

BUBBLE SORT

Priyanka kunjwal

*Dronacharya College of Engineering
Haryana*

ABSTRACT:-Sorting algorithms provide a way to arrange a series of numbers or letters in some predefined order based on some measurable quantity in the numbers or letters. Thus we may arrange a series of numbers according to their values in an increasing order or we may arrange the letters according to decreasing order of their ASCII values using sorting. In this paper we will describe a simple and easy to implement sorting algorithm called Bubble Sort.

INTRODUCTION

Given a list of n numbers or letters the objective of any sorting algorithm is to arrange the same in a particular order where the ordering is done based on some intrinsic property of the inputs. The simplest way to sort a list of input values is to compare them pairwise and obtain the proper ordering. Sorting algorithms that achieve their goal by comparison are called comparison based sorting algorithm. Bubble Sort is a simple and easy to implement comparison based sorting algorithm. We will describe the algorithm by sorting a list of n numbers in increasing order of magnitude. The same process can be followed to sort a list of letters according to some criteria. The input to the algorithm will be a list of n numbers in a random order and the output will be a list of the same n numbers arranged in an increasing order of magnitude.

PERFORMANCE

Bubble sort has worst-case and average complexity both $O(n^2)$, where n is the number of items being sorted. There exist many sorting algorithms with substantially better worst-case or average complexity of $O(n \log n)$. Even other $O(n^2)$ sorting algorithms, such as insertion sort, tend to have better performance than bubble sort. Therefore, bubble sort is not a practical sorting algorithm when n is large.

The only significant advantage that bubble sort has over most other implementations, even quicksort, but not insertion sort, is that the ability to detect that the list is sorted is efficiently built into the algorithm.

When the list is already sorted (best-case), the complexity of bubble sort is only $O(n)$. By contrast, most other algorithms, even those with better average-case complexity, perform their entire sorting process on the set and thus are more complex. However, not only does insertion sort have this mechanism too, but it also performs better on a list that is substantially sorted (having a small number of inversions).

Bubble sort should be avoided in the case of large collections. It will not be efficient in the case of a reverse-ordered collection

STEP BY STEP EXAMPLE

Let us take the array of numbers "5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort. In each step, elements written in bold are being compared. Three passes will be required.

First Pass

(5 1 4 2 8) \rightarrow (1 5 4 2 8), Here, algorithm compares the first two elements, and swaps since $5 > 1$.

(1 5 4 2 8) \rightarrow (1 4 5 2 8), Swap since $5 > 4$ (1 4 5 2 8) \rightarrow (1 4 2 5 8), Swap since $5 > 2$ (1 4 2 5 8) \rightarrow (1 4 2 5 8),

Now, since these elements are already in order ($8 > 5$), algorithm does not swap them.

Second Pass

(1 4 2 5 8) \rightarrow (1 4 2 5 8)

(1 4 2 5 8) \rightarrow (1 2 4 5 8), Swap since $4 > 2$ (1 2 4 5 8) \rightarrow (1 2 4 5 8)

(1 2 4 5 8) \rightarrow (1 2 4 5 8)

Now, the array is already sorted, but the algorithm does not know if it is completed. The algorithm needs one whole pass without any swap to know it is sorted.

Third Pass

(1 2 4 5 8) → (1 2 4 5 8) (1 2 4 5 8) → (1 2 4 5 8)
 (1 2 4 5 8) → (1 2 4 5 8) (1 2 4 5 8) → (1 2 4 5 8)

IMPLEMENTATION

PSEUDOCODE IMPLEMENTATION:

The bubble sort algorithm can be easily optimized by observing that the n-th pass finds the n-th largest element and puts it into its final place. So, the inner loop can avoid looking at the last n-1 items when running for the n-th time:

```
procedure bubble Sort( A : list of sortable items ) n
= length(A)
```

```
repeat
swapped = false
```

```
for i = 1 to n-1 inclusive do if A[i-1] > A[i] then
swap(A[i-1], A[i])
```

```
swapped = true end if
```

```
end for n = n - 1
```

```
until not swapped end procedure
```

More generally, it can happen that more than one element is placed in their final position on a single pass. In particular, after every pass, all elements after the last swap are sorted, and do not need to be checked again. This allows us to skip over a lot of the elements, resulting in about a worst case 50% improvement in comparison count (though no improvement in swap counts), and adds very little complexity because the new code subsumes the "swapped" variable.

To accomplish this in [pseudocode](#) we write the following:

```
procedure bubbleSort( A : list of sortable items ) n =
length(A)
```

```
repeat
```

```
new n = 0
```

```
for i = 1 to n-1 inclusive do if A[i-1] > A[i] then
swap(A[i-1], A[i])
```

```
new n = i
```

```
end if end for n = newn
```

```
until n = 0 end procedure
```

BUBBLE SORT EXAMPLE

```
/* Bubble sort code */
```

```
#include <stdio.h>
```

```
int main()
```

```
{
```

```
int array[100], n, c, d, swap;
```

```
printf("Enter number of elements\n"); scanf("%d",
&n);
```

```
printf("Enter %d integers\n", n);
```

```
for (c = 0; c < n; c++) scanf("%d", &array[c]);
```

```
for (c = 0 ; c < ( n - 1 ); c++)
```

```
{
```

```
for (d = 0 ; d < n - c - 1; d++)
```

```
{
```

```
if (array[d] > array[d+1]) /* For decreasing order
use < */
```

```
{
swap = array[d];
```

```
array[d] = array[d+1];
```

```
array[d+1] = swap;
```

```
}
```

```
}
```

```
}
```

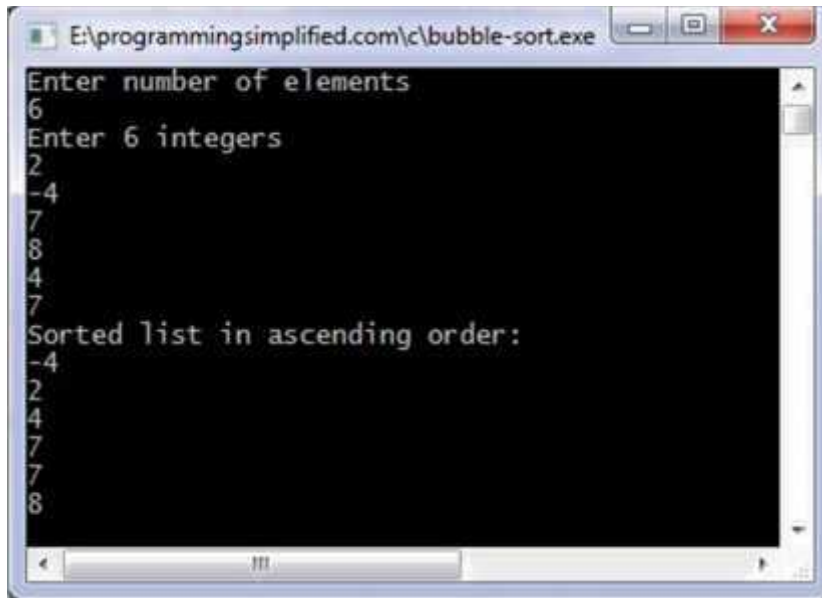
```
printf("Sorted list in ascending order:\n");
```

```
for ( c = 0 ; c < n ; c++ ) printf("%d\n", array[c]);
```

```
return 0;
```

```
}
```

OUTPUT :



```
E:\programmingsimplified.com\c\bubble-sort.exe
Enter number of elements
6
Enter 6 integers
2
-4
7
8
4
7
Sorted list in ascending order:
-4
2
4
7
7
8
```

REFERENCES:

- [1] [Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Problem 2-2, pg.40.](#)
- [2] [Sorting in the Presence of Branch Prediction and Caches](#)
- [3] 3. Fundamentals of Data Structures by Ellis Horowitz, [Sartaj Sahni](#) and Susan Anderson-Freed [ISBN 81-7371-605-6](#)