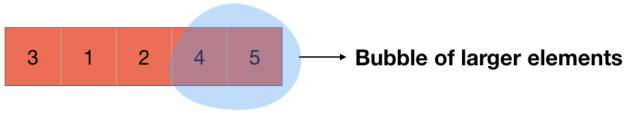
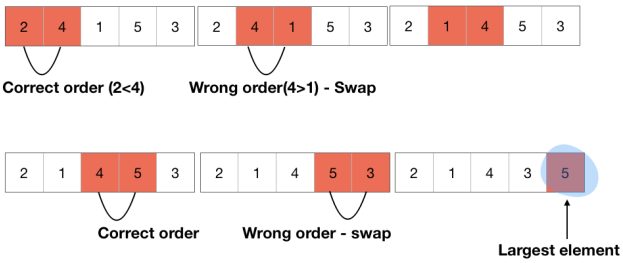
Bubble Sort

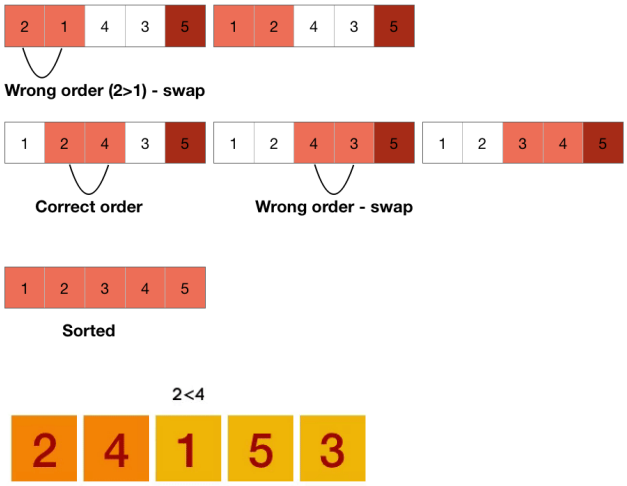
Bubble sort is a really weird name but this algorithm actually bubbles up the largest element at the end after each iteration and that's why the name is Bubble Sort.



In this algorithm, we start with iterating over an array and compare the first element to the second one and swap them if they are in the wrong order and then compare the second and the third one and so on. After this iteration, the largest element goes to the last of the array as shown in the picture given below.



Now the largest element is already in the last position, so we again repeat this process, leaving the elements which are already in their correct positions and this gives us a sorted array.



We can write the entire process in the following steps:

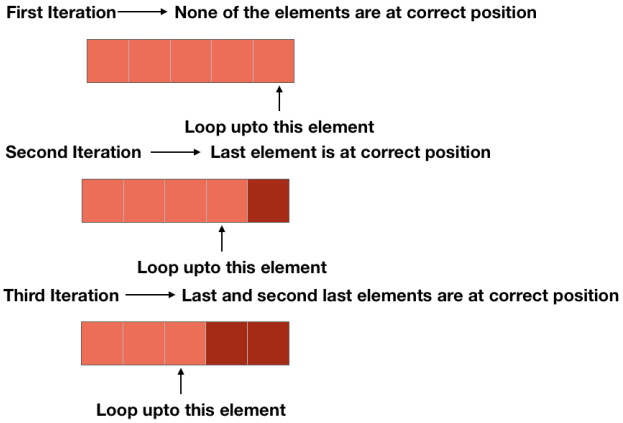
1. Start iterating over the array.
2. Compare the adjacent elements. For example, the first and second, second and third, etc.
3. Swap them if they are not in order.
4. Repeat these steps except for the elements which are placed at their correct positions.

Now, the next part is to write a code for Bubble sort and implement these logics.

Code for Bubble Sort

We have to start by iterating over the array. So, let's use a for loop for that - `for i in 1 to A.length`. Now we have to repeatedly compare the adjacent elements and swap them if they are in the wrong order (see the above picture). Also at the end of the initial for loop, the largest element is at the end of the array, so this repetition should not be done for those elements which are already at correct place. So, we can iterate with a variable 'j' from 1 to A.length-i and then compare the j^{th} and $j+1^{th}$ elements i.e., `for j`

`in 1 to A.length-i -> compare A[j] and A[j+1]`. Here, `A.length-i` will make the iteration not touch the elements which are already at their correct position. For example, in the first iteration, none of the elements are at correct position, so j will go up to `A.length-1` and thus compare `A[A.length-1]` and `A[A.length]` at the last; similarly, at the second iteration, the last element is at the correct position, so j will go up to `A.length-2` and compare `A[A.length-2]` and `A[A.length-1]` and hence leaving the last element i.e., `A[A.length]` and so on.



So, let's write the full code for Bubble sort.

```
Bubble-SORT(A)
  for i in 1 to A.length
    for j in 1 to A.length-i
      if A[j] > A[j+1]
        swap(A[j], A[j+1])
```

The first loop (`for i in 1 to A.length`) is for iterating over the array. Now for every iteration, we are again iterating from the first element of the array to the (A.length-i)th element (`for j in 1 to A.length-i`) and then comparing `A[j]` and `A[j+1]`.

After the comparison, we are swapping `A[j]` and `A[j+1]` if they are in the wrong order.

C **Python** **Java**

```c
#include <stdio.h>

// function to swap values of two variables
void swap(int *a, int *b) {
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}

void bubble_sort(int a[], int size) {
  int i, j;
  for(i=0; i<size; i++) {
    for(j=0; j<size-i-1; j++) {
      if (a[j] > a[j+1]) {
        swap(&a[j], &a[j+1]);
      }
    }
  }
}

int main() {
  int a[] = {4, 8, 1, 3, 10, 9, 2, 11, 5, 6};
  bubble_sort(a, 10);

  //printing array
  int i;
  for(i=0; i<10; i++) {
    printf("%d ",a[i]);
  }
  printf("\n");
  return 0;
}
```

One thing anyone should note here is that even if the array gets sorted in the middle of the process, there is no mechanism to stop the process and the process will continue to occur unnecessarily. So, this is where we can optimize our algorithm.

Optimized Version of Bubble Sort

To check if the array is sorted or not, we can check if any element is getting swapped in the iteration or not. If none of the elements are getting swapped, then it means that the array is sroted and we can stop the process of Bubble Sort.

So, let's use a variable to keep the record of swapping and modify the previous code according to it.

```
Bubble-SORT-OPTIMIZED(A)
  for i in 1 to A.length
    swapped = FALSE
    for j in 1 to A.length-i
      if A[j] > A[j+1]
        swap(A[j], A[j+1])
        swapped = TRUE
    if not(swapped)
      break
```

Here, we are using an extra variable 'swapped' to store whether we are swapping in the middle of the algorithm or not. We are setting the 'swapped' value to true after swapping something - `swapped = TRUE` and at last, we are checking if something was swapped or not in the process - `if not(swapped)`, then we are breaking the loop if nothing was swapped and hence meaning that the loop is already sorted.

C **Python** **Java**

```c
#include <stdio.h>

// function to swap values of two variables
void swap(int *a, int *b) {
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}

void optimized_bubble_sort(int a[], int size) {
  int i, j;
  for(i=0; i<size; i++) {
    int swapped = 0;
    for(j=0; j<size-i-1; j++) {
      if (a[j] > a[j+1]) {
        swap(&a[j], &a[j+1]);
        swapped = 1;
      }
    }
    if(!swapped)
      break;
  }
}

int main() {
  int a[] = {4, 8, 1, 3, 10, 9, 2, 11, 5, 6};
  optimized_bubble_sort(a, 10);

  //printing array
  int i;
  for(i=0; i<10; i++) {
    printf("%d ",a[i]);
  }
  printf("\n");
  return 0;
}
```

Analysis of Bubble sort

Let's first talk about the non-optimized version of the Bubble sort.

Non-Optimized Bubble Sort

Regardless of the input, the two loops and the if statements are going to execute every time. Only the execution of the swap statement will depend upon the input.

So, every time the first for loop (`for i in 1 to A.length`) is going to execute n+1 times (n is the size of the array 'A' and 1 extra time when the condition of the loop will be check and failed) and the second for loop is going to execute (n-i)+1 times for every iteration of the outer loop. For example, it is going to execute (n-1)+1 times for the first iteration of the outer loop (when i is 1), (n-2)+1 times in the second iteration of the outer loop (i is 2) and so on. So, it will run a total of $(n) + (n-1) + \ldots + 1 = \frac{(n)(n+1)}{2}$ times.

Now, the `if` statement is going to run every time the statements of inner loop are going to be executed i.e., $(n-1) + (n-2) + \ldots + 1 = \frac{(n-1)(n)}{2}$ times.

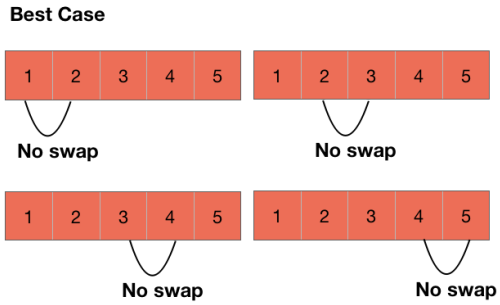
Till now, we know that our running time is going to be quadratic regardless of the input. The swap statement can run a minimum of 0 times when the array is already sorted or a maximum of $\frac{(n-1)(n)}{2}$ times (equal to the number of times `if` is executed). In both cases, the rate of the growth of our function will be quadratic because we already have quadratic terms in our function ($\frac{(n)(n+1)}{2}$ and $\frac{(n-1)(n)}{2}$). So, the running time of this algorithm is $\Theta(n^2)$.

Optimized Bubble Sort

In the worst case, there won't be any difference between the optimized and the non-optimized versions because the worst case will be when the `break` statement never gets executed and this is going to occur when the array is sorted in the reverse order.

Thus, the optimized version has $O(n^2)$ worst case running time.

Now, let's talk about the best case. The best case would be when the outer for loop (`for i in 1 to A.length`) just breaks after its first iteration. And this can occur when there is nothing swapped inside the second loop which means the array is already sorted.



In this case, the outer for loop will run only one time and the inner for loop will run **n** times and then the algorithm will stop. So, the best case running time of this algorithm is going to be a linear one and thus this algorithm has $\Omega(n)$

Thus, we have seen two sorting algorithms till now. Both of these algorithms are coding implementations of how we basically sort in our real life. And this is exactly what we need to develop our algorithms. First, think and solve the problem in our mind. If we are able to do so then implementing that solution in code is not going to be a difficult task after going through all the algorithms in this course.

> “ I do not fear computers. I fear lack of them. ”
>
> - Isaac Asimov

PREV (/course/algorithms-sort-it-out/) (/course/algorithms-count-and-sort/) NEXT

Further Readings

- ➔ Sorting a list using bubble sort in Python (/blog/article/sorting-a-list-using-bubble-sort-in-python/)
- ➔ Sorting an array using bubble sort in C (/blog/article/sorting-an-array-using-bubble-sort-in-c/)
- ➔ Sorting an array using bubble sort in Java (/blog/article/sorting-an-array-using-bubble-sort-in-java/)