
# DEPARTMENT OF COMPUTER SCIENCE

## COMP242

## Project #4
## Sorting Algorithms

- **Instructor : Dr. Radi Jarrar**
- **Student's Name : Bara Adnan**
- **Student's ID : 1161357**
- **Date : 11/30/2017**

## *Table Of Contents :*

# Counting Sort

## Counting Sort

Counting Sort does not work for negative numbers

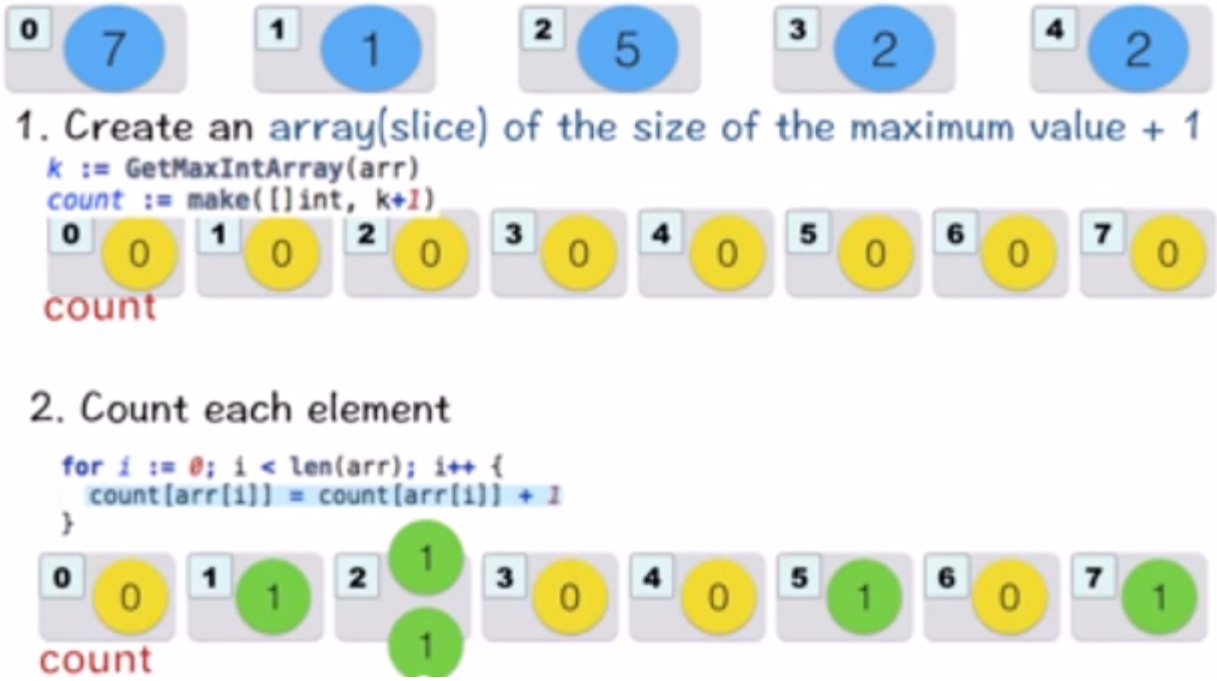
Counting Sort assumes that each element is SMALL INTEGER

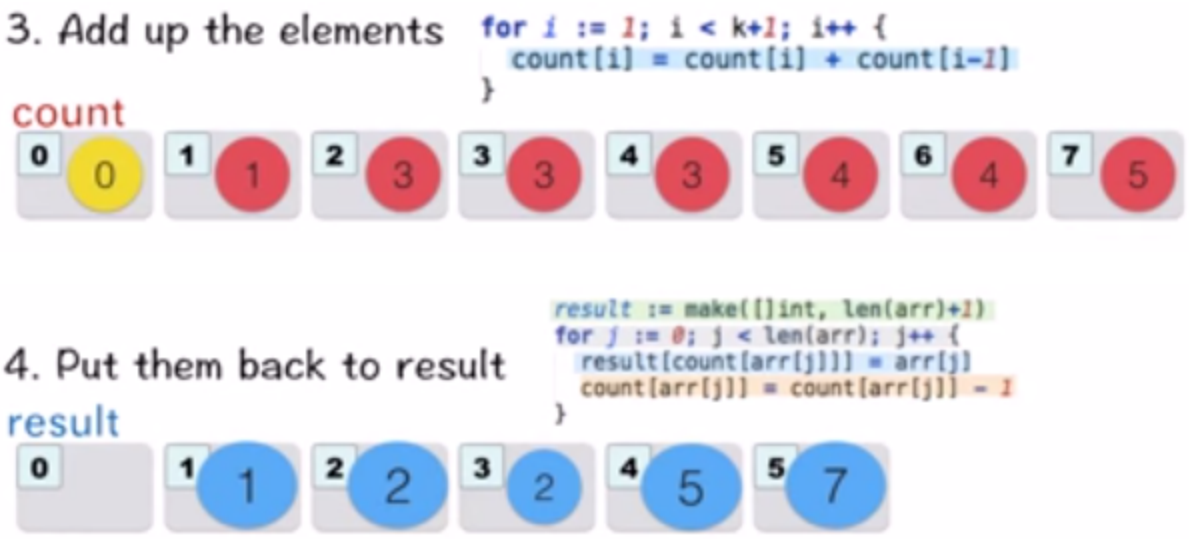
Running time is O(Maximum value - Minimum value)

which is linear

So it is useful only when the difference is NOT large

## *Algorithm :*






# *Advantages :*

We can take advantage of the situation to produce a linear - O(**n**) - sorting algorithm. That's Counting sort. assume that we are asked to sort **n** elements, but we are informed that each element is in the range of **0-k**, where ***k*** is much smaller than **n**. And it is stable, as counting sort can not only be used for sorting a list of integers, it can also be used for sorting a list of elements whose key is an integer, and these elements will be sorted by their keys while having additional information associated with each of them. if, for example, we have a list of students sorted by last name, then a counting sort is used to sort by grade A B C D F, the final list will be sorted primarily by grade, but all the students who got the same grade will still be sorted by last name.

# *Time :*

Time complexity is O(n+k) in all cases, where n is the number of elements in input array and k is the range of input.

## *Space :*

Counting sort does not sort in place, since it creates two other arrays, counting array C and output array. Auxiliary Space: O(n+k)

# Bucket Sort

Works by partitioning an array into a number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm.

## *Algorithm :*

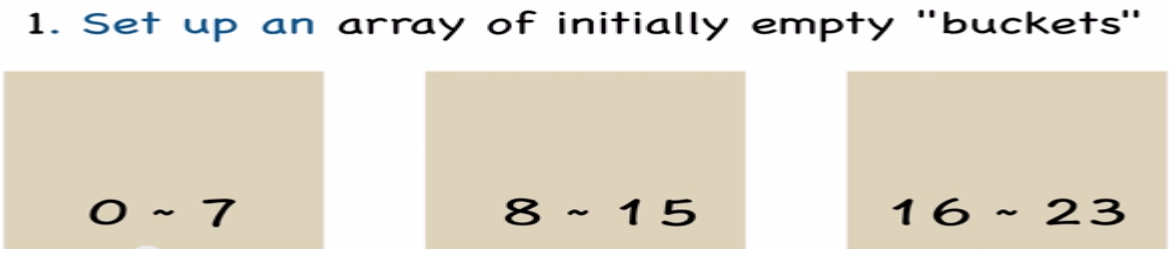
1. Set up an array of initially empty "buckets"

2. Scatter: Traverse the original array and extract each element into its bucket

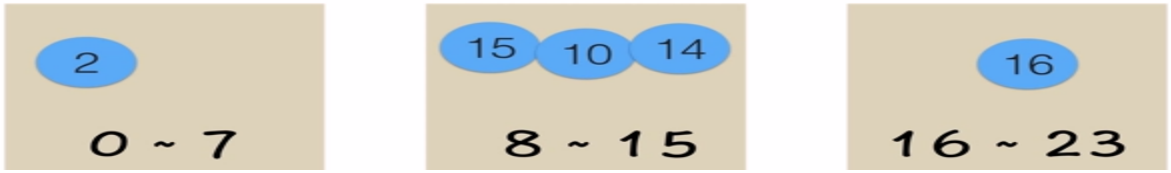
3. Sort each non-empty buckets

4. Visit the buckets and put them back in order

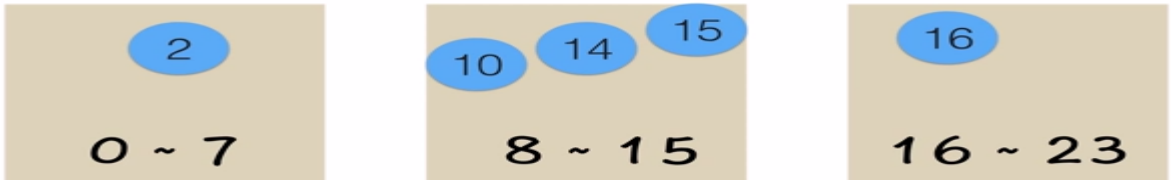
***FOR INSTANCE***


2. Scatter: Traverse the original array and extract each element into its bucket




3. Sort each non-empty buckets




4. Visit the buckets and put them back in order


## *Advantages :*

This sort is most effective when an optimal number of buckets are used so as to best complement the sort. And it's a stable sorting algorithm.

## *Time :*

Worst Case $O(n^2)$ Average Case $\Theta(n)$ Best Case $\Omega(n)$

## *Space :*

Bucket sort is not an in place sorting algorithm. Buckets require extra space.
Worst Space Complexity: $O(n)$

# Comb Sort

Comb Sort, an extension over Bubble sort. This sorting technique helps removing the rabbits and turtles from the traditional bubble sort approach. The basic idea is to eliminate turtles, or small values near the end of the list, since in a bubble sort these slow the sorting down tremendously.

*Shrink Factor* is the factor by which the gap is made to shrink between iterations. Tests show that *SF* = 1.3 gives optimal performance.
*Gap* = (*List Length*)/(*SF*)

## *Algorithm :*

- Start at one end of the unsorted list
  - Compare entries that are separated by the *gap* as calculated
    *gap* = (*List Length*)/(*SF*)
- Swap if they are in the wrong order
- Move on to the next entry if they are in the right order
- *New gap = gap/SF*
- Continue comparison and swapping until *gap* <= 1
- Bubble sort here on

## *Advantages :*

Comb sort is a pretty advanced sorting algorithm with average case under O(nlogn), it is reliable, and unlike other high level sorting algorithm, it is relatively simplistic, and in the same time a stable one.

## *Time and space complexities:*

Best Case $\Omega(n)$, Average Case $\Theta(n \log n)$, Worst Case $O(n^2)$

Auxiliary Space: O(1), as it is an in-place sorting algorithm.

*Example :*

*gap* = 5/1.3 = 4 (Rounded off to nearest integer)

| It. | *gap* | Current List | Result |
|---|---|---|---|
| 1 | 4 | **3**, 5, 7, 2, **0** | **0**, 5, 7, 2, **3** |
| 2 | 3 | **0**, 5, 7, **2**, 3 | **0**, 5, 7, **2**, 3 |
| | | 0, **5**, 7, 2, **3** | 0, **3**, 7, 2, **5** |
| 3 | 2 | **0**, 3, **7**, 2, 5 | **0**, 3, **7**, 2, 5 |
| | | 0, **3**, 7, **2**, 5 | 0, **2**, 7, **3**, 5 |
| | | 0, 2, **7**, 3, **5** | 0, 2, **5**, 3, **7** |
| 4 | 1 | **0**, **2**, 5, 3, 7 | **0**, **2**, 5, 3, 7 |
| | | 0, **2**, **5**, 3, 7 | 0, **2**, **5**, 3, 7 |
| | | 0, 2, **5**, **3**, 7 | 0, 2, **3**, **5**, 7 |
| | | 0, 2, 3, **5**, **7** | 0, 2, 3, **5**, **7** |

# Odd – Even Sort

In computing, an odd–even sort (or brick sort) is a relatively simple sorting algorithm, developed originally for use on parallel processors with local interconnections. It is a comparison sort related to bubble sort, with which it shares many characteristics.

## *Algorithm :*

It functions by comparing all odd/even indexed pairs of adjacent elements in the list and, if a pair is in the wrong order (the first is larger than the second) the elements are switched. The next step repeats this for even/odd indexed pairs (of adjacent elements). Then it alternates between odd/even and even/odd steps until the list is sorted.

```
function oddEvenSort(list) {
 function swap( list, i, j ){
  var temp = list[i];
  list[i] = list[j];
  list[j] = temp;
 }

 var sorted = false;
 while(!sorted)
 {
  sorted = true;
  for(var i = 1; i < list.length-1; i += 2)
  {
   if(list[i] > list[i+1])
   {
    swap(list, i, i+1);
    sorted = false;
   }
  }

  for(var i = 0; i < list.length-1; i += 2)
```

```
    {
     if(list[i] > list[i+1])
     {
       swap(list, i, i+1);
       sorted = false;
     }
    }
  }
}
```

# *Advantages :*

Simple, easy to implement. It is considered a stable sorting algorithm.

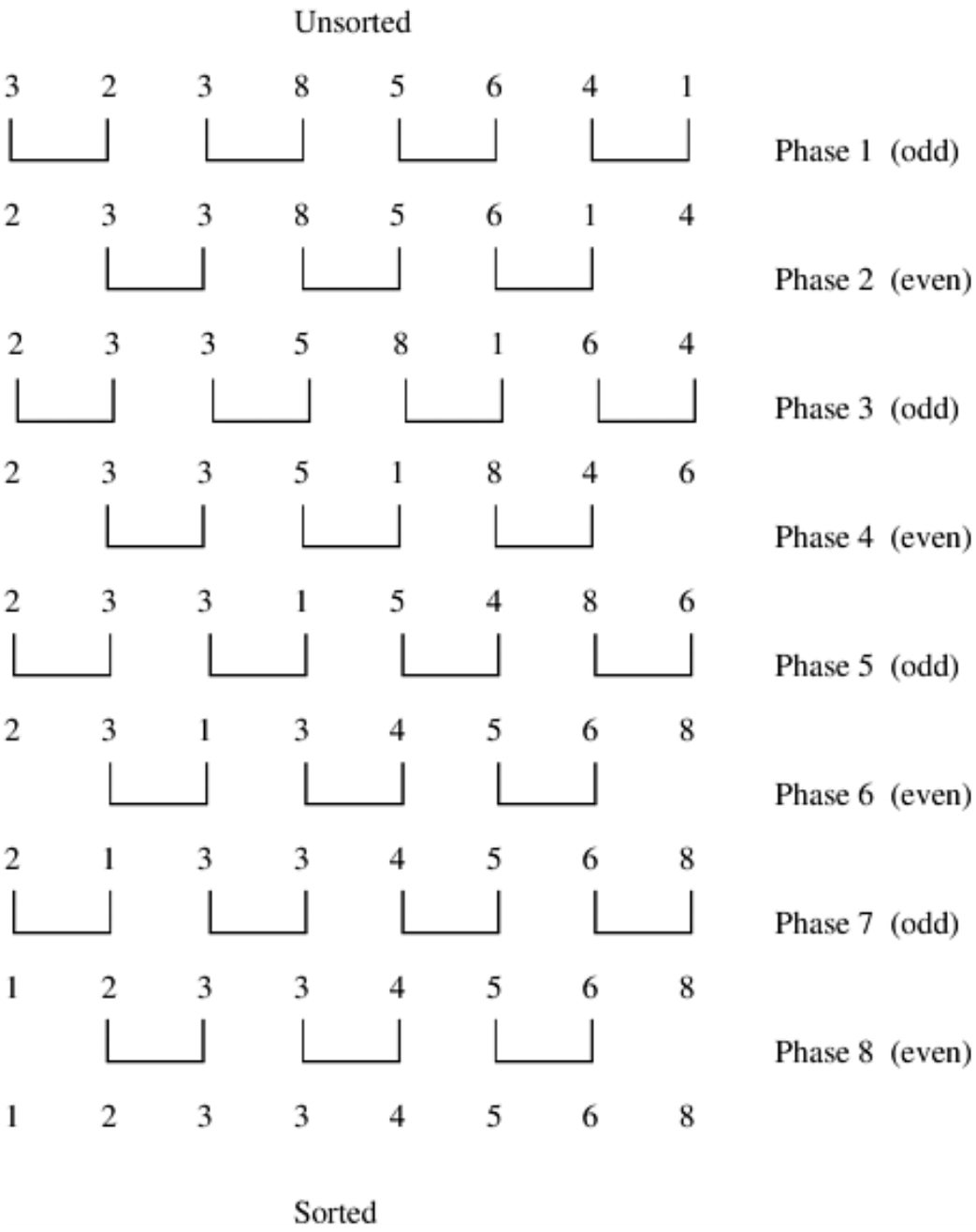
# *Time and space complexities:*

In best case, if the data is sorted, then 2 loops (n/2 each) will be performed only without any swap operation, so the time complexity will be O(n). But if it is not sorted, then the operation will take more than 1*n loop, and in the worst case the time complexity will be $O(n^2)$.

Auxiliary Space: O(1), as it is an in-place sorting algorithm.

| | |
|---|---|
| **Worst case performance** | $O(n^2)$ |
| **Average case performance** | $\Theta(n^2)$ |
| **Best case performance** | $\Omega(n)$ |
| **Worst case space complexity** | $O(1)$ |

## *Example :*


Unsorted
3 2 3 8 5 6 4 1
Phase 1 (odd)
2 3 3 8 5 6 1 4
Phase 2 (even)
2 3 3 5 8 1 6 4
Phase 3 (odd)
2 3 3 5 1 8 4 6
Phase 4 (even)
2 3 3 1 5 4 8 6
Phase 5 (odd)
2 3 1 3 4 5 6 8
Phase 6 (even)
2 1 3 3 4 5 6 8
Phase 7 (odd)
1 2 3 3 4 5 6 8
Phase 8 (even)
1 2 3 3 4 5 6 8
Sorted

# Cocktail sort

Cocktail sort, also known as Cocktail shaker sort, bidirectional bubble sort, shaker sort, ripple sort, shuffle sort, or shuttle sort, is a stable comparison sorting algorithm. It is a variation of bubble sort, that is both a stable sorting algorithm and a comparison sort.

In bubble sort, values only bubble in one direction. In cocktail sort, values bubble both directions, thus avoiding turtles.

## *Algorithm :*

Each iteration of the algorithm is broken up into 2 stages:

1.The first stage loops through the array from left to right, just like the Bubble Sort. During the loop, adjacent items are compared and if value on the left is greater than the value on the right, then values are swapped. At the end of first iteration, largest number will reside at the end of the array.

2.The second stage loops through the array in opposite direction- starting from the item just before the most recently sorted item, and moving back to the start of the array. Here also, adjacent items are compared and are swapped if required.

## *Advantages :*

Relatively simple sorting algorithm. Performs better than Bubble sort. Time complexities are same, but typically cocktail sort is less than two times faster than bubble sort. Consider the example (2,3,4,5,1). Bubble sort requires four traversals of array for this example, while Cocktail sort requires only two traversals.

# *Time and space complexities:*

Worst and Average Case Time Complexities: $\Theta(n^2)$.
Best Case Time Complexity: $\Omega(n)$. Best case occurs when array is already sorted.

Auxiliary Space (space complexity): O(1)

Sorting In Place: Yes
Stable: Yes

# *Know Thy Complexities!*

| Algorithm | Time Complexity | | | Space Complexity | Stable | Sorting in place |
|---|---|---|---|---|---|---|
| | **Best** | **Average** | **Worst** | **Worst** | | |
| **Counting Sort** | $\Omega(n+k)$ | $\Theta(n+k)$ | $O(n+k)$ | $O(n+k)$ | Yes | No |
| **Bucket Sort** | $\Omega(n)$ | $\Theta(n)$ | $O(n^2)$ | $O(n)$ | Yes | No |
| **Comb Sort** | $\Omega(n)$ | $\Theta(n \log n)$ | $O(n^2)$ | $O(1)$ | Yes | Yes |
| **Odd-Even Sort** | $\Omega(n)$ | $\Theta(n^2)$ | $O(n^2)$ | $O(1)$ | Yes | Yes |
| **Cocktail sort** | $\Omega(n)$ | $\Theta(n^2)$ | $O(n^2)$ | $O(1)$ | Yes | Yes |

# *References*


[1] http://en.wikipedia.org/wiki/Sorting_algorithm#Comb_sort

[2] http://www.youtube.com/watch?v=FjoNnz9fwVU

[3] http://www.youtube.com/playlist?list=PLT6aABhFfinum-5PglO49yOzD9c7o6T2N

[4] http://www.youtube.com/watch?v=jVXsjswWo44

[5] http://en.wikipedia.org/wiki/Timsort

[6] http://www.beingjavaguys.com/2013/08/insertion-sort-in-java.html

[7] http://www.algorithmist.com/index.php/Counting_sort

[8] http://www.growingwiththeweb.com/2016/10/odd-even-sort.html