Project Report

INSERTION SORT AND BUBBLE SORT WITH TIME COMPLEXITY

Computer Organization & Assembly Language

Group Members:

1. Muhammad Nomir 11330
2. Shehryar Faisal 11346
3. Zunaira Ahmed 11354

Submitted To:

Yumna Shahzad

Date: July 13, 2021

ACKNOWLEDGEMENT:

“This Report Have Been Prepared Through The Help Of Online Consultation Of Different Webpages For Datasheets & Also In This Report We Explain Our Project Through Algorithms And Flowcharts. This Project Report Have Been Created Within The Given Time Spectrum. All Of The Given Requirement Are In This Report With Correct Format.”

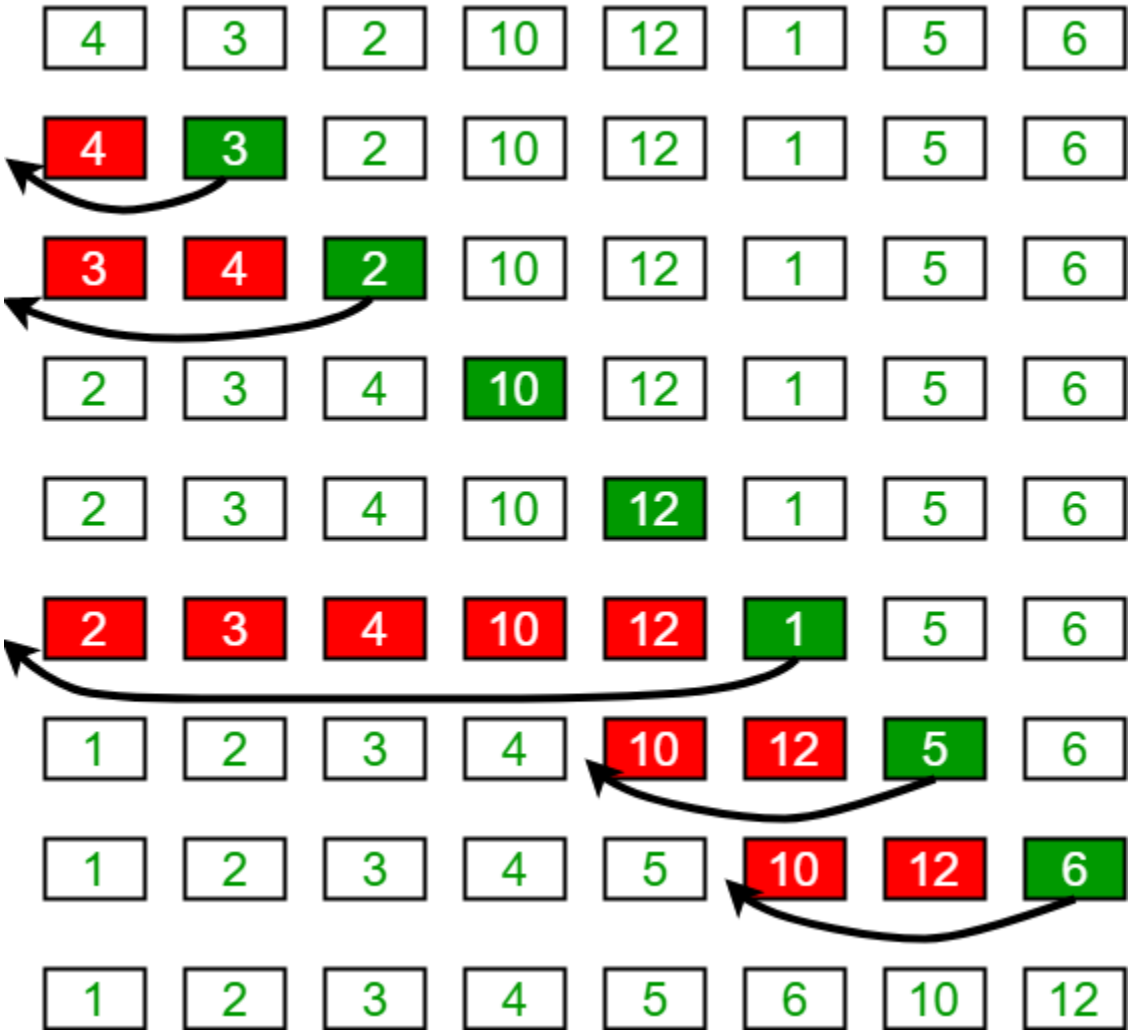
ABSTRACT:

In this whole report you will see all our research and hard-work put into this report. We created a project of insertion sort and bubble sort with their time complexity and explains its functionalities. In this report we tell you about how insertion sort and bubble sort works and also show there algorithm with the help of flowchart and code. In this report you will also be shown that how time complexity works for both the sorting methods.

WORKING:

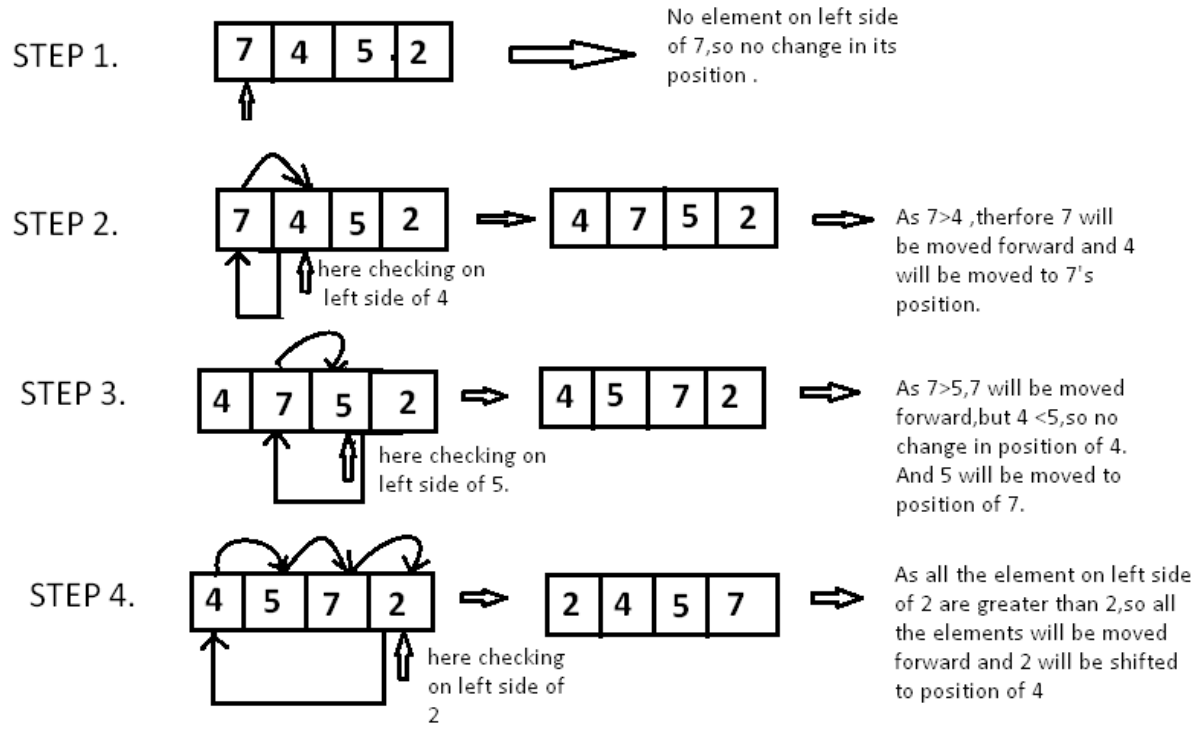
WORKING OF INSERTION SORT:

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.



ALGORITHM:

To sort an array of size n in ascending order:
1: Iterate from arr[1] to arr[n] over the array.
2: Compare the current element (key) to its predecessor.
3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.



INSERTION-SORT(A)	*cost*	*times*
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // Insert $A[j]$ into the sorted sequence $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8 $A[i+1] = key$	c_8	$n-1$

TIME COMPLEXITY:

We denote with n the number of elements to be sorted. For example n = 6.

The two nested loops are an indication that we are dealing with quadratic effort, meaning with time complexity of $O(n^2)$*. This is the case if both the outer and the inner loop count up to a value that increases linearly with the number of elements.

With the outer loop, this is obvious as it counts up to n.

And the inner loop? We'll analyze that in the next three sections.

AVERAGE TIME COMPLEXITY:

This is the array [6, 2, 4, 9, 3, 7] we will be working with.

We shifted one element from the array. If the element to be sorted had already been in the right place, we would not have had to shift anything. This means that we have an average of 0.5 move operations in the first step.

Then we have also shifted one element. But here it could also have been zero or two shifts. On average, it is one shift in this step.

Then we did not need to shift any elements. However, it could have been necessary to shift one, two, or three elements; the average here is 1.5.

Then we have on average two shift operations.
In the last step the average here is 2.5.

So, in total we have on average 0.5 + 1 + 1.5 + 2 + 2.5 = 7.5 shift operations.

We can also calculate this as follows:
6 × 5 × ½ × ½ = 30 × ¼ = 7,5

If we replace 6 with n, we get

n × (n - 1) × ¼

When multiplied, that's:

¼ n^2 - ¼ n

The highest power of n in this term is n^2; the time complexity for shifting is, therefore, $O(n^2)$. So,the average time complexity of Insertion Sort = $O(n^2)$

BEST-CASE TIME COMPLEXITY:

If the elements already appear in sorted order, there is precisely one comparison in the inner loop and no swap operation at all.

With n elements, that is, n-1 steps (since we start with the second element), we thus come to n-1 comparison operations. So, the best-case time complexity of Insertion Sort = O(n)

WORST-CASE TIME COMPLEXITY:

In the worst case, the elements are sorted completely descending at the beginning. In each step, all elements of the sorted sub-array must, therefore, be shifted to the right so that the element to be

sorted which is smaller than all elements already sorted in each step can be placed at the very beginning.
The term from the average case, therefore, changes in that the second dividing by two is omitted:

$6 \times 5 \times ½$

Or:

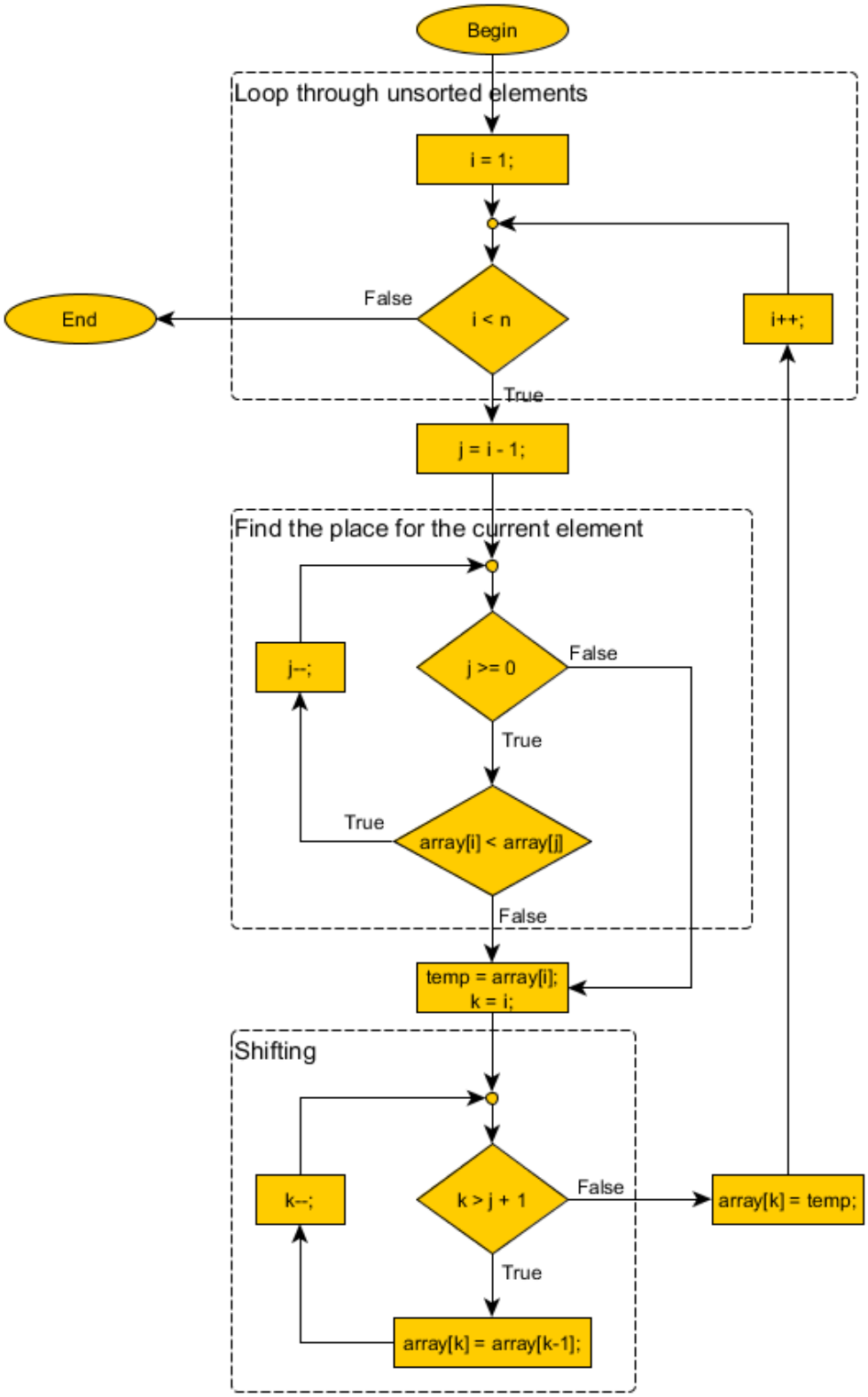
$n \times (n - 1) \times ½$

When we multiply this out, we get:

$½ n^2 - ½ n$

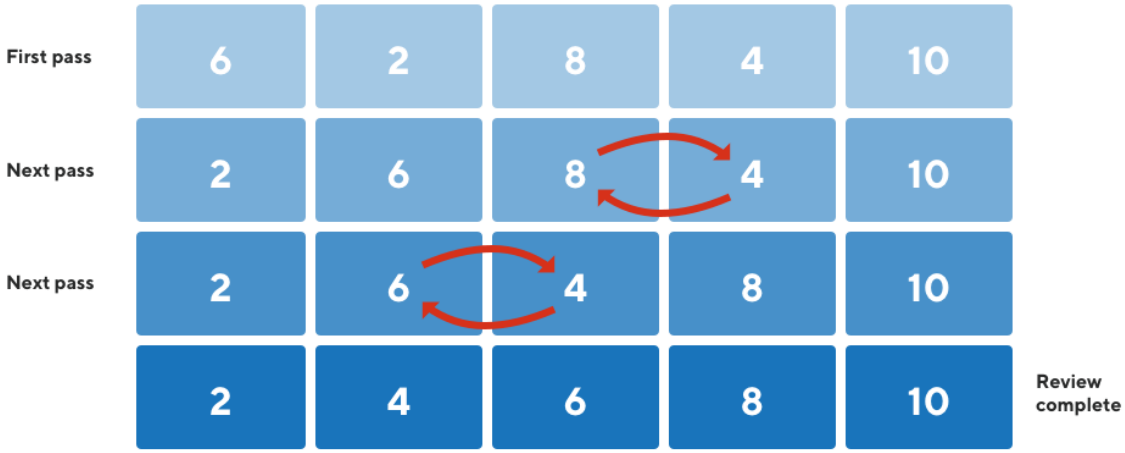
Even if we have only half as many operations as in the average case, nothing changes in terms of time complexity the term still contains n^2. So, the worst-case time complexity of Insertion Sort = $O(n^2)$

FLOWCHART:


Begin
Loop through unsorted elements
i = 1;
False
i < n
End
i++;
True
j = i - 1;
Find the place for the current element
j--;
j >= 0
False
True
True
array[i] < array[j]
False
temp = array[i];
k = i;
Shifting
k--;
k > j + 1
False
array[k] = temp;
True
array[k] = array[k-1];

WORKING OF BUBBLE SORT:

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.



ALGORITHM:

First Pass:

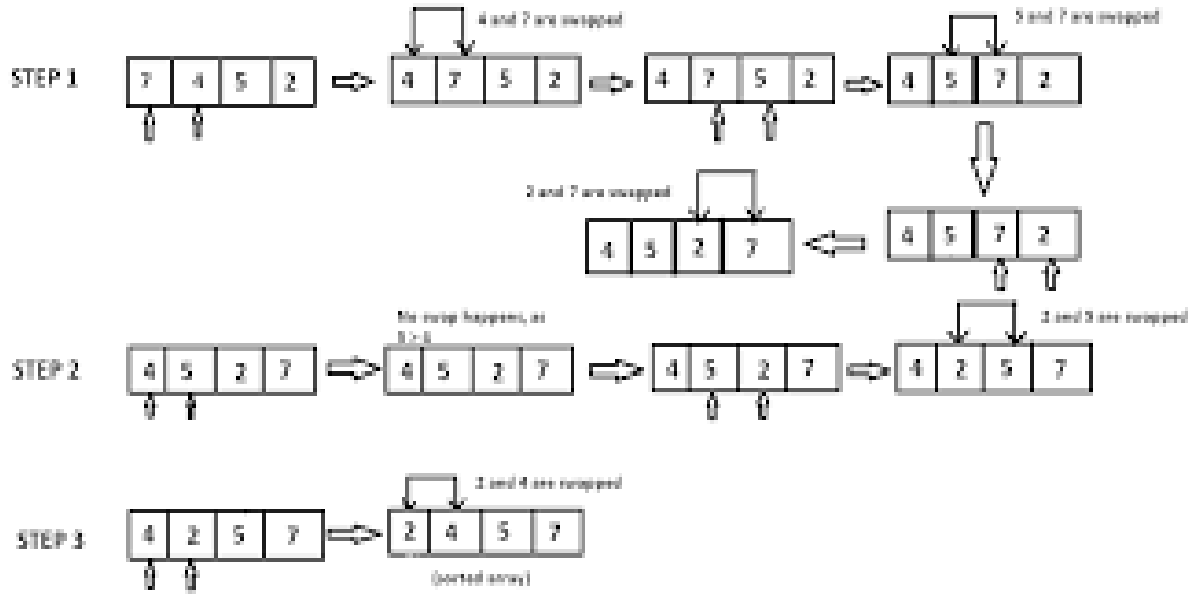
(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.
(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4
(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2
(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:

(**1 4** 2 5 8) –> (**1 4** 2 5 8)
(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2
(1 2 **4 5** 8) –> (1 2 **4 5** 8)
(1 2 4 **5 8**) –> (1 2 4 **5 8**)
Now, the array is already sorted, but our 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**)



Algorithm 1: Bubble sort

Data: Input array *A[]*
Result: Sorted *A[]*
int i, j, k;
N = length(A);
for *j = 1 to N* **do**
  for *i = 0 to N-1* **do**
    if *A[i] > A[i+1]* **then**
      temp = A[i];
      A[i] = A[i+1];
      A[i+1] = temp;
    end
  end
end

TIME COMPLEXITY:

We denote by n the number of elements to be sorted. For example, n = 6. The two nested loops suggest that we are dealing with quadratic time, i.e., a time complexity* of $O(n^2)$. This will be the case if both loops iterate to a value that grows linearly with n. With Bubble Sort, we have to examine best, worse, and average case separately. We will do this in the following subsections.

BEST CASE TIME COMPLEXITY:

Let's start with the most straightforward case: If the numbers are already sorted in ascending order, the algorithm will determine in the first iteration that no number pairs need to be swapped and will then terminate immediately. The algorithm must perform n-1 comparisons. So, the best-case time complexity of Bubble Sort = O(n)

WORST CASE TIME COMPLEXITY:

We will demonstrate the worst case with an example. Let's assume we want to sort the descending array [6, 5, 4, 3, 2, 1] with Bubble Sort.

In the first iteration, the largest element, the 6, moves from far left to far right. We omitted the five single steps (swapping the pairs 6/5, 6/4, 6/3, 6/2, 6/1).

In the second iteration, the second largest element, the 5, is moved from the far left – via four intermediate steps – to the second last position.

In the fourth iteration, the 3 is moved – via two single steps – to its final position.

And finally, the 2 and the 1 are swapped.

So, in total we have 5 + 4 + 3 + 2 + 1 = 15 comparison and exchange operations.

$6 \times 5 \times ½ \;=\; 30 \times ½ \;=\; 15$

If we replace 6 with n, we get:

$n \times (n - 1) \times ½$

When multiplied, that gives us:

$½ (n^2 - n)$

The highest power of n in this term is n^2. So, the worst-case time complexity of Bubble Sort = $O(n^2)$

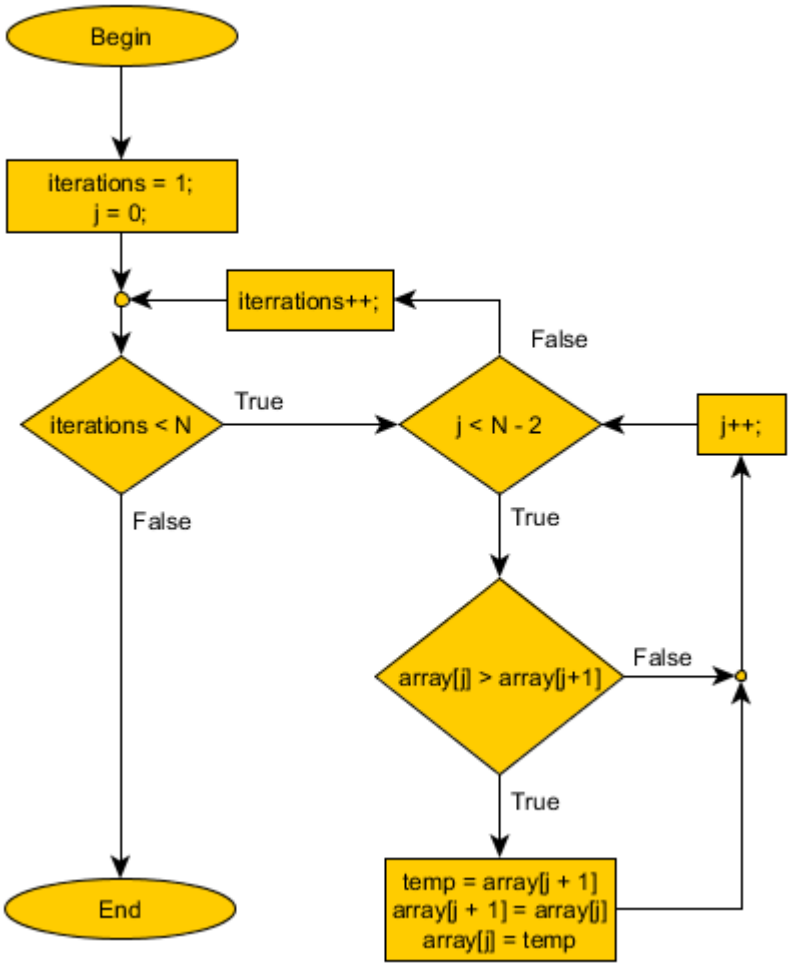
AVERAGE TIME COMPLEXITY:

In the average case, one has about half as many exchange operations as in the worst case since about half of the elements are in the correct position compared to the neighboring element. So, the number of exchange operations = $¼ (n^2 - n)$

It becomes even more complicated with the number of comparison operations, which amounts to:
$½ (n^2 - n \times \ln(n) - (\gamma + \ln(2) - 1) \times n) + O(\sqrt{n})$

In both terms, the highest power of n is again n^2. So, the average time complexity of Bubble Sort case = $O(n^2)$

FLOWCHART:



WORKING OF PROGRAM:

Our program works on the same basis of bubble sort and insertion sort but combined as we are using both of them separately and combined. Our program shows Insertion Sort, Bubble Sort, Both of them combined and also the amount of time they took to complete the sorting on individual basis

FUTURE ENHANCEMENTS:

In terms of future enhancements of the program everything can be improved nothing reaches its perfection and everything can be improved so yes there are future enhancements that could be made. First the program can always be made faster more efficient, less time consuming this is one of the enhancements, Secondly better use of instructions as this program was made with intermediate knowledge and expertise so in the future less and better mnemonics/instructions can be used to make the program better, Thirdly a better way to show the time complexity with in the program so that it becomes more user friendly and easier to understand even after these improvements it won't be a perfect program whatsoever because perfection isn't achieved with in few steps and most of the times there is no perfection as everything can be improved with proper expertise and knowledge.

In terms of future enhancements of a sorting algorithm it is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input unsorted array. Some sorting algorithms are stable by nature like Insertion sort, Merge Sort, Bubble Sort, etc. But there were some deficiencies in earlier work related to sorting algorithms. By going through all the experimental results and their analysis, it is concluded that the proposed algorithm is efficient. In all existing algorithms, first complete list is entered, then the list is processed for sorting, but in case of proposed approach, the list is sorted simultaneously. The proposed sorting technique saves the time for traversing the list after entering all the elements, as it sorts all elements before entering any new.