**Bubble Sort**

# Bubble Sort

Bubble sort (sinking sort), is a simple sorting algorithm that repeatedly steps through the list to be sorted, compares each pair of adjacent items and swaps them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.

The algorithm, which is a comparison sort, is named for the way smaller or larger elements "bubble" to the top of the list.

Bubble sort is a stable sort algorithm, like insertion sort.

# Advantages and Application

1. It can be practical if the input is usually in sorted order but may occasionally have some out-of-order elements nearly in position.
2. In computer graphics bubble sort is popular for its capability to detect a very small error (like swap of just two elements) in almost-sorted arrays and fix it with just linear complexity (2n).
3. it is used in a polygon filling algorithm, where bounding lines are sorted by their x coordinate at a specific scan line (a line parallel to the x axis) and with incrementing y their order changes (two elements are swapped) only at intersections of two lines.

# Disadvantages

1. Although the algorithm is simple, it is too slow and impractical for most problems even when compared to insertion sort.
2. Bubble sort should be avoided in the case of large collections. It will not be efficient in the case of a reverse-ordered collection.
3. Bubble sort also interacts poorly with modern CPU hardware.
4. It produces at least twice as many writes as insertion sort, twice as many cache misses, and asymptotically more branch mispredictions.
5. Experiments by Astrachan sorting strings in Java show bubble sort to be roughly one-fifth as fast as an insertion sort and 70% as fast as a selection sort.

# Algorithm

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 ) ->{\displaystyle \to } ( **1** **5** 4 2 8 ), Here, algorithm compares the first two elements, and swaps since 5 > 1.  
( 1 **5** **4** 2 8 ) ->{\displaystyle \to } ( 1 **4** **5** 2 8 ), Swap since 5 > 4  
( 1 4 **5** **2** 8 ) ->{\displaystyle \to } ( 1 4 **2** **5** 8 ), Swap since 5 > 2  
( 1 4 2 **5** **8** ) ->{\displaystyle \to } ( 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 ) ->{\displaystyle \to } ( **1** **4** 2 5 8 )  
( 1 **4** **2** 5 8 ) ->{\displaystyle \to } ( 1 **2** **4** 5 8 ), Swap since 4 > 2  
( 1 2 **4** **5** 8 ) ->{\displaystyle \to } ( 1 2 **4** **5** 8 )  
( 1 2 4 **5** **8** ) ->{\displaystyle \to } ( 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 ) ->{\displaystyle \to } ( **1** **2** 4 5 8 )  
( 1 **2** **4** 5 8 ) ->{\displaystyle \to } ( 1 **2** **4** 5 8 )  
( 1 2 **4** **5** 8 ) ->{\displaystyle \to } ( 1 2 **4** **5** 8 )  
( 1 2 4 **5** **8** ) ->{\displaystyle \to } ( 1 2 4 **5** **8** )

# Pseudocode

## Simple Bubble Sort

The algorithm can be expressed as (0-based array):

procedure bubbleSort( A : list of sortable items )

n = length(A)

repeat

swapped = false

for i = 1 to n-1 inclusive do

/\* if this pair is out of order \*/

if A[i-1] > A[i] then

/\* swap them and remember something changed \*/

swap( A[i-1], A[i] )

swapped = true

end if

end for

until not swapped

end procedure

## Optimized Bubble Sort

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 bubbleSort( 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 than one element is placed in their final position on a single pass

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

newn = 0

for i = 1 to n-1 inclusive do

if A[i-1] > A[i] then

swap(A[i-1], A[i])

newn = i

end if

end for

n = newn

until n = 0

end procedure

# Complexity

**Worst and Average Case Time Complexity:** O(n\*n). Worst case occurs when array is reverse sorted.

**Best Case Time Complexity:** O(n). Best case occurs when array is already sorted.

**Auxiliary Space:** O(1)

**Boundary Cases:** Bubble sort takes minimum time (Order of n) when elements are already sorted.

**Sorting In Place:**Yes

**Stable:** Yes

|  |  |  |  |
| --- | --- | --- | --- |
| **Algorithm** | **Time Complexity** | | |
| Name | **Best** | **Average** | **Worst** |
| [Selection Sort](http://geeksquiz.com/selection-sort/) | Ω(n^2) | θ(n^2) | O(n^2) |
| [Bubble Sort](http://geeksquiz.com/bubble-sort/) | Ω(n) | θ(n^2) | O(n^2) |
| [Insertion Sort](http://geeksquiz.com/insertion-sort/) | Ω(n) | θ(n^2) | O(n^2) |
| [Heap Sort](http://geeksquiz.com/heap-sort/) | Ω(n log(n)) | θ(n log(n)) | O(n log(n)) |
| [Quick Sort](http://geeksquiz.com/quick-sort/) | Ω(n log(n)) | θ(n log(n)) | O(n^2) |
| [Merge Sort](http://geeksquiz.com/merge-sort/) | Ω(n log(n)) | θ(n log(n)) | O(n log(n)) |
| [Bucket Sort](https://www.geeksforgeeks.org/bucket-sort-2/) | Ω(n+k) | θ(n+k) | O(n^2) |
| [Radix Sort](https://www.geeksforgeeks.org/radix-sort/) | Ω(nk) | θ(nk) | O(nk) |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Name | Memory | Stable | Method | Other notes |
| [Selection Sort](http://geeksquiz.com/selection-sort/) | 1 | **No** | Selection | Stable with O(n) extra space, for example using lists |
| [Bubble Sort](http://geeksquiz.com/bubble-sort/) | 1 | Yes | Exchanging | Tiny code size. |
| [Insertion Sort](http://geeksquiz.com/insertion-sort/) | 1 | Yes | Insertion | O(*n* + *d*), in the worst case over sequences that have *d* [inversions](https://en.wikipedia.org/wiki/Inversion_(discrete_mathematics)) |
| [Heap Sort](http://geeksquiz.com/heap-sort/) | 1 | No | Selection |  |
| [Quick Sort](http://geeksquiz.com/quick-sort/) | log n on average worst case space complexity n  Sedgewick variation is log n worst case | Typical in-place sort is not stable; stable versions exist | Partitioning | Quicksort is usually done in-place with O(log n) stack space |
| [Merge Sort](http://geeksquiz.com/merge-sort/) | A hybrid [block merge sort](https://en.wikipedia.org/wiki/Block_sort) is O(1) mem | Yes | Merging | [Highly parallelizable](https://en.wikipedia.org/wiki/Merge_sort#Parallel_merge_sort) (up to O(log *n*) using the Three Hungarians' Algorithmor, more practically, Cole's parallel merge sort) for processing large amounts of data. |
| [Bucket Sort](https://www.geeksforgeeks.org/bucket-sort-2/) |  |  |  |  |
| [Radix Sort](https://www.geeksforgeeks.org/radix-sort/) |  |  |  |  |

# Example

1. Simple Bubble Sort

// C program for implementation of Bubble sort

#include <stdio.h>

void swap(int \*xp, int \*yp)

{

int temp = \*xp;

\*xp = \*yp;

\*yp = temp;

}

**// A function to implement bubble sort**

**void bubbleSort(int arr[], int n) {**

**int i, j;**

**for (i = 0; i < n-1; i++)**

**// Last i elements are already in place**

**for (j = 0; j < n-i-1; j++)**

**if (arr[j] > arr[j+1])**

**swap(&arr[j], &arr[j+1]);**

**}**

/\* Function to print an array \*/

void printArray(int arr[], int size)

{

int i;

for (i=0; i < size; i++)

printf("%d ", arr[i]);

printf("n");

}

// Driver program to test above functions

int main()

{

int arr[] = {64, 34, 25, 12, 22, 11, 90};

int n = sizeof(arr)/sizeof(arr[0]);

bubbleSort(arr, n);

printf("Sorted array: \n");

printArray(arr, n);

return 0;

}

Output:

Sorted array:

11 12 22 25 34 64 90

1. Optimized Bubble Sort

// Optimized implementation of Bubble sort

#include <stdio.h>

void swap(int \*xp, int \*yp)

{

int temp = \*xp;

\*xp = \*yp;

\*yp = temp;

}

**// An optimized version of Bubble Sort**

**void bubbleSort(int arr[], int n)**

**{**

**int i, j;**

**bool swapped;**

**for (i = 0; i < n-1; i++)**

**{**

**swapped = false;**

**for (j = 0; j < n-i-1; j++)**

**{**

**if (arr[j] > arr[j+1])**

**{**

**swap(&arr[j], &arr[j+1]);**

**swapped = true;**

**}**

**}**

**// IF no two elements were swapped by inner loop, then break**

**if (swapped == false)**

**break;**

**}**

**}**

/\* Function to print an array \*/

void printArray(int arr[], int size)

{

int i;

for (i=0; i < size; i++)

printf("%d ", arr[i]);

printf("n");

}

// Driver program to test above functions

int main()

{

int arr[] = {64, 34, 25, 12, 22, 11, 90};

int n = sizeof(arr)/sizeof(arr[0]);

bubbleSort(arr, n);

printf("Sorted array: \n");

printArray(arr, n);

return 0;

}

Output:

Sorted array:

11 12 22 25 34 64 90

# Variants

1. Odd–even sort is a parallel version of bubble sort, for message passing systems.
2. Cocktail shaker sort is another parallel version of the bubble sort
3. In some cases, the sort works from right to left (the opposite direction), which is more appropriate for partially sorted lists, or lists with unsorted items added to the end.

# References

<https://www.geeksforgeeks.org/sorting-algorithms/>

<https://www.geeksforgeeks.org/bubble-sort-algorithm/>