consider it as  $A - \text{min\_element}$ , so the length of the frequency array will be changed from  $\text{max\_element} + 1$  to  $\text{max\_element} - \text{min\_element} + 1$

```
function countingSort(arr) {
  let maxEl = Math.max(...arr);
  let minEl = Math.min(...arr);
  let range = maxEl - minEl + 1;
  let freq = new Array(range).fill(0);
  for(let i = 0; i < arr.length; i++) {
    let currElement = arr[i];
    freq[currElement - minEl]++;
  }
  for(let i = 1; i < freq.length; i++) {
    freq[i] += freq[i-1];
  }
  let output = new Array(arr.length);
  for(let i = arr.length - 1; i >= 0; i--) {
    let currelement = arr[i];
    let index = freq[currelement - minEl];
    output[index - 1] = currelement;
    freq[currelement - minEl]--;
  }
  return output;
}
```

## Radix Sort

Let's say we have two positive numbers  $x$  and  $y$

And we know that  $x < y$

Now lets say we have two more positive number  $a$  and  $b$

And we know that  $a < b$

can we setup some relation between  $xa$  and  $yb$  ?

Then  $xa < yb$

$xa$  can be written as  $10 * x + a$

$yb$  can be written as  $10 * y + b$

Let's derive:

```

x < y
multiplying 10 on both sides
10*x < 10*y
and a < b
adding a on lhs and b on rhs
10*x + a and 10*y + b will follow the relation
10*x + a < 10*y + b
Because we are adding smaller value i.e. `a` on the smaller side i.e. `10*x` and bigger value i.e. `b` on the bigger side `10*y`

This will be true for only 2 digit numbers
for multiple digits
x*(10^place-1) + a < y*(10^place-1) + b irrespective of the fact whether a is less than b or greater.

```

Now, let's say we have an array:

[129,431,234,653,232,824,2,921,54]

Now, let's forget about all the digits in every number except digits on ones place.

So array technically looks like - [9,1,4,3,2,4,2,1,4]

If we sort our data based on just the ones place then it will look like

[431,921,232,2,653,234,824,054,129]

Now if we try to sort the data based on tens place viz → [3,2,3,0,5,3,2,5,2]

Sorted array based on tens place looks like → [2,921,824,129,431,232,234,653,54]

Now let's try to map any two numbers considering their 10s and 1s place as **a** and **b** and same two numbers considering their 100th place as **x** and **y**, and map these four numbers based on the relation stated above

So let's pick 824 and 234 such that

**a = 24, b = 34**

and

**x = 8, y = 2**

$a < b$  and  $y < x$ , so if I write,  $yb$  and  $xa$ , the relationship will be

```

x > y
100*x > 100*y -> 100*x + a > 100*y + b
a < b
Smaller value will be added on bigger side, so what will be the relation ?

```

```
So as 8 is the most significant digit, the contribution will be the highest , so addition of small value doesn't matter
```

So, what I am trying to convey here is, in radix sort, if we have two values like 824 and 234 because  $8 > 2$  it doesn't matter what are the remaining digits, it's sure that 824 is bigger than 234.

But In cases where the most significant bit is equal, 234 and 232 here  $2 == 2$  so the number which will be bigger will be decided by the remaining digits.

Hence we can say,

```
if x < y
and doesn't matter a is less than b or not
xa will be always less than yb
```

So if we use this technique to sort the array then we can say that, if we have all the numbers sorted with respect to 10s and 1s place already, then we can easily sort them based on the 100th place by just comparing the 100th place of each element with each other. If the 100th place of an element is bigger than the whole element is bigger, but if the 100th place is equal, then because the numbers are already sorted with respect to 10s and 1s place so the number will be automatically arranged in correct order.

That's why

[2,921,824,129,431,232,234,653,54]

We try to sort the array with respect to 100th place, then

If we consider 2 and 54, their 100th place are equal i.e. 0, but remaining elements have 100th place greater than 0 so it's sure that 2 and 54 are less than remaining elements, and we know 54 is place after 2 in the above array as the above array is already sorted based on 10s and 1s place so 2 is less than 54.

[2,54,...]

Same thing for 129, it has 1 in the 100th place so it is less than the remaining elements

[2,54,129...]



So this whole observation gives us intuition about Radix sort which says, if we want to sort the data, we can sort the numbers place by place, at last we will be having sorted output.

[129, 431, 234, 653, 232, 824, 2, 921, 54]

129		431		002		002
431		921		921		054
234		232		824		129
653		002		129		232
232	=>	653	=>	431	=>	234
824	1's place	234	10's place	232	100's place	431
002		824		234		653
921		054		653		824
054		129		054		921

So what radix sort does is, it starts sorting the numbers w.r.t 1s place then 10s place then 100th place and so on and so forth. For sorting the numbers w.r.t a place it uses counting sort. Why counting sort ? Because when we are sorting the data based on a particular place, then we can have at max 10 unique values i.e.[0,9] so range is always fixed, so complexity of every sorting call is  $O(n)$

And this counting sort will be called  $d$  number of time where  $d$  is the max number of digits in a given integer of the array.

and there is a relation between  $d$  and max element which is  $d \sim \log_b \text{maxelement}$  where  $b$  is the number of unique digits the number system is having

So for binary  $b$  is 2 for decimal  $b$  is 10 for octal  $b$  is 8 etc.

So, complexity will be  $O(n + b) * d$  as for decimals  $b$  is always 10 so this is going to be  $O(n) * d$  and  $d \sim \log_b \text{maxelement}$  so complexity is  $O(n * \log_b \text{maxelement})$

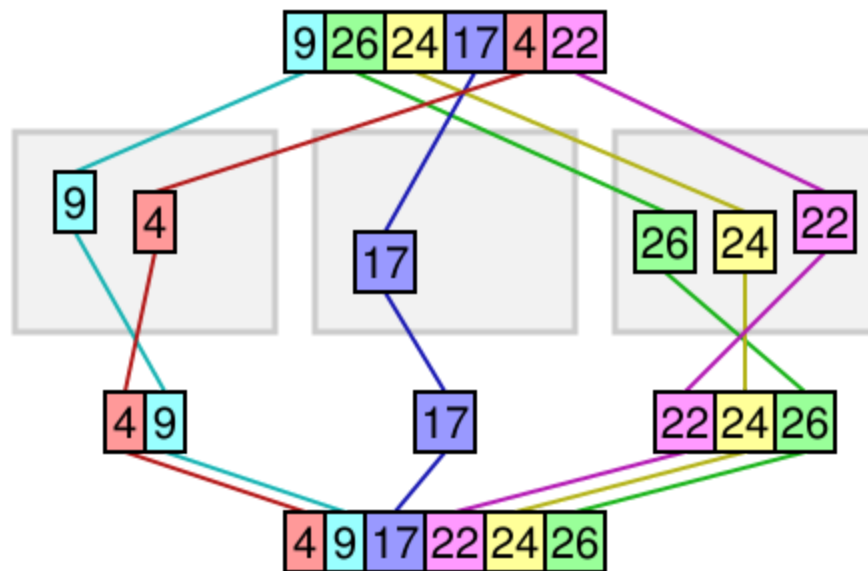
## Disadvantages:

- It is based on number of digits. So for very small array having large numbers it's not going to perform well.
- It's not inplace as it uses counting sort, so it wastes  $O(n)$  space

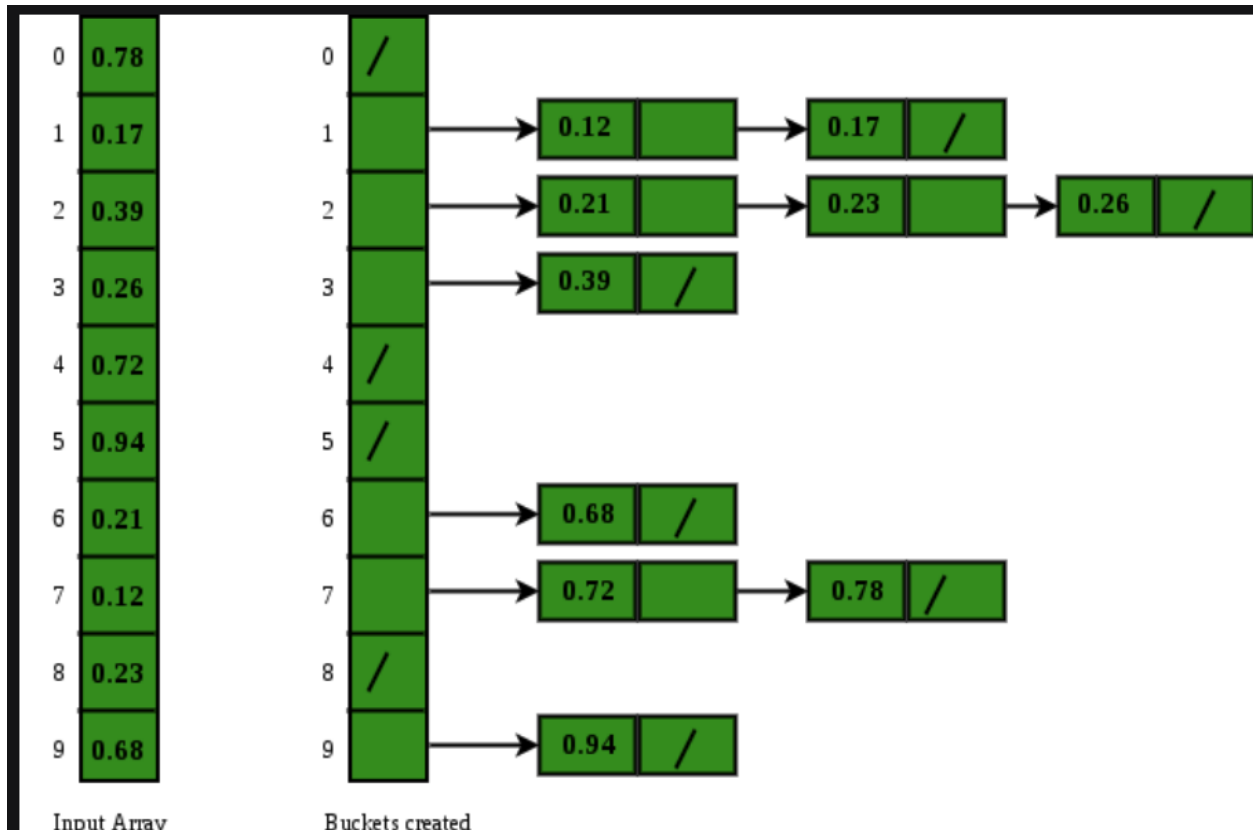
## HW, code radix sort

### Bucket sort

Bucket sort is a very simple sorting algorithm, in which we distribute our elements if the array into a different number of buckets based on ranges, Each bucket is then sorted individually, either using a different sorting algorithm. Then concatenate the results of the individual buckets together




So sometimes to sort decimal number we can use bucket sort, as we can do the bucketing based on the 1st decimal digit or first two decimal digits, and then sort the individual buckets and combine the answers.



```
function bucketSort(arr) { // implementation to support decimal sorting from 0-1
  let buckets = Array(arr.length);
  for(let i = 0; i < arr.length; i++) {
    let bucketIndex = Math.floor(arr[i]*10);
    buckets[bucketIndex].push(arr[i]);
  }
  for(let i = 0; i < buckets.length; i++) {
    buckets[i].sort((a, b) => a-b);
  }
  let output = [];
  for(let i = 0; i < buckets.length; i++) { // n
    for(let j = 0; j < buckets[i].length; j++) {
      output.push(buckets[i][j]);
    }
  }
  return output;
}
```

## Bucket Sort Algorithm

Bucket sort is an algorithm used to assign elements into a bucket to be sorted.... Seems easy enough...🤖 The sorting is usually done by using an algorithm of a second type... Yes...you will need to

 <https://medium.com/@alanna.noguchi/bucket-sort-algorithm-67373f7cbef1>



Now we assume that the bucketing algorithm distributes the elements so well, that each bucket has got very less elements so that it can be sorted in almost constant time. So in the best case, complexity will be  $\Omega(n+k)$   $n \rightarrow$  for going to all the elements and distributing in buckets,  $k \rightarrow$  for sorting  $k$  different buckets in constant time.

But in the worst case, may be we have non uniform distribution of data,  $O(n*n)$