DAA Bubble Sort

Bubble Sort, also known as Exchange Sort, is a simple sorting algorithm. It works by repeatedly stepping throughout the list to be sorted, comparing two items at a time and swapping them if they are in the wrong order. The pass through the list is duplicated until no swaps are desired, which means the list is sorted.

This is the easiest method among all sorting algorithms.

Algorithm

Step 1 ➤ Initialization

1. set 1 ← n, p ← 1

Step 2 ➤ loop,

1. Repeat through step 4 **while** (p ≤ n-1)
2. set E ← 0 ➤ Initializing exchange variable.

Step 3 ➤ comparison, loop.

1. Repeat **for** i ← 1, 1, ... l-1.
2. **if** (A [i] > A [i + 1]) then
3. set A [i] ↔ A [i + 1] ➤ Exchanging values.
4. Set E ← E + 1

Step 4 ➤ Finish, or reduce the size.

1. **if** (E = 0) then
2. exit
3. **else**
4. set l ← l - 1.

How Bubble Sort Works

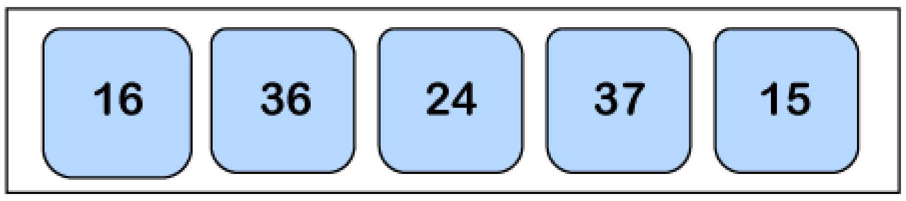
1. The bubble sort starts with the very first index and makes it a bubble element. Then it compares the bubble element, which is currently our first index element, with the next element. If the bubble element is greater and the second element is smaller, then both of them will swap.
After swapping, the second element will become the bubble element. Now we will compare the second element with the third as we did in the earlier step and swap them if required. The same process is followed until the last element.
2. We will follow the same process for the rest of the iterations. After each of the iteration, we will notice that the largest element present in the unsorted array has reached the last index.

For each iteration, the bubble sort will compare up to the last unsorted element.

Once all the elements get sorted in the ascending order, the algorithm will get terminated.

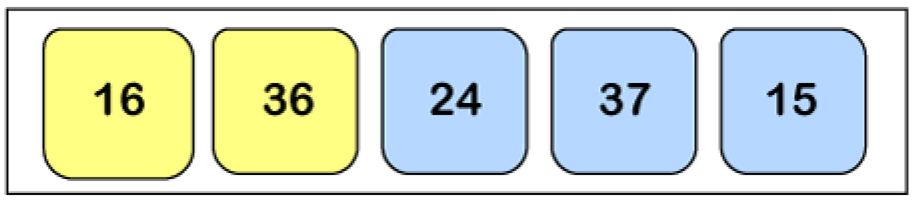
Consider the following example of an unsorted array that we will sort with the help of the Bubble Sort algorithm.

Initially,



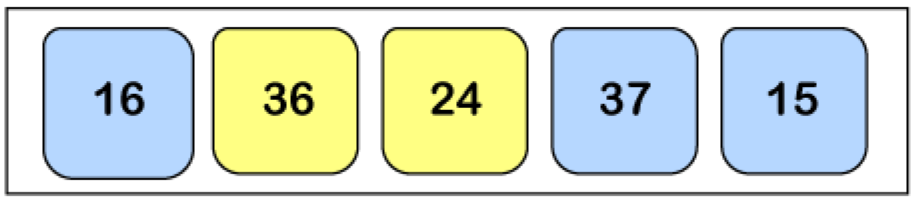
Pass 1:

- **Compare a_0 and a_1**

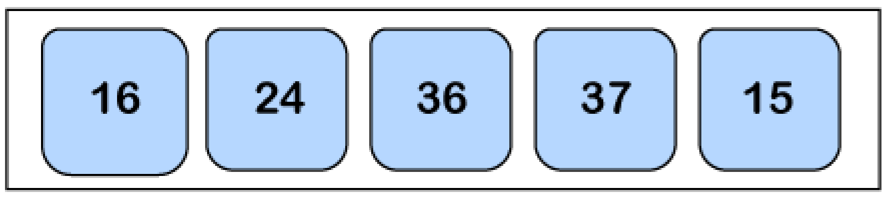


As $a_0 < a_1$ so the array will remain as it is.

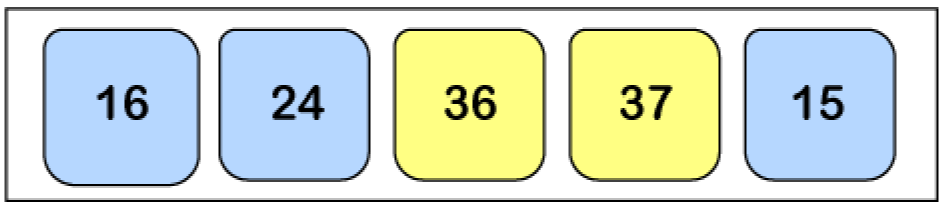
- **Compare a_1 and a_2**



Now $a_1 > a_2$, so we will swap both of them.

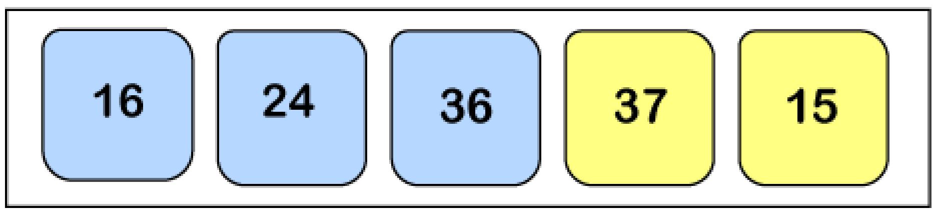


- **Compare a_2 and a_3**

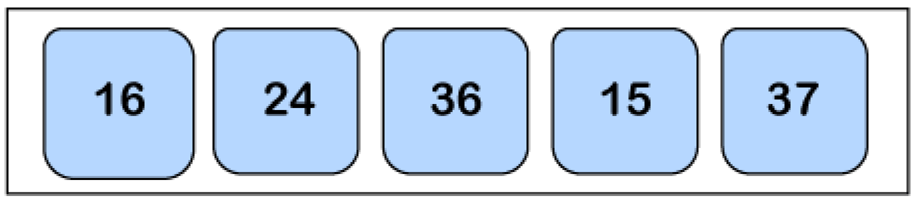


As $a_2 < a_3$ so the array will remain as it is.

- **Compare a_3 and a_4**

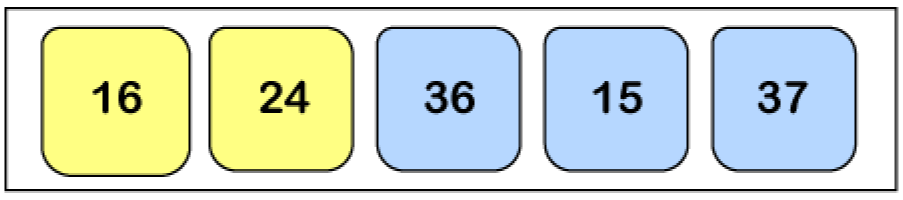


Here $a_3 > a_4$, so we will again swap both of them.



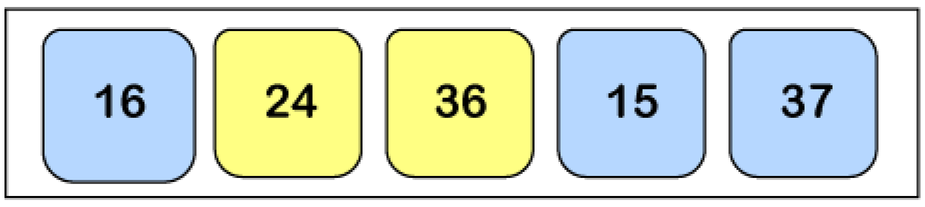
Pass 2:

- **Compare a_0 and a_1**



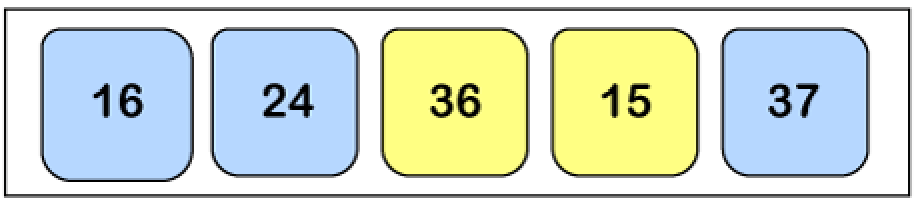
As $a_0 < a_1$ so the array will remain as it is.

- **Compare a_1 and a_2**

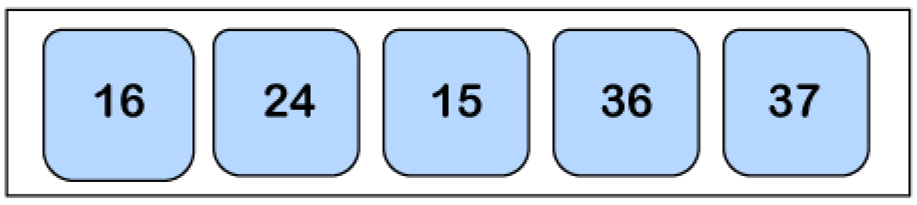


Here $a_1 < a_2$, so the array will remain as it is.

- **Compare a_2 and a_3**

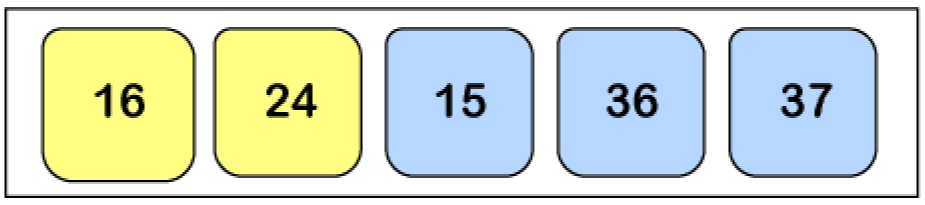


In this case, $a_2 > a_3$, so both of them will get swapped.



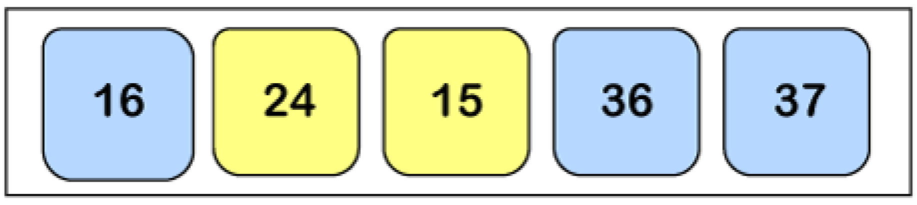
Pass 3:

- **Compare a_0 and a_1**

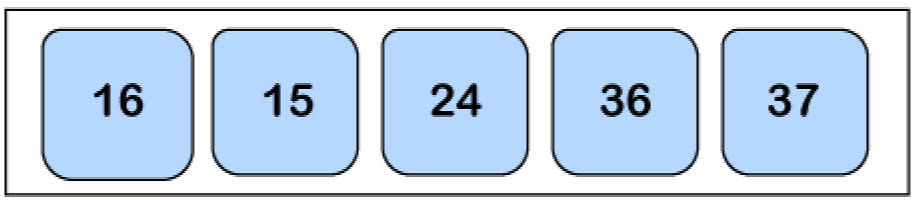


As $a_0 < a_1$ so the array will remain as it is.

- **Compare a_1 and a_2**

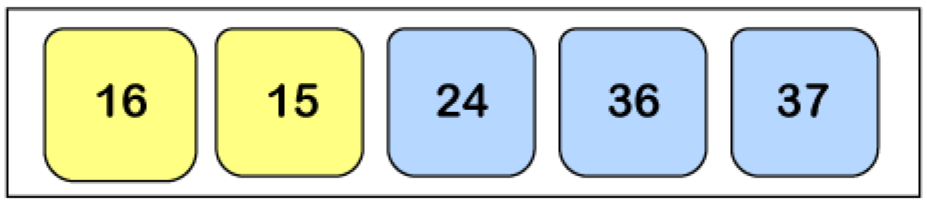


Now $a_1 > a_2$, so both of them will get swapped.

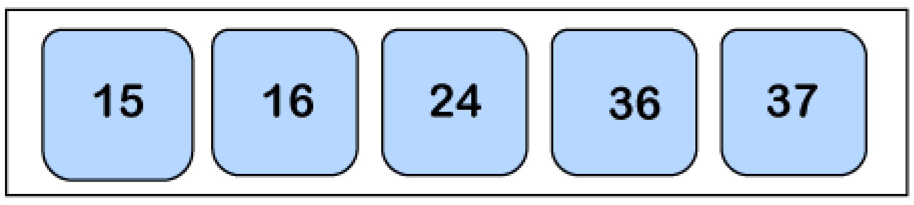


Pass 4:

- **Compare a_0 and a_1**



Here $a_0 > a_1$, so we will swap both of them.



Hence the array is sorted as no more swapping is required.

Complexity Analysis of Bubble Sort

Input: Given n input elements.

Output: Number of steps incurred to sort a list.

Logic: If we are given n elements, then in the first pass, it will do n-1 comparisons; in the second pass, it will do n-2; in the third pass, it will do n-3 and so on. Thus, the total number of comparisons can be found by;

Output;

(n-1) + (n-2) + (n-3) + (n-4) + ... + 1

$$Sum = \frac{n(n-1)}{2}$$

i.e., $O(n^2)$

Therefore, the bubble sort algorithm encompasses a time complexity of **$O(n^2)$** and a space complexity of **O(1)** because it necessitates some extra memory space for temp variable for swapping.

Time Complexities:

- **Best Case Complexity**: The bubble sort algorithm has a best-case time complexity of **O(n)** for the already sorted array.
- **Average Case Complexity**: The average-case time complexity for the bubble sort algorithm is **$O(n^2)$**, which happens when 2 or more elements are in jumbled, i.e., neither in the ascending order nor in the descending order.
- **Worst Case Complexity**: The worst-case time complexity is also **$O(n^2)$**, which occurs when we sort the descending order of an array into the ascending order.

Advantages of Bubble Sort

1. Easily understandable.
2. Does not necessitates any extra memory.
3. The code can be written easily for this algorithm.
4. Minimal space requirement than that of other sorting algorithms.

Disadvantages of Bubble Sort

1. It does not work well when we have large unsorted lists, and it necessitates more resources that end up taking so much of time.
2. It is only meant for academic purposes, not for practical implementations.
3. It involves the n^2 order of steps to sort an algorithm.

Selection Sort

The selection sort enhances the bubble sort by making only a single swap for each pass through the rundown. In order to do this, a selection sort searches for the biggest value as it makes a pass and, after finishing the pass, places it in the best possible area. Similarly, as with a bubble sort, after the first pass, the biggest item is in the right place. After the second pass, the following biggest is set up. This procedure proceeds and requires n-1 goes to sort n item since the last item must be set up after the (n-1) th pass.

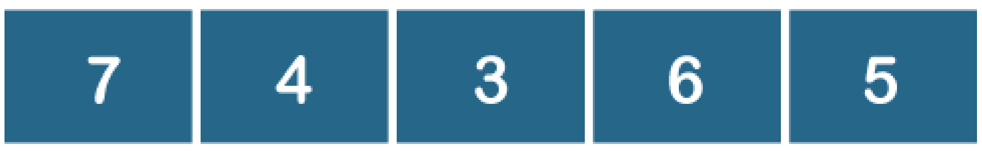
ALGORITHM: SELECTION SORT (A)

```
1. 1. k ← length [A]
2. 2. for j ←1 to n-1
3. 3. smallest ← j
4. 4. for I ← j + 1 to k
5. 5. if A [i] < A [ smallest]
6. 6. then smallest ← i
7. 7. exchange (A [j], A [smallest])
```

How Selection Sort works

1. In the selection sort, first of all, we set the initial element as a **minimum**.
2. Now we will compare the minimum with the second element. If the second element turns out to be smaller than the minimum, we will swap them, followed by assigning to a minimum to the third element.
3. Else if the second element is greater than the minimum, which is our first element, then we will do nothing and move on to the third element and then compare it with the minimum.
   We will repeat this process until we reach the last element.
4. After the completion of each iteration, we will notice that our minimum has reached the start of the unsorted list.
5. For each iteration, we will start the indexing from the first element of the unsorted list. We will repeat the Steps from 1 to 4 until the list gets sorted or all the elements get correctly positioned.
   Consider the following example of an unsorted array that we will sort with the help of the Selection Sort algorithm.

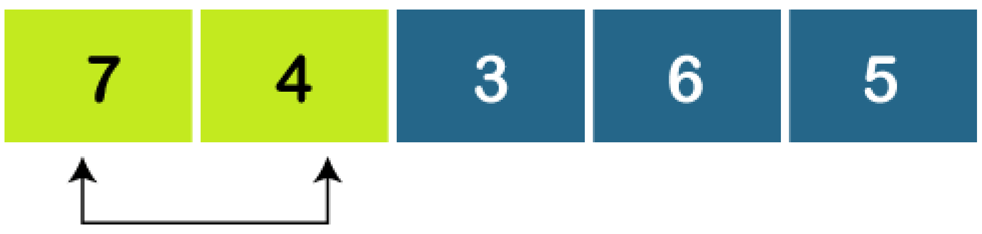
   A [] = (7, 4, 3, 6, 5).
   A [] =



1st Iteration:

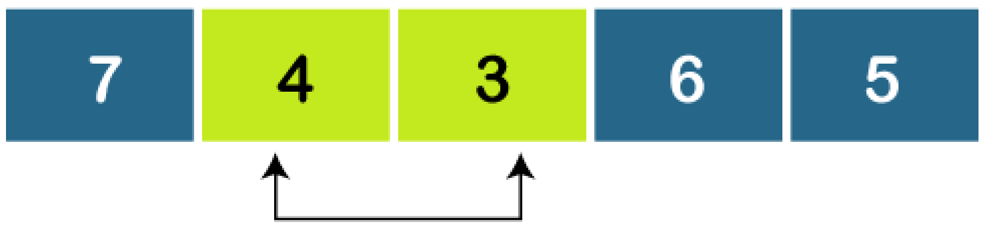
Set minimum = 7

- Compare a_0 and a_1



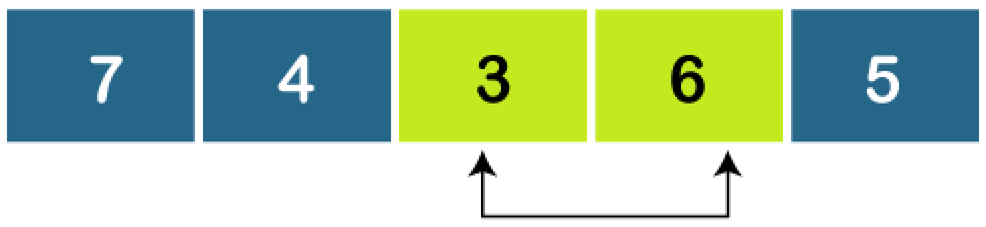
As, $a_0 > a_1$, set minimum = 4.

- Compare a_1 and a_2



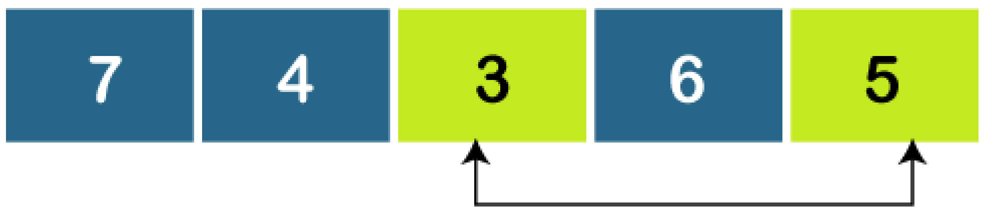
As, $a_1 > a_2$, set minimum = 3.

- Compare a_2 and a_3



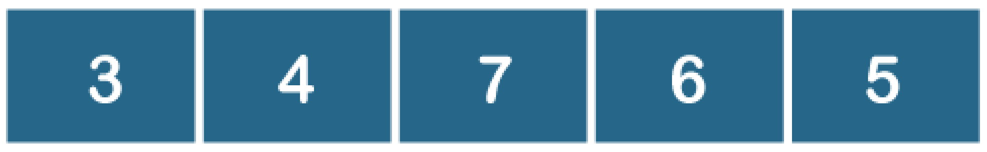
As, $a_2 < a_3$, set minimum= 3.

- Compare a_2 and a_4



As, $a_2 < a_4$, set minimum =3.

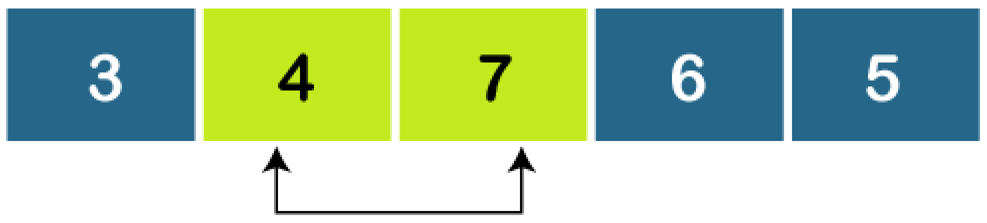
Since 3 is the smallest element, so we will swap a_0 and a_2.



2nd Iteration:

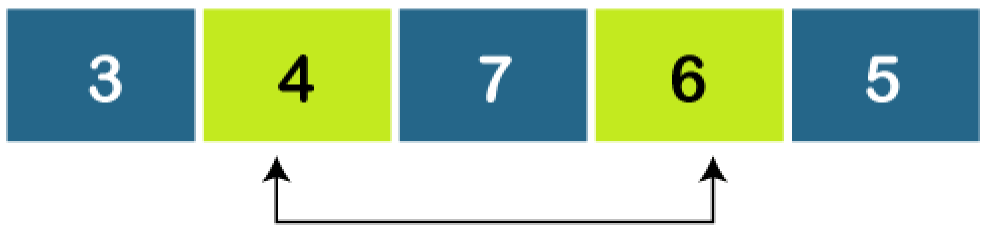
Set minimum = 4

- Compare a_1 and a_2



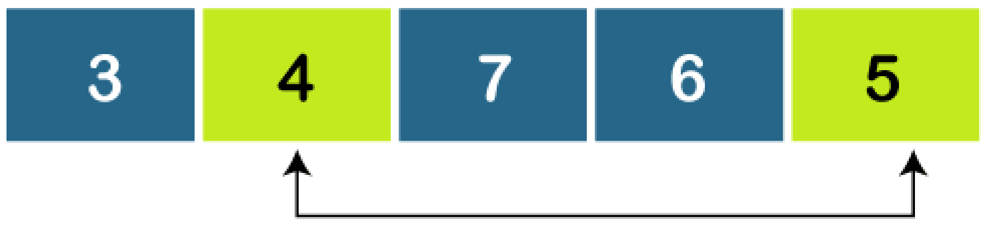
As, $a_1 < a_2$, set minimum = 4.

- Compare a_1 and a_3



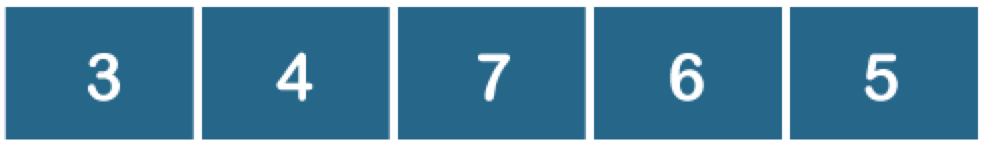
As, A[1] < A[3], set minimum = 4.

- Compare a_1 and a_4



Again, $a_1 < a_4$, set minimum = 4.

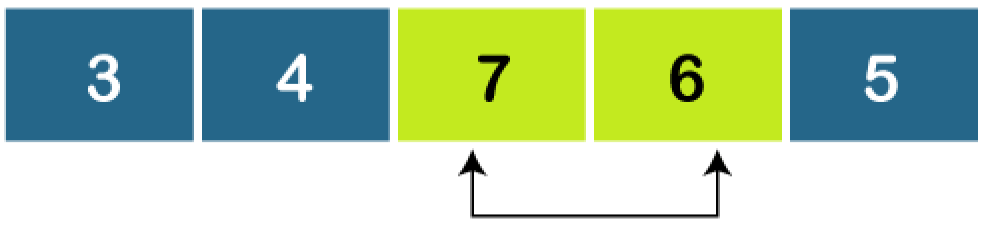
Since the minimum is already placed in the correct position, so there will be no swapping.



3rd Iteration:

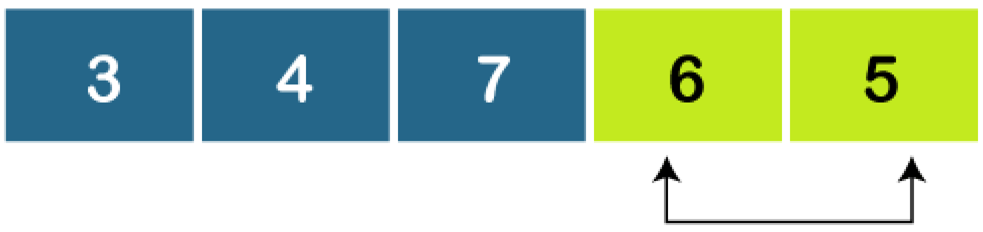
Set minimum = 7

- Compare a_2 and a_3



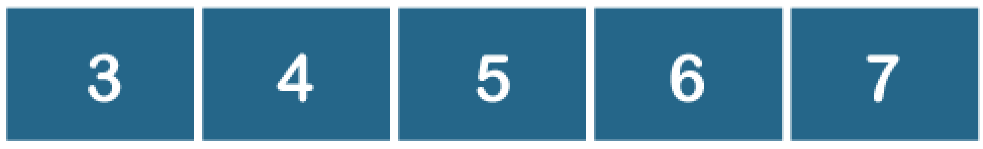
As, $a_2 > a_3$, set minimum = 6.

- Compare a_3 and a_4



As, $a_3 > a_4$, set minimum = 5.

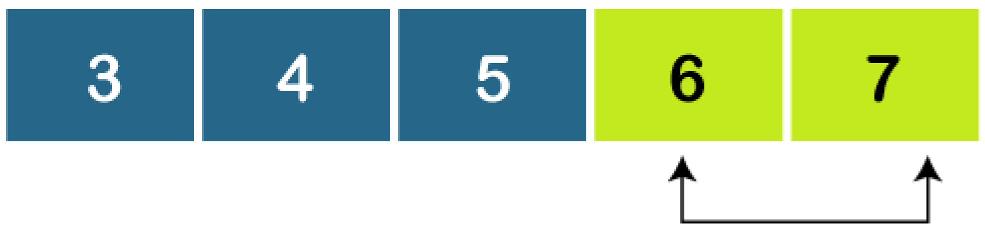
Since 5 is the smallest element among the leftover unsorted elements, so we will swap 7 and 5.



4th Iteration:

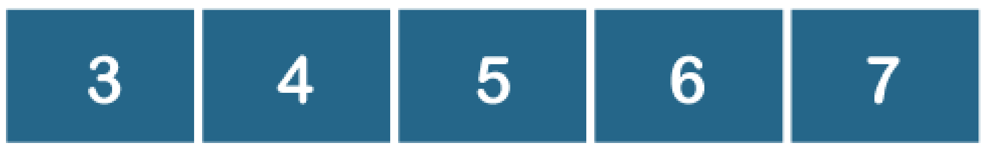
Set minimum = 6

- Compare a_3 and a_4



As $a_3 < a_4$, set minimum = 6.

Since the minimum is already placed in the correct position, so there will be no swapping.



Complexity Analysis of Selection Sort

Input: Given **n** input elements.

Output: Number of steps incurred to sort a list.

Logic: If we are given n elements, then in the first pass, it will do **n-1** comparisons; in the second pass, it will do **n-2**; in the third pass, it will do **n-3** and so on. Thus, the total number of comparisons can be found by;

Output;

(n-1) + (n-2) + (n-3) + (n-4) + ... + 1

$$Sum = \frac{n(n-1)}{2}$$

i.e., $O(n^2)$

Therefore, the selection sort algorithm encompasses a time complexity of **$O(n^2)$** and a space complexity of **O(1)** because it necessitates some extra memory space for temp variable for swapping.

Time Complexities:

- **Best Case Complexity:** The selection sort algorithm has a best-case time complexity of **$O(n^2)$** for the already sorted array.
- **Average Case Complexity:** The average-case time complexity for the selection sort algorithm is **$O(n^2)$**, in which the existing elements are in jumbled ordered, i.e., neither in the ascending order nor in the descending order.
- **Worst Case Complexity:** The worst-case time complexity is also **$O(n^2)$**, which occurs when we sort the descending order of an array into the ascending order.

In the selection sort algorithm, the time complexity is **$O(n^2)$** in all three cases. This is because, in each step, we are required to find **minimum** elements so that it can be placed in the correct position. Once we trace the complete array, we will get our minimum element.

Insertion Sort

Insertion sort is one of the simplest sorting algorithms for the reason that it sorts a single element at a particular instance. It is not the best sorting algorithm in terms of performance, but it's slightly more efficient than selection sort and bubble sort in practical scenarios. It is an intuitive sorting technique.

Let's consider the example of cards to have a better understanding of the logic behind the insertion sort.

Suppose we have a set of cards in our hand, such that we want to arrange these cards in ascending order. To sort these cards, we have a number of intuitive ways.

One such thing we can do is initially we can hold all of the cards in our left hand, and we can start taking cards one after other from the left hand, followed by building a sorted arrangement in the right hand.

Assuming the first card to be already sorted, we will select the next unsorted card. If the unsorted card is found to be greater than the selected card, we will simply place it on the right side, else to the left side. At any stage during this whole process, the left hand will be unsorted, and the right hand will be sorted.

In the same way, we will sort the rest of the unsorted cards by placing them in the correct position. At each iteration, the insertion algorithm places an unsorted element at its right place.

ALGORITHM: INSERTION SORT (A)

```
1.  1. for j = 2 to A.length
2.  2.  key = A[j]
3.  3.  // Insert A[j] into the sorted sequence A[1.. j - 1]
4.  4.  i = j - 1
5.  5.  while i > 0 and A[i] > key
6.  6.      A[i + 1] = A[i]
7.  7.      ii = i -1
8.  8.  A[i + 1] = key
```

How Insertion Sort Works

1. We will start by assuming the very first element of the array is already sorted. Inside the **key**, we will store the second element.

Next, we will compare our first element with the **key**, such that if **the key** is found to be smaller than the first element, we will interchange their indexes or place the key at the first index. After doing this, we will notice that the first two elements are sorted.

2. Now, we will move on to the third element and compare it with the left-hand side elements. If it is the smallest element, then we will place the third element at the first index.

Else if it is greater than the first element and smaller than the second element, then we will interchange its position with the third element and place it after the first element. After doing this, we will have our first three elements in a sorted manner.

3. Similarly, we will sort the rest of the elements and place them in their correct position.

Consider the following example of an unsorted array that we will sort with the help of the Insertion Sort algorithm.

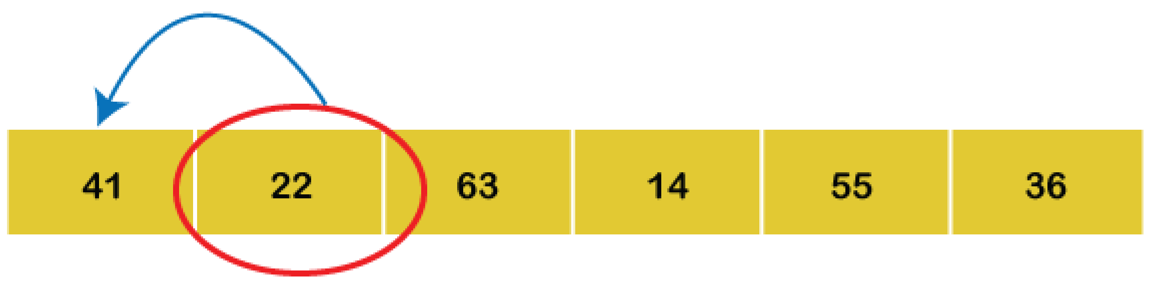
A = (41, 22, 63, 14, 55, 36)

Initially,

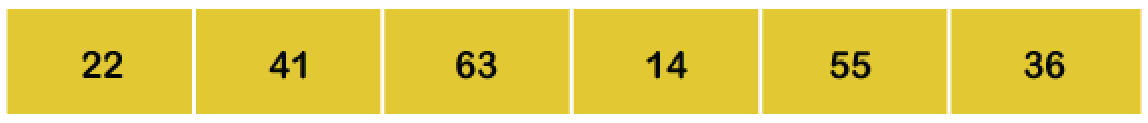
1st Iteration:

Set key = 22

Compare a1 with a0



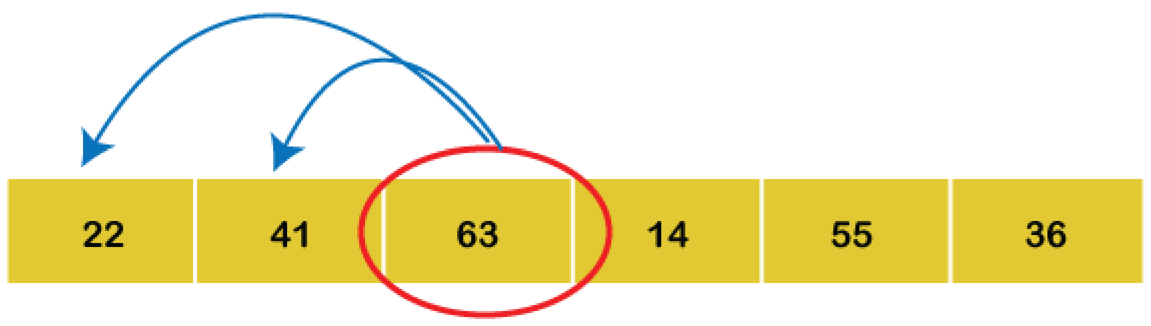
Since a0 > a1, swap both of them.



2nd Iteration:

Set key = 63

Compare a2 with a1 and a0



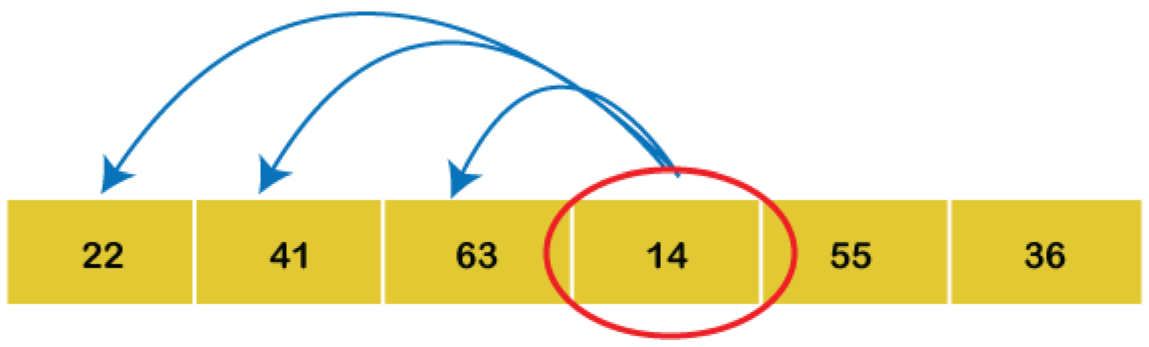
Since a2 > a1 > a0, keep the array as it is.

22	41	63	14	55	36

3rd Iteration:

Set key = 14

Compare a3 with a2, a1 and a0



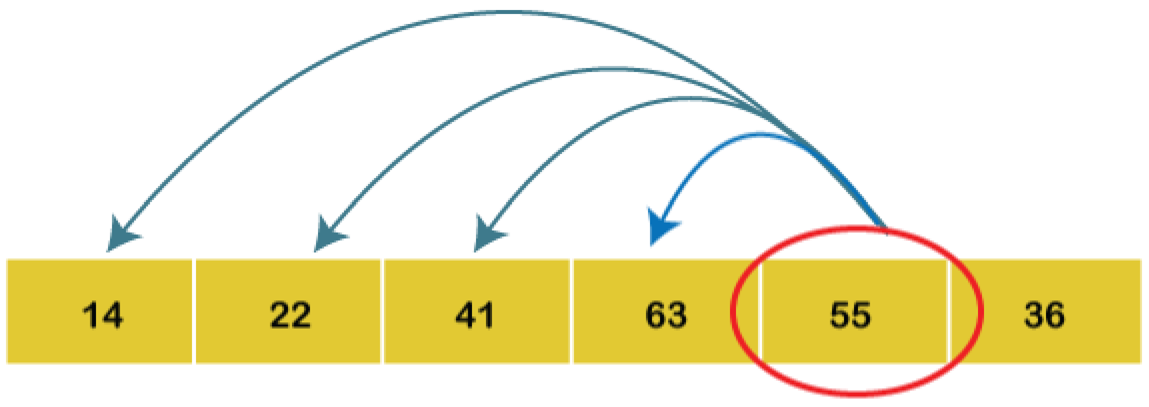
Since a3 is the smallest among all the elements on the left-hand side, place a3 at the beginning of the array.

14	22	41	63	55	36

4th Iteration:

Set key = 55

Compare a4 with a3, a2, a1 and a0.



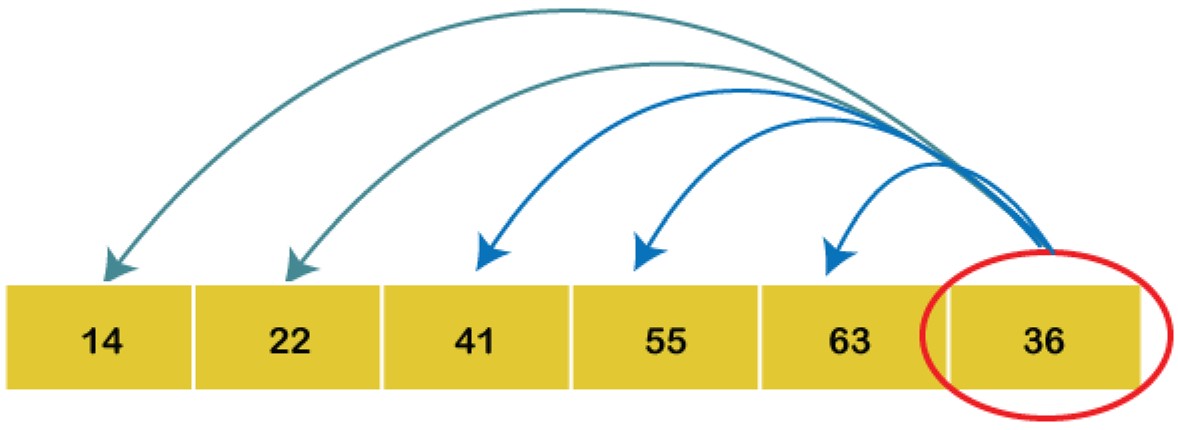
As a4 < a3, swap both of them.

14	22	41	55	63	36

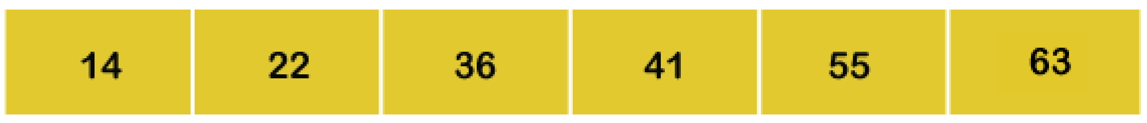
5th Iteration:

Set key = 36

Compare a5 with a4, a3, a2, a1 and a0.



Since a5 < a2, so we will place the elements in their correct positions.



Hence the array is arranged in ascending order, so no more swapping is required.

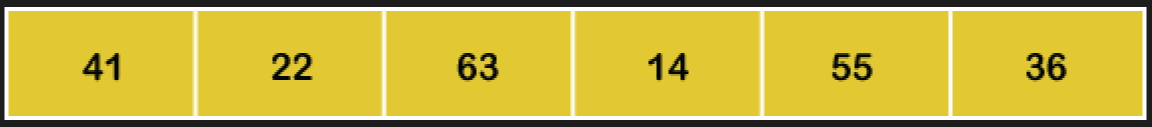
Complexity Analysis of Insertion Sort

Input: Given n input elements.

Output: Number of steps incurred to sort a list.

Logic: If we are given **n** elements, then in the first pass, it will make **n-1** comparisons; in the second pass, it will do **n-2**; in the third pass, it will do **n-3** and so on. Thus, the total number of comparisons can be found by;

```
Output;
(n-1) + (n-2) + (n-3) + (n-4) + ...... + 1
```



```
Sum=
i.e., O(n²)
```

Therefore, the insertion sort algorithm encompasses a time complexity of **O(n^2)** and a space complexity of **O(1)** because it necessitates some extra memory space for a **key** variable to perform swaps.

Time Complexities:

- **Best Case Complexity:** The insertion sort algorithm has a best-case time complexity of **O(n)** for the already sorted array because here, only the outer loop is running n times, and the inner loop is kept still.
- **Average Case Complexity:** The average-case time complexity for the insertion sort algorithm is **O(n^2)**, which is incurred when the existing elements are in jumbled order, i.e., neither in the ascending order nor in the descending order.
- **Worst Case Complexity:** The worst-case time complexity is also **O(n^2)**, which occurs when we sort the ascending order of an array into the descending order.
  In this algorithm, every individual element is compared with the rest of the elements, due to which n-1 comparisons are made for every n^{th} element.

The insertion sort algorithm is highly recommended, especially when a few elements are left for sorting or in case the array encompasses few elements.

Space Complexity

The insertion sort encompasses a space complexity of **O(1)** due to the usage of an extra variable **key**.

Insertion Sort Applications

The insertion sort algorithm is used in the following cases:

- When the array contains only a few elements.

- When there exist few elements to sort.

Advantages of Insertion Sort

1. It is simple to implement.
2. It is efficient on small datasets.
3. It is stable (does not change the relative order of elements with equal keys)
4. It is in-place (only requires a constant amount O (1) of extra memory space).
5. It is an online algorithm, which can sort a list when it is received.