*I.J. Mathematical Sciences and Computing,* **2019, 2, 30-43**
Published Online April 2019 in MECS (http://www.mecs-press.net)
DOI: 10.5815/ijmsc.2019.02.03

***Available online at*** ***http://www.mecs-press.net/ijmsc***

# Sorting using a combination of Bubble Sort, Selection Sort & Counting Sort

Sahil Kumar, Prerna Singla

*Department of Computer Science, Punjabi University, Patiala, 147002, India.*

Received: 12 July 2018; Accepted: 16 October 2018; Published: 08 April 2019

**Abstract**

One of the most important problems in computer science is the ordering of the data. Although sorting is a very old computer science problem, it still attracts a great deal of research. Usually, when we face a problem, we're concerned with finding the solution, then getting it out of our heads and into a text editor, white-board, or down on a piece of paper. Eventually, we start transforming that idea into code, and the code is pretty terrible the first time around. But at some point, once we've made it work and made it right, we find ourselves asking: Can I make it fast? Can I make it better? This paper presents an enhanced sorting algorithm which comprises of a combination of Bubble Sort, Selection Sort, and Counting Sort. The new algorithm is analyzed, implemented, tested, compared and the results were promising.

**Index Terms:** Sorting, bubble sort, selection sort, counting sort.



## 1. Introduction

In today's world, size of the data is increasing at a rapid pace, so to search or to perform various operations data needs to be sorted in many cases. Therefore, sorting has become very important now a days.

Sorting is basically the process of arranging the data in a sequence. In computer science, a sorting algorithm is a process that undergoes intermediate steps to arrange the elements in a particular order-ascending or descending. Many algorithms are present that help us to sort the unordered data like Bubble sort, Selection sort,

*Corresponding author.
E-mail Address: sahilucoe@hotmail.com, prernasingla23@gmail.com

Quick sort, Merge sort and the list is endless.

In this research paper, we have combined three Sorting Algorithms- Bubble Sort, Selection Sort and Counting Sort.

Bubble Sort is a simple comparison-based sorting algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. Bubble sort is not suitable for large data sets as its average and worst case complexities are O ($n^2$) where n is the number of items.

Selection sort is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is entire list. The smallest element is selected from the unsorted array and swapped with the leftmost element, and the element becomes a part of the sorted array. This process continues till the elements are sorted. This algorithm is also not suitable for large data sets as its average and worst case complexities are of O ($n^2$) where n is the number of items.

Counting sort can be used when we know the set of numbers to be sorted are positive integers. This sorting algorithm counts the number of unique objects and uses some math to determine their position to arrange the numbers in sorted order.

## 2. Bubble Sort

Bubble sort is a sorting algorithm, which is commonly used in computer science. It is a simple comparison-based sorting algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. Bubble sort is not suitable for large data sets as its average and worst case complexities are O ($n^2$) where n is the number of items. We will discuss the complexity of bubble sort in detail.

*Procedure*

The procedure of the algorithm is described in following steps:

- Calling “BUBBLE_SORT” function, passing the array and its size as parameters.
- Repeat the following two steps i times where i is equal to n-1 and n is the size of the array.
- Starting from the first element of the array till j, repeat the following step where j is equal to n-i-1.
- Compare the current element with next element of the array. If the current element is greater than the next element of the array swap them.
- Return from “BUBBLE_SORT” function.

*Pseudo code*

*function BUBBLE_SORT (array, size)*

```
1. If size>1 then:
2. var swapped, iteration, index
3.   for iteration: 0 to size-1 do:
4.         swapped=false
5.       for index:0 to size-i-1 do:
6.             If array[index]>array[index+1] do:
7.                   Swap array [index] and array [index+1]
8.                       swapped=true
9.             End if
10.       End for
```

```
11.          If swapped is false
12.               Return array
13.          End if
14.    End for
15. End if
16. Return array
```

*Analysis*

Suppose we have an unsorted array as follows:

| 5 | 1 | 4 | 2 | 8 |
|---|---|---|---|---|

**In First Pass**: Algorithm compares first two elements and swaps since 5 >1.

| 1 | 5 | 4 | 2 | 8 |
|---|---|---|---|---|

Swap second and third elements since 5>4.

| 1 | 4 | 5 | 2 | 8 |
|---|---|---|---|---|

Swap third and forth elements since 5>2.

| 1 | 4 | 2 | 5 | 8 |
|---|---|---|---|---|

Since 5 and 8 are already in correct order, therefore they are not swapped.

| 1 | 4 | 2 | 5 | 8 |
|---|---|---|---|---|

**In Second Pass**:Since 1 and 4 are already in correct order, therefore they are not swapped.
Swap second and third elements since 4>2.

| 1 | 2 | 4 | 5 | 8 |
|---|---|---|---|---|

Since 4, 5 and 8 are already in correct sequence there they are not swapped.

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one more pass without any swap to know it is completed. Therefore after third pass, elements of the array will be sorted.

*Time Complexity*

In Bubble Sort, n-1 comparisons will be done in first iteration, n-2 comparisons will be done in second iteration, n-3 comparisons can be done in third iteration and so on. So the total number of comparisons can be defined as,

$$T(n) = (n-1) + (n-2) + (n-3) + \ldots\ldots + 3 + 2 + 1$$
$$= n*(n-1)/2$$
$$= O(n^2)$$

Best Case: When array is already sorted (in ascending order):

When array is already sorted in ascending order, (n-1) comparisons will be done and it will break. Therefore, in best case, the complexity will be:

$$T(n) = O(n-1)$$
$$= O(n)$$

Worst Case and Average Case complexities of Bubble sort algorithm is $O(n^2)$.

The main advantage of Bubble Sort is simplicity of the algorithm. The space complexity of the algorithm is O (1).

**3. Selection Sort**

Selection sort is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is entire list. The smallest element is selected from the unsorted array and swapped with the leftmost element, and the element becomes a part of the sorted array. This process continues till the elements are sorted. This algorithm is also not suitable for large data sets as its average and worst case complexities are of O ($n^2$) where n is the number of items.

*Procedure*

The procedure of the algorithm is described in following steps:

- Calling "SELECTION_SORT" function, passing the array and its size as parameters.
- Repeat the following two steps i times where i is equal to n-1 and n is the size of the array.
- Repeat the next step j times where j ranges from i+1 to n.
- Search the minimum element 'min' in the list
- Swap 'min' with the $i^{th}$ element of the array.
- Return from "SELECTION_SORT" function.

*Pseudo code*

*function SELECTION_SORT (array, size)*

```
1. If size>1 then:
2.         var iteration, j
3.      for iteration: 0 to size-1 do:
    /* set current element as minimum*/
4.             min = i
     /* check the element to be minimum */
5.              for j: i+1 to size do:
```

```
6.                  If array[j] < array[min] then
7.                      min = j;
8.                  End if
9.              End for
/* swap the minimum element with the current element*/
10.             If min ! = i    then
11.                 Swap array [min] and array[i]
12.             End if
13.         End for
14. End if
```

*Analysis*

Suppose we have an unsorted array as follows:

| 10 | 26 | 12 | 21 | 69 |
|---|---|---|---|---|

In First step, we will find the minimum element in array from index 0 to index 4 and place it at the beginning. Since the minimum element is 10, the array remains the same:

| 10 | 26 | 12 | 21 | 69 |
|---|---|---|---|---|

In Second step, we will find the minimum element in array from index 1 to index 4 and place it at index 1. Since the minimum element is 12, the array becomes:

| 10 | 12 | 26 | 21 | 69 |
|---|---|---|---|---|

In Third step, we will find the minimum element in array from index 2 to index 4 and place it at index 2. Since the minimum element is 21, the array becomes:

| 10 | 12 | 21 | 26 | 69 |
|---|---|---|---|---|

In Forth step, we will find the minimum element in array from index 3 to index 4 and place it at index3. Since the minimum element is 26, the array becomes:

| 10 | 12 | 21 | 26 | 69 |
|---|---|---|---|---|

This is the array sorted in ascending order using Selection Sort.

*Time Complexity*

In Selection Sort, selecting the lowest element requires scanning all n elements(this takes n-1 comparisons in first iteration) and then swapping it with the first position. Finding the next lowest element requires scanning the remaining (n-1) elements (this takes n-2 comparisons) and so on. So the total number of comparisons can be defined as,

$$T(n) = (n-1) + (n-2) + (n-3) + \ldots\ldots\ldots + 3 + 2 + 1$$
$$= n*(n-1)/2$$
$$= O(n^2)$$

So, time complexity of Selection Sort Algorithm is $O(n^2)$.

**4. Counting Sort**

In computer science, counting sort is an algorithm for sorting a collection of objects according to keys that are small integers; that is, it is an integer sorting algorithm. It operates by counting the number of objects that have each distinct key value, and using arithmetic on those counts to determine the positions of each key value in the output sequence. Its running time is linear in the number of items and the difference between the maximum and minimum key values, so it is only suitable for direct use in situations where the variation in keys is not significantly greater than the number of items. However, it is often used as a subroutine in another sorting algorithm.

Because counting sort uses key values as indexes into an array, it is not a comparison sort, and the Ω (n log n) lower bound for comparison sorting does not apply to it. Bucket sort may be used for many of the same tasks as counting sort, with a similar time analysis; however, compared to counting sort, bucket sort requires linked lists, dynamic arrays or a large amount of pre-allocated memory to hold the sets of items within each bucket, whereas counting sort instead stores a single number (the count of items) per bucket.

*Procedure*

The procedure of the algorithm is described in following steps:

- Calling “COUNTING_SORT” function, passing the array “arr” and its size as parameters.
- Declare an array “c*ount”* where the indices represent numbers from our input array and the values represent how many times the index number appears. Start each value at 0.
- In one pass of the input array, update “count” as you go, so that at the end the values in “count” are correct.
- Declare an array “sortedArray” where we'll store our sorted numbers.
- In one in-order pass of “count” put each number, the correct number of times, into “sortedArray”.
- Return “sortedArray” from “COUNTING_SORT” function.

*Pseudo code*

```
function COUNTING_SORT(array, size)

1. If size>1 then:
2.      var iteration , k

//initialize count array
3. for iteration: =0 to k do:
4.      count[iteration]=0
5. End for

// store frequency of element x in count[x-1]
```

```
6. for iteration: = 0 to size do:
7.      count [array[iteration]]++
8. End for

// after this loop, count[y] will contain number of elements less than or equal to (y)
9. for iteration: =2 to k do:
10.     count[iteration]=count[iteration]+count[iteration-1]
11. End for
12. declare an array temp of size equal to size of array "arr"

// after this loop temp array will contain the sorted elements
13. for iteration:size to 0 do:
14.      temp[count[array[iteration]]]=array[iteration]
15       count[array[iteration]]--
15. End for

// copy sorted elements from temp array to array
17. for iteration: =0 to temp.length do:
18.     array [iteration]=temp[iteration]
19. End for
20. End if
```

*Analysis*

Suppose we have an unsorted array as follows:

| 1 | 4 | 1 | 2 | 7 | 5 | 2 |
|---|---|---|---|---|---|---|

For simplicity, consider the data set in the range 0 to 7. Take a count array to store the count of each unique object.

| 0 | 2 | 2 | 0 | 1 | 1 | 0 | 1 |
|---|---|---|---|---|---|---|---|

Modify the count array such that each element at each index.

| 0 | 2 | 4 | 4 | 5 | 6 | 6 | 7 |
|---|---|---|---|---|---|---|---|

The modified count array indicates the position of each object in the output sequence.

Output each object from the input sequence followed by decreasing its count by 1. Store data in temp array. Example, Position of 1 is 2. Put the data 1 at index 2 in temp array. Decrease the count by 1 to place next data 1 at an index 1 smaller than this index. Therefore, array after sorting becomes:

| 1 | 1 | 2 | 2 | 4 | 5 | 7 |
|---|---|---|---|---|---|---|

*Time Complexity:*

In Counting sort algorithm, suppose each of the elements is an integer in the range 1 to k:

Loop of step 3 takes O (k) time.
Loop of step 6 takes O (n) time.
Loop of step 9 takes O (k) time.
Loop of step 13 takes O (n) time.

Therefore, the overall time complexity of counting sort can be defined as:

$$T(n,k) = O(k) + O(n) + O(k) + O(n)$$
$$= O(k+n)$$

## 5. Combination of Bubble & Selection Sort

The concept behind the combination of Bubble and Selection Sort is finding the minimum element and swapping it with the first element and finding the maximum element and then swapping it with the last element of the array. The process continues for n/2 iterations where n is the number of elements in an array. The complexity of this algorithm is O ($n^2$). Although complexity of bubble and selection sort is O ($n^2$) as well, but the number of iterations in this algorithm is half as compared to bubble and selection sort individually.

*Procedure*

The procedure of the algorithm is described in following steps:

- Calling "BS_SORT" function, passing the array and its size as parameters.
- Repeat the following two steps till you reach the middle element of the array.
- Find the minimum and maximum from the array and swap it with the first element and last element of the array respectively.
- Reduce the size of array to be sorted by 1 from both the sides of the array.
- Return from "BS_SORT" function.

*Pseudo code*

```
function BS_SORT (array, size)

1. If size>1 then:
2.      var iteration, index
3.      for iteration: =0 to size/2 do:
4.              for index: =iteration to (size-iteration-1) do
5.                      If array (index)>array (index+1)
6.                              Swap array (index) and array (index+1)
7.                      End if
8.                      If array (iteration)>array (index)
9.                          Swap array (iteration) and array (index)
10.                     End if
11.             End for
12.     End for
13. End if
14. Return array
```

## 6. Combination of Bubble, Selection and Counting Sort

While using the combination of Bubble and Selection Sort, suppose after k iterations, difference between maximum and minimum number in an unsorted array is less than the number of unsorted items. If so, then we will apply Counting Sort to the remaining items.

While applying counting sort we subtract minimum element of the array from position (k) to position (n -k) in the first step. After this, we apply the counting sort and then at last we add the minimum element to all the sorted elements from position (k) to position (n-k). Counting sort is an algorithm for sorting a collection of objects according to keys that are small integers; that is, it is an integer sorting algorithm. It operates by counting the number of objects that have each distinct key value, and using arithmetic on those counts to determine the positions of each key value in the output sequence.

*Procedure*

The procedure of the algorithm is described in the following steps supposing counting sort algorithm is applied after k iterations:

- Call "BCS_SORT" function, passing the array and its size as parameters.
- Repeat the following two steps till it fulfills the condition of counting sort(i.e. difference between maximum and minimum number in the unsorted array is less than the number of unsorted items) and Call "countingSort" function. If it does not fulfill condition of counting sort then repeat the following two steps till you reach the middle element of the array.
- Find the minimum and maximum from the array and swap it with the first element and last element of the array respectively.
- Reduce the size of array to be sorted by 1 from both the sides of the array.
- Call "countingSort" function, passing the array "arr", k as *startIndex* ,(n-k) as *endIndex* , *minimum* element from index (k) to index (n-k) and the *maximum* element from index (k) to index (n-k).
- Define a variable '*range*' = *maximum-minimum*+1.
- Declare an array *Count* of length '*range*'.
- Array *Count* stores the frequency of each element in the array "arr".
- After that array *Count* stores the position of each element in an array using Arithmetic.
- Declare array *temp* of size (*endIndex-startIndex*).
- Array *temp* will store sorted elements of the array "arr" from index *startIndex* to *endIndex*.
- In the last step, copy the elements from the array temp to array "arr" from index *startIndex* till *endIndex* of array "arr".
- Return from "BCS_SORT" function.

*Pseudo code*

*function BCS_sort(array, size)*

```
1. If size>1 then:
2.      var iteration, index , max, min
3.      for iteration: =0 to size/2 do:
4.              for index: = iteration to (size-iteration-1) do
5.                      If array (index)>array (index+1)
```

```
6.                      Swap array (index), array (index+1)
7.                  End if
8.                  If array (iteration)>array (index)
9.                      Swap array (iteration) & array (index)
10.             End if
11.         End for
12.         max=array(size - iteration - 1)
13.         min=array(iteration)
14          if (max-min) < size – (2*iteration):
15.             call counting_sort(array, iteration,
          (size- iteration), max, min)
16.           Return array
17.       End if
18.   End for
19. End if
20. Return array

function counting_sort (array, startIndex, endIndex, max, min)
1. var range= maximum- minimum+1,
2. Declare an array count of size range

//initialize count array
3. for index: =0 to range do:
4.      count[index]=0
5. End for

// store frequency of element x in count[x-min]
5. for index: = startIndex to endIndex do:
6.      count [array[index]-min]++
7. End for

// after this loop, count[y] will contain number of elements less than or equal to (y+min)
8. for index: =1 to range do:
9.      count[index]=count[index]+count[index-1]
10. End for

11. Declare an array temp of size (endIndex-startIndex)


// after this loop temp array will contain the sorted elements
12. for index:startIndex to endIndex do:
13.       temp[count[array[index]-min]-1]=array[index]
14        count [array [index]-min]--
15. End for

// copy sorted elements from temp array to array
16. for index: =0 to temp.length do:
17.      array [startIndex+index]=temp[index]
18. End for
```

*Analysis*

Suppose we have an unsorted array as follows:

| 50 | 5 | 6 | 4 | 3 | 3 | 2 |
|---|---|---|---|---|---|---|

According to the algorithm, after first iteration the array becomes:

| 2 | 6 | 5 | 4 | 3 | 3 | 50 |
|---|---|---|---|---|---|---|

Maximum element is equal to 50 and minimum elements equals to 2. Since difference between maximum and minimum elements is not less than size of the array – (2*iteration), therefore it goes for second iteration.

After second iteration the array becomes:

| 2 | 3 | 5 | 4 | 3 | 6 | 50 |
|---|---|---|---|---|---|---|

Since difference between maximum and minimum elements is less than size of the array – (2*iteration), therefore it goes inside counting sort function:

Counting sort function initializes an array count of size 4 to store the count of each unique object:

| 2 | 1 | 1 | 1 |
|---|---|---|---|

After that modify the count array such that each element at each index:

| 2 | 3 | 4 | 5 |
|---|---|---|---|

After this step, data is stored in array temp in sorted form:

| 3 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|

Copy the data from array temp from index 1 to index 5 in array and return. The array after copying the data becomes:

| 2 | 3 | 3 | 4 | 5 | 6 | 50 |
|---|---|---|---|---|---|---|

*Time Complexity:*

Suppose after k iterations, function counting_sort is invoked then, the time complexity of BCS_sort can be defined as:

$$T(n, k) = [O(k) * O(n)] + [O(n-2k) + O(n-2k) + O(n-2k) + O(n- 2k) + O(n-2k)]$$
$$= [O(k) * O(n)] + 5 * [O(n-2k)]$$
$$= [O(k) * O(n)] + [O(n-2k)]$$
$$= O(k*n)$$

Best Case: When counting sort is invoked after 1st iteration i.e. k=1:

$$T(n) = O(n)$$

Worst Case: When counting sort is never invoked i.e. k=n;

$$T(n) = O(n^2)$$

## 7. Comparison with Bubble Sort

Let's call the algorithm proposed in the paper BCS Sort. Let's observe the differences in the time complexities of Bubble Sort Algorithm and BCS Sort Algorithm.

Table 1. shows the main difference between the time complexities of Bubble Sort Algorithm and BCS Sort Algorithm.

Table 1. Main Difference between the Time Complexities of Bubble Sort Algorithm and BCS Sort Algorithm.

| | Bubble Sort | BCS Sort |
|---|---|---|
| Best Case | O(n) | O(n) |
| Average Case | $O(n^2)$ | O(k*n) |
| Worst Case | $O(n^2)$ | $O(n^2)$ |

Now, we will observe differences in the running time of both the algorithms on different datasets.

Table 2 shows the running time (ns) of BCS Sort, Bubble Sort for sorting a random dataset containing elements less than *100*. Horizontally we have taken the size of dataset or the number of items to be sorted.

Table 2. Running time (ns) of BCS Sort, Bubble Sort for Sorting a Random Dataset Containing Elements Less than 100

| | 100 | 1000 | 10000 | 50000 |
|---|---|---|---|---|
| BCS | 14897 | 76555 | 629875 | 3888563 |
| Bubble | 141887 | 2635815 | 98362612 | 3231598743 |

Table 3 shows the running time (ns) of BCS Sort, Bubble Sort for sorting a random dataset containing elements less than *1000*. Horizontally we have taken the size of dataset or the number of items to be sorted.

Table 3. Running time (ns) of BCS Sort, Bubble Sort for Sorting a Random Dataset Containing Elements Less than 1000

| | 100 | 1000 | 10000 | 50000 |
|---|---|---|---|---|
| BCS | 38836 | 53707 | 171633 | 509525 |
| Bubble | 18907 | 866356 | 90310624 | 3288393645 |

Table 4 shows the running time (ns) of BCS Sort, Bubble Sort for sorting a random dataset containing elements less than *10000*. Horizontally we have taken the size of dataset or the number of items to be sorted.

Table 4. Running time (ns) of BCS Sort, Bubble Sort for Sorting a Random Dataset Containing Elements Less than 10000

| | 1000 | 10000 | 20000 | 50000 |
|---|---|---|---|---|
| BCS | 1376091 | 387857 | 421234 | 537869 |
| Bubble | 859892 | 94829710 | 444733194 | 3266561839 |

Table 5 shows the running time (ns) of BCS Sort, Bubble Sort for sorting a random dataset containing elements less than *100000*. Horizontally we have taken the size of dataset or the number of items to be sorted.

Table 5. Running time (ns) of BCS Sort, Bubble Sort for Sorting a Random Dataset Containing Elements Less than 100000

| | 1000 | 10000 | 20000 | 50000 |
|---|---|---|---|---|
| BCS | 715750 | 77564546 | 416515115 | 2796099665 |
| Bubble | 658070 | 89211844 | 435138355 | 3231805384 |

## References

[1] Aho A., Hopcroft J., and Ullman J., *The Design and Analysis of Computer Algorithms*, Addison Wesley, 1974.
[2] http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/countingSort.htm
[3] Niklaus Wirth, Algorithms + Data Structures=Programs, 1976.
[4] Thomas H. Cormen, Charles E. Leiserson, Ronald Rivest Clifford Stein, *Introduction to Algorithms*, 1989.
[5] Adam Drozdek, Data Structures and Algorithms in C++, 1995.
[6] Robert Lafore, Data Structures and Algorithms in JAVA, 1998.
[7] Hemant Jain, Problem Solving in Data Structures and Algorithms Using Java, 2016.
[8] Robert Sedgewick, *Algorithms*, 1983.
[9] Bruno R. Preiss, Data Structures and Algorithms, 1998.
[10] Mark Allen Weiss, Data Structures and Algorithms in C++, 1993.
[11] Ellis Horowitz, Fundamentals of data structures, 1975.
[12] Richard F. Gilberg, *Data Structures*, 1998.
[13] William Ford, Data Structures with C++, 1996.
[14] James A Storer, An Introduction to Data Structures and Algorithms, 2001.

## Authors' Profiles

**Sahil Kumar** currently working as Full Stack Developer in ZS Associates Pvt. Ltd. He holds B.Tech degree in Computer Science. He is well organized, reliable and enthusiastic computer science engineer with experience in .Net, C#, HTML, CSS, Javascript, Nhibernate, Jquery and AngularJS.

**Prerna Singla** is a Software Developer in Cleartrip Pvt. Ltd. She pursued her B.Tech degree in Computer Science from Punjabi University, Patiala. She has worked on various technologies like Java, Elasticsearch, SpringBoot etc.

**How to cite this paper:** Sahil Kumar, Prerna Singla,"Sorting using a combination of Bubble Sort, Selection Sort & Counting Sort", International Journal of Mathematical Sciences and Computing(IJMSC), Vol.5, No.2, pp.30-43, 2019.DOI: 10.5815/ijmsc.2019.02.03