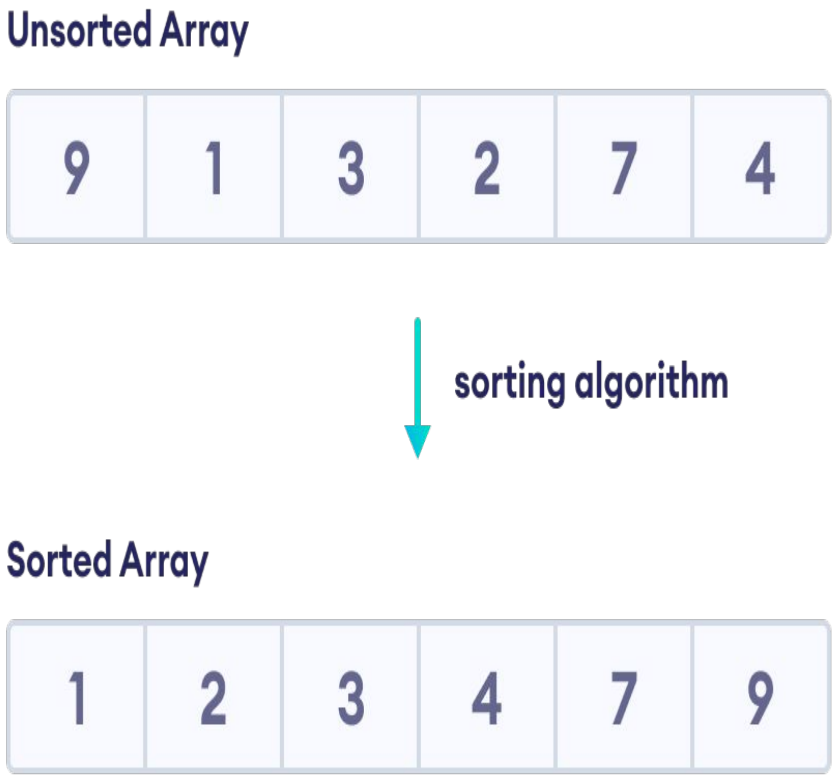
Sorting Basics



Lecture Flow

1) Pre-requisites
2) Problem Definitions and Applications
3) Different approaches
4) Bubble sort
5) Selection sort
6) Insertion sort
7) Counting sort
8) Practice questions
9) Resources
10) Quote of the day

A2SV
Africa To Silicon Valley

Pre-requisites

- Basic python
- Asymptotic Analysis
- Arrays

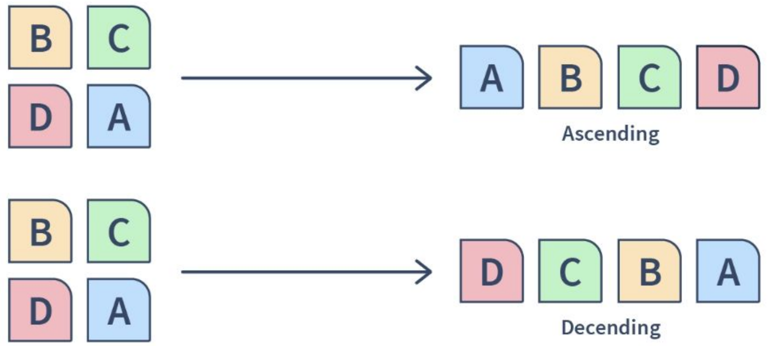
Problem Definition

Real Life Example

Playing Cards

What is Sorting?

- Arranging a set of data in some logical order.
- **Increasing** or **Decreasing** manner.
- Helps finding **largest**, **smallest**, **median** and **nth** value, or **group** items by quality.

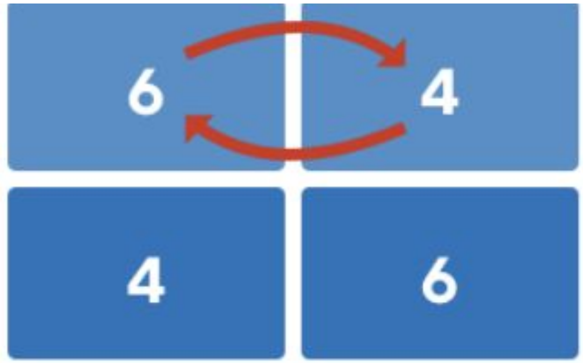


Sorting by Element Comparison

Sorting by Element Comparison

A data item is **compared** with other items in the list of items in order to find its place in the sorted list.

Ex: Bubble, Selection, Insertion …



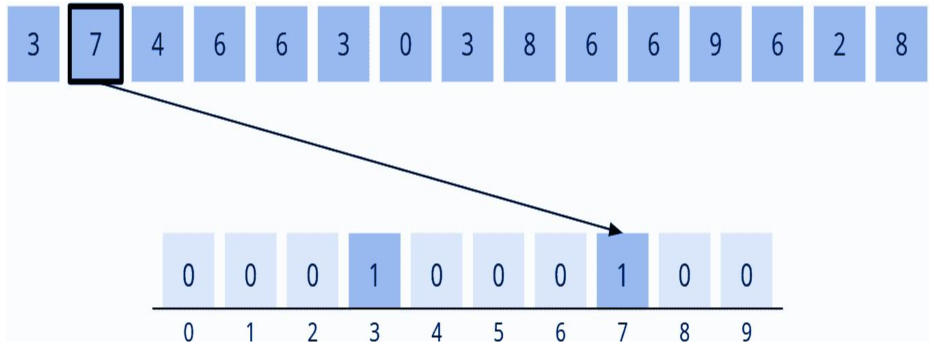
Q: How else can we sort?

Sorting by Distribution

Sorting by Distribution

All items under sorting are distributed over a storage space and then grouped together to get the sorted list.

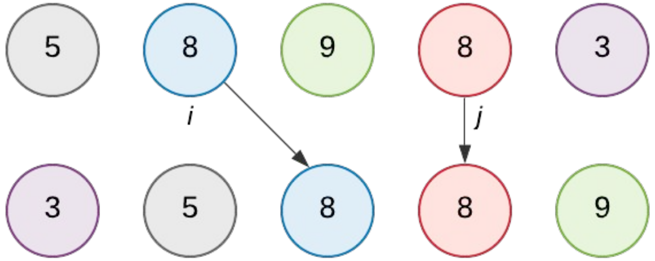
Ex: Counting Sort, Bucket Sort …



Stable vs Unstable Sorting

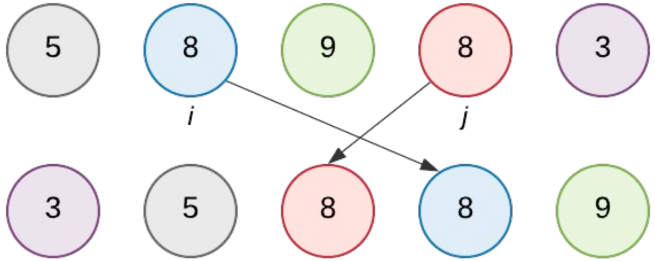
Stable Sorting

Maintains the **original order** of similar items after sorting.

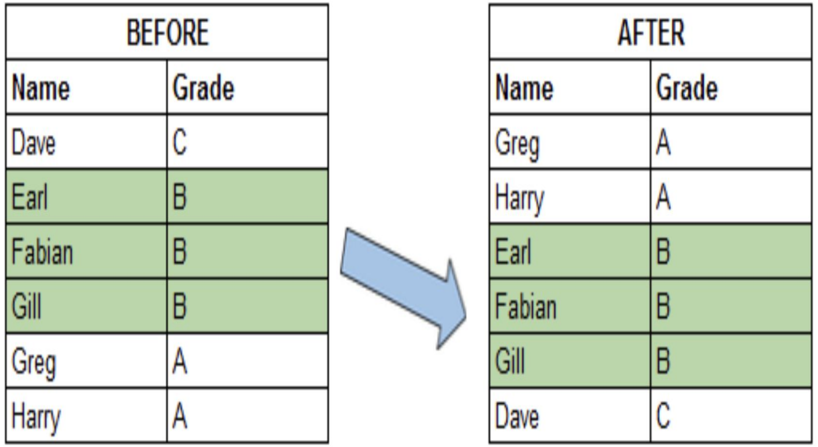


Unstable Sorting

Does **not** preserve the initial arrangement of exact items after sorting.



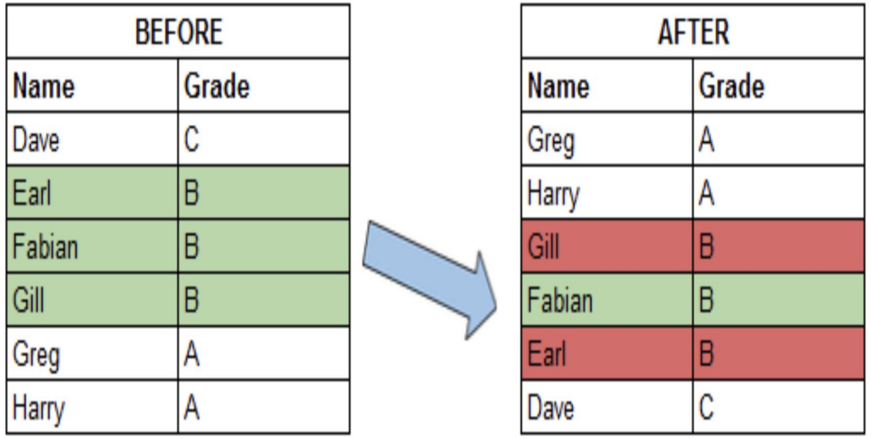
Stable Sorting Vs Unstable Sorting



BEFORE	
Name	Grade
Dave	C
Earl	B
Fabian	B
Gill	B
Greg	A
Harry	A

AFTER	
Name	Grade
Greg	A
Harry	A
Earl	B
Fabian	B
Gill	B
Dave	C

Image showing the effect of stable sorting



BEFORE	
Name	Grade
Dave	C
Earl	B
Fabian	B
Gill	B
Greg	A
Harry	A

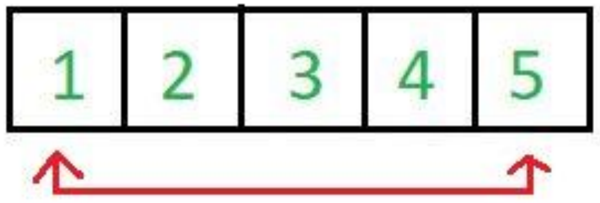
AFTER	
Name	Grade
Greg	A
Harry	A
Gill	B
Fabian	B
Earl	B
Dave	C

Image showing the effect of unstable sorting

In-Place vs Out-of-Place

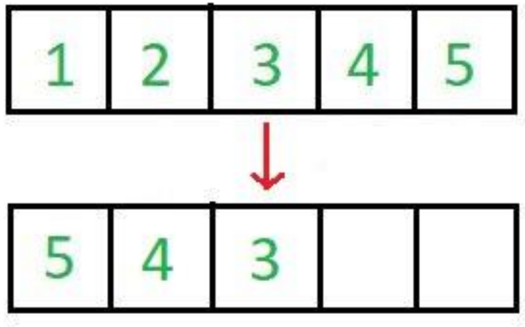
In-Place

Uses constant space by modifying the order of the elements within the list.



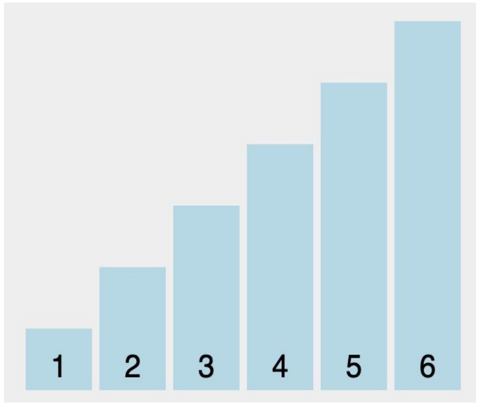
Out-of-Place

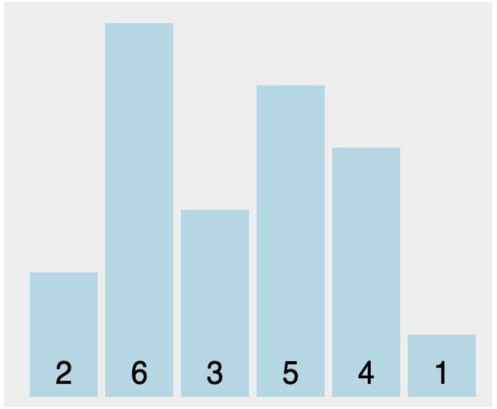
Uses extra space to modify order of elements.

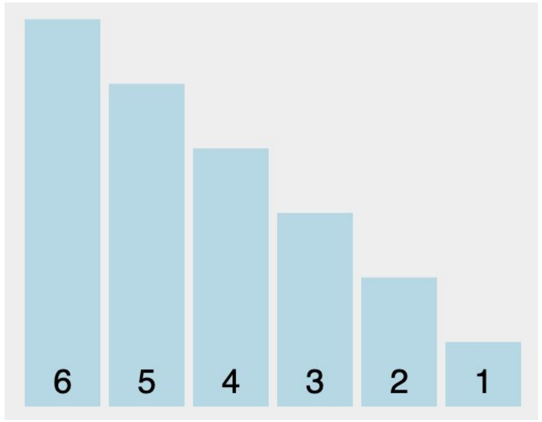


Efficiency of Sorting Algorithm

- The complexity of a sorting algorithm measures the running time of a function in which **n** number of items are to be sorted.
- Various sorting algorithms are analyzed in the cases like - **Best case, Worst case** or **Average case.**

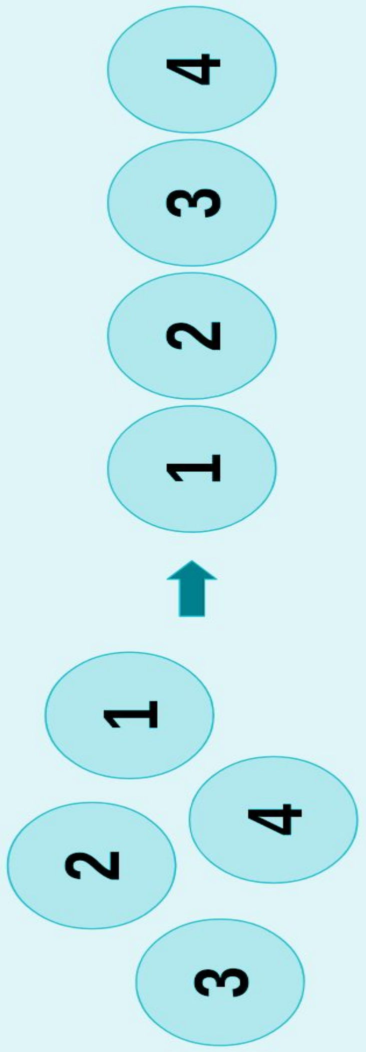




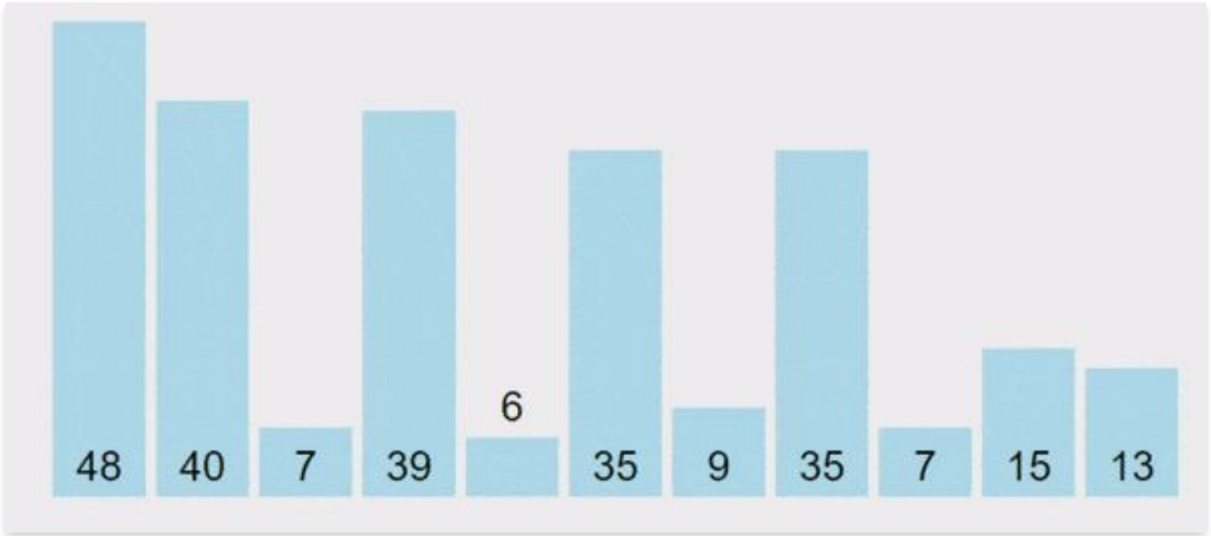


Comparison Sorting

1. Bubble Sort



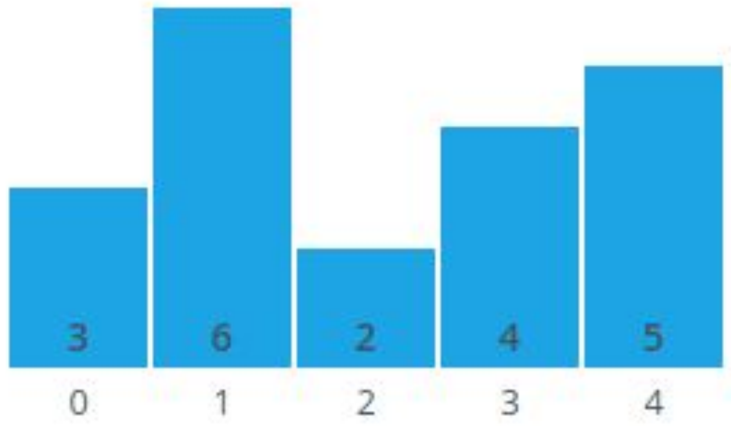
Bubble Sort



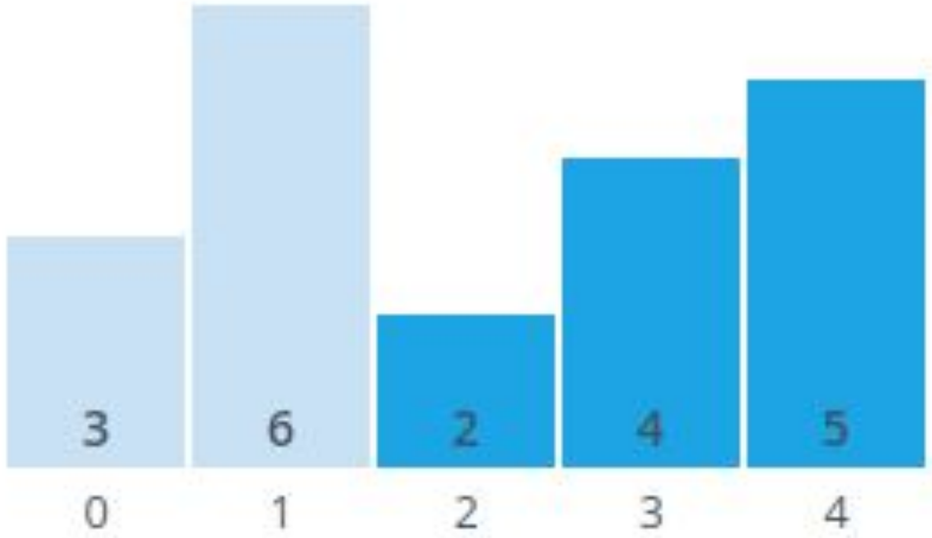
Bubble Sort works by repeatedly **swapping** the adjacent elements if they are in the **wrong order**. Keep doing this until sorted.

Bubble Sort Simulation

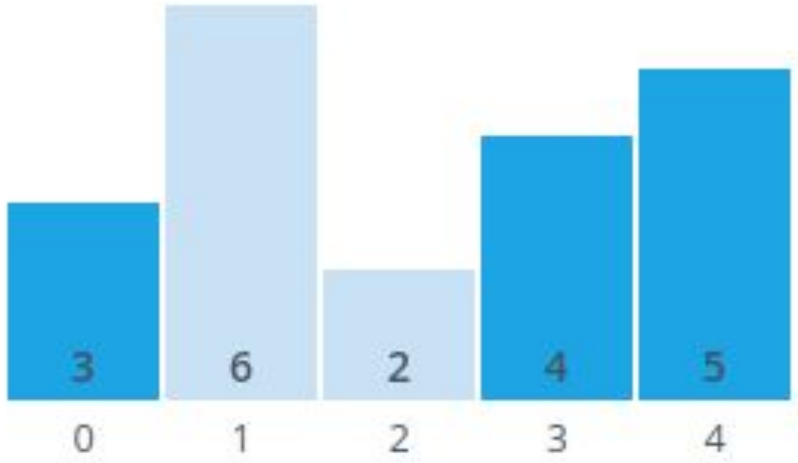
For each pass, we will move left to right swapping adjacent elements as needed. Each pass moves the next largest element into its final position (these will be shown in green).

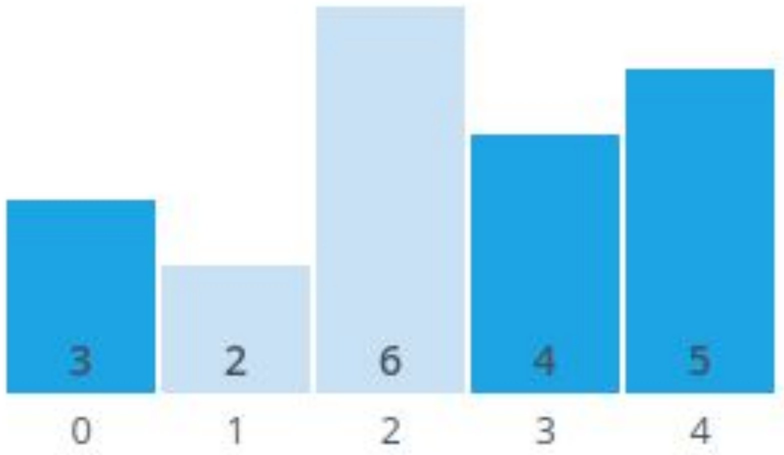


Compare The Elements



Compare The Elements and Swap

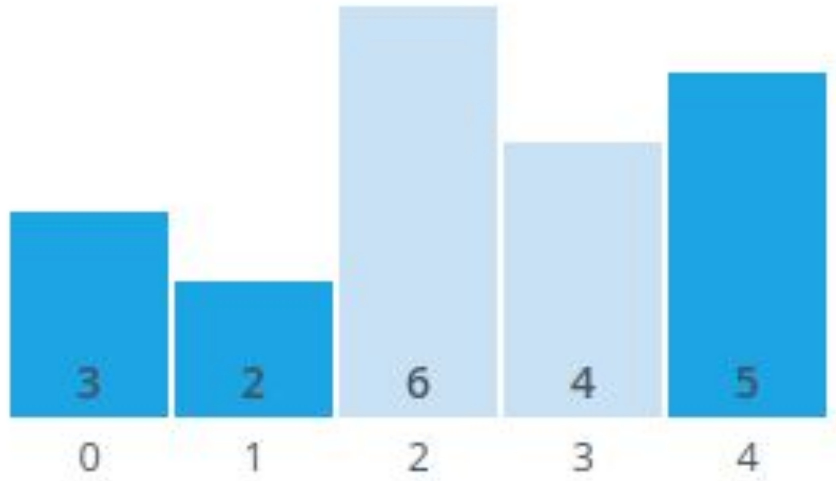




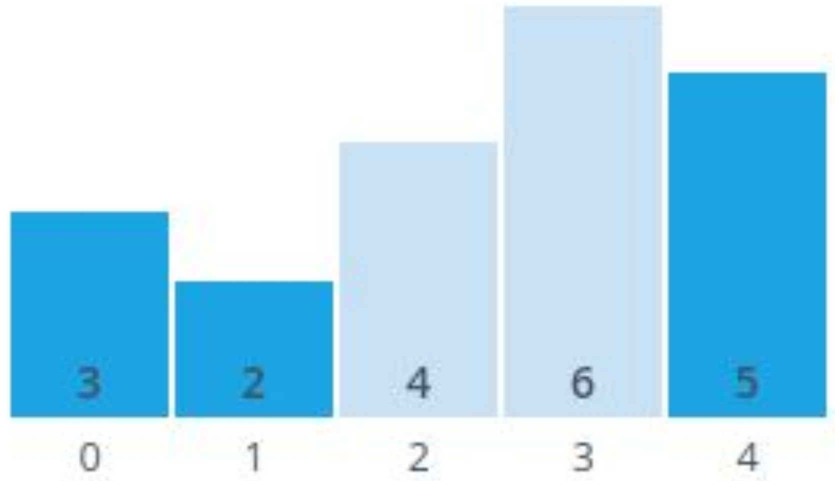
Compare Elements

Swap

Compare The Elements and Swap

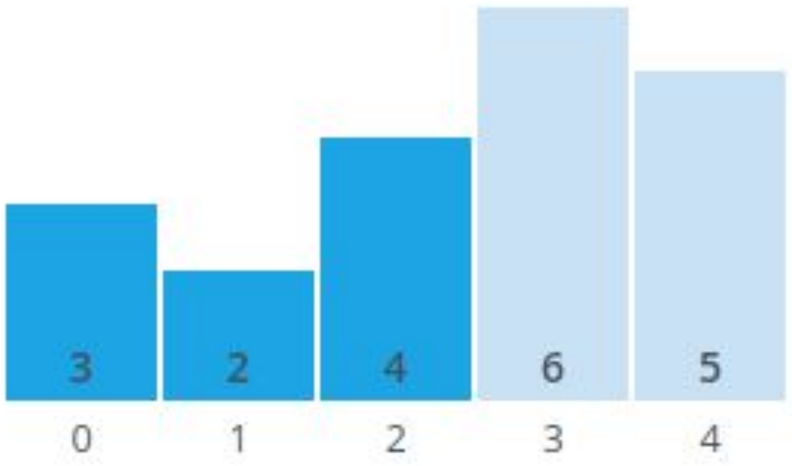


Compare Elements

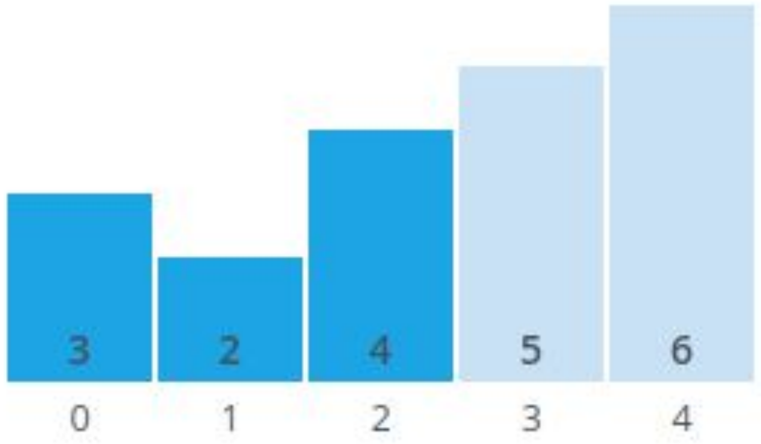


Swap

Compare The Elements and Swap



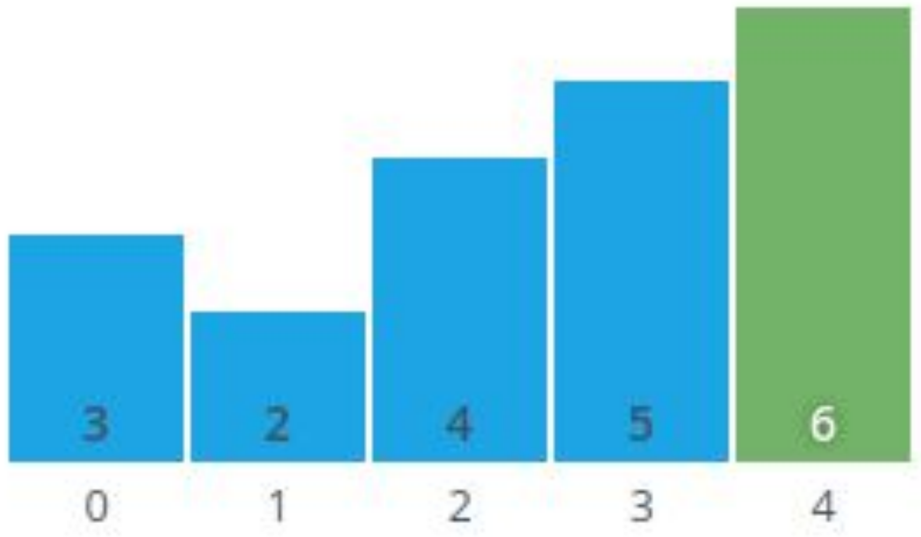
Compare Elements



Swap

First Pass Done

The last element processed is now in its final position. Now we will start the next pass for each element moving through the list.

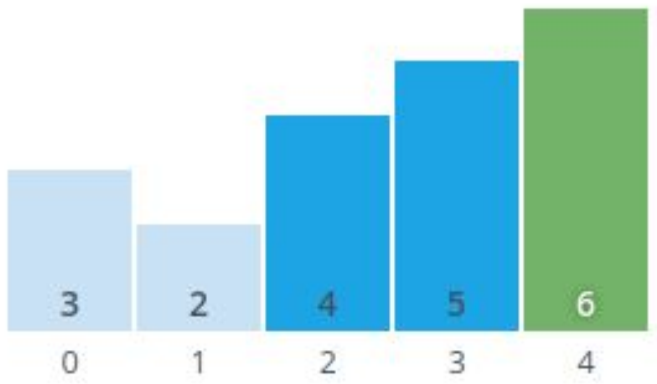


Q: What happens with the second pass?

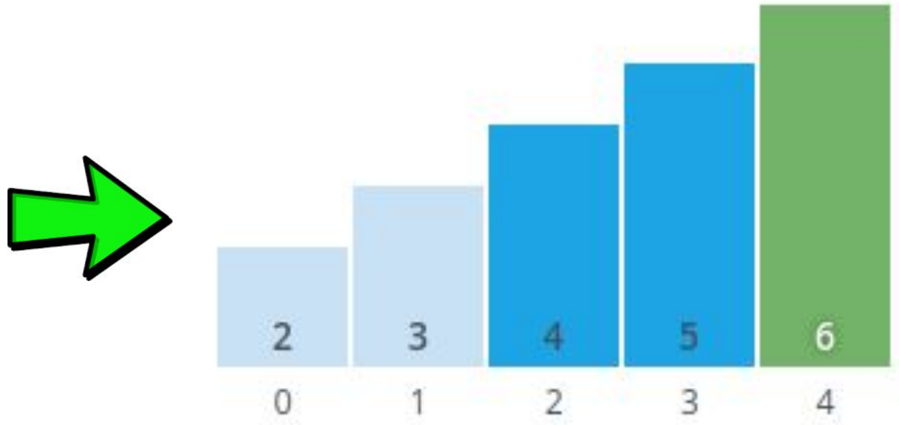
?

A2SV
Africa To Silicon Valley

Compare The Elements and Swap

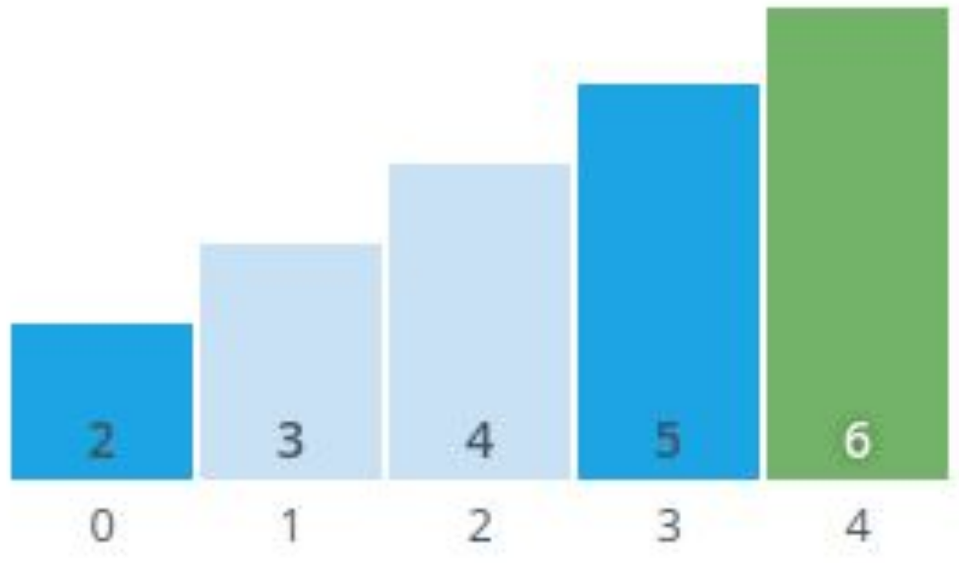


Compare the elements



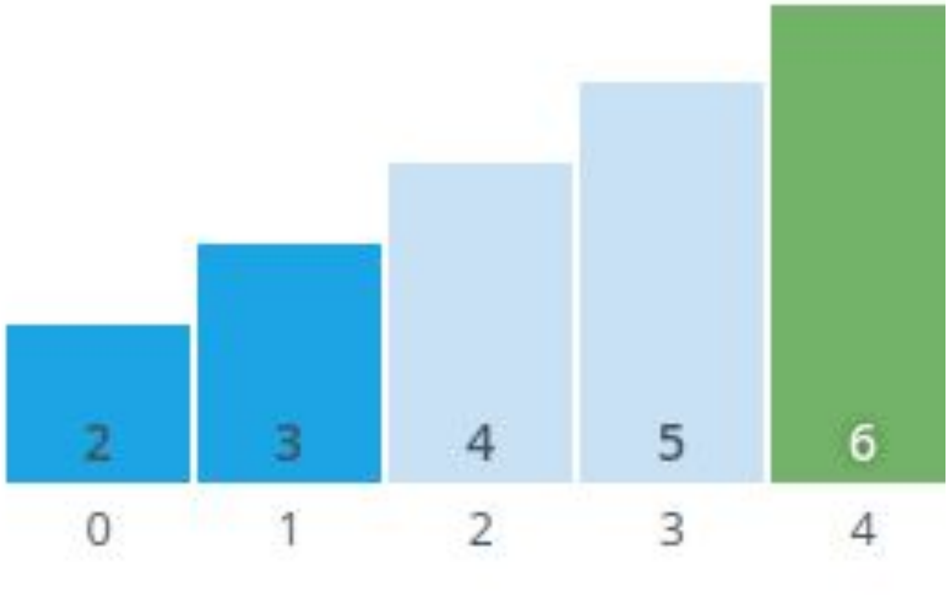
Swap

Compare The Elements



Compare elements

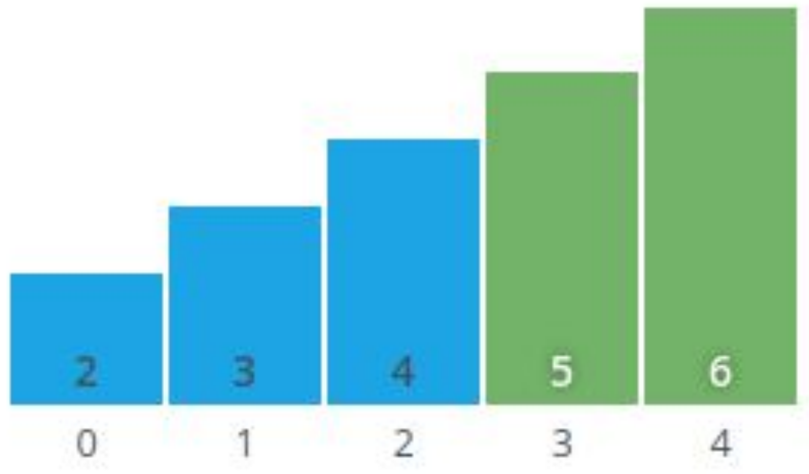
Compare The Elements



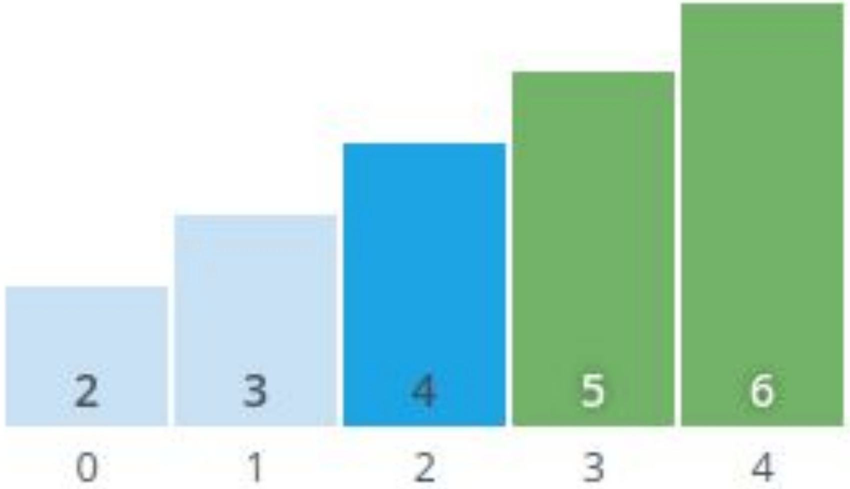
Compare elements

Done!

The last element processed is now in its final position. Now we will start the next move For each element moving through the list.

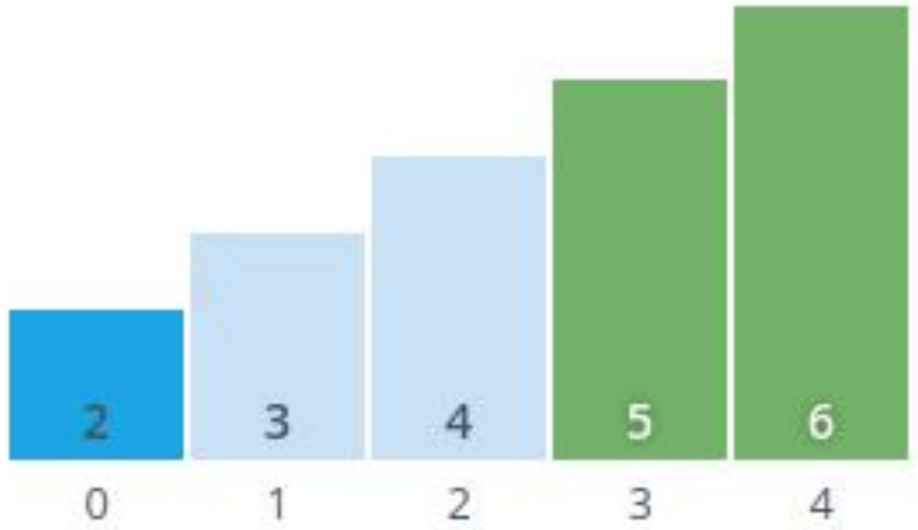


Compare The Elements



Compare elements

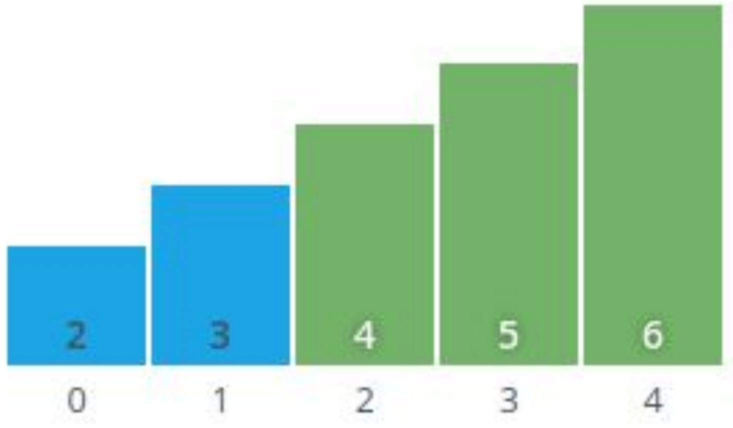
Compare The Elements



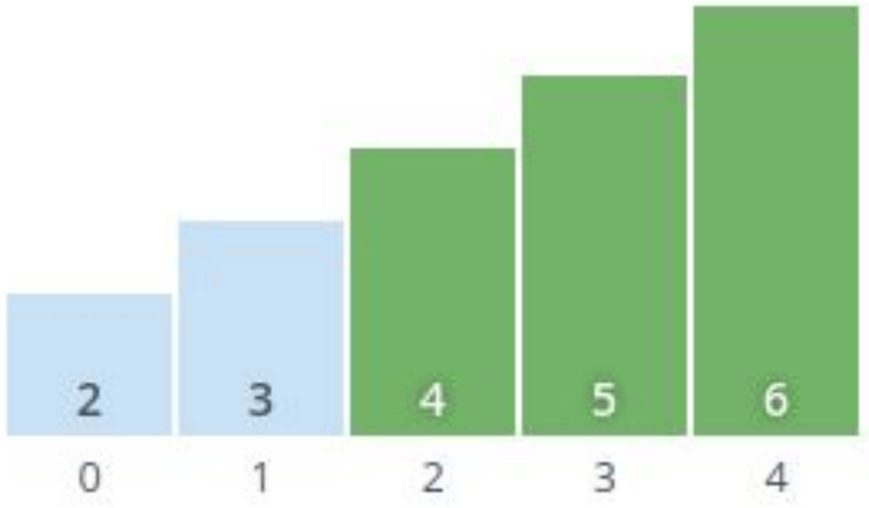
Compare elements

Done!

The last element processed is now in its final position. We will start the next path For each element moving through the list



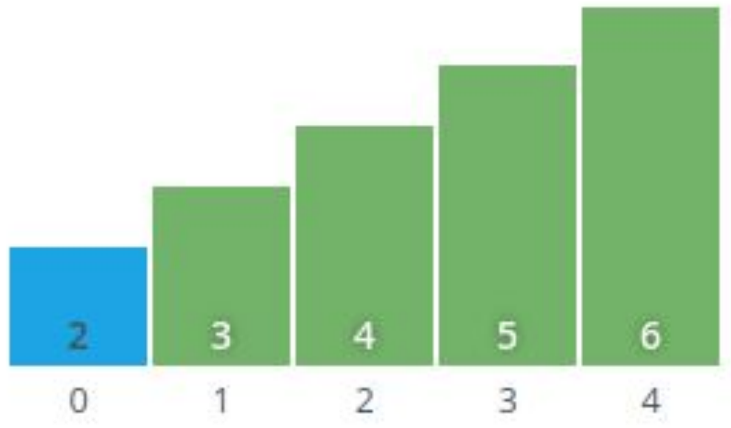
Compare The Elements



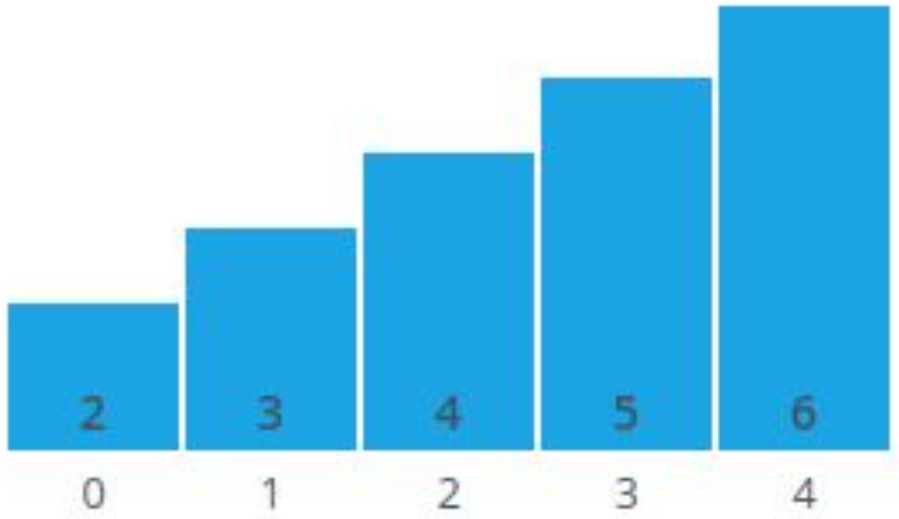
Compare elements

Done!

The last element processed is now in its final position.



Final Sorted Output



Time & Space Complexity

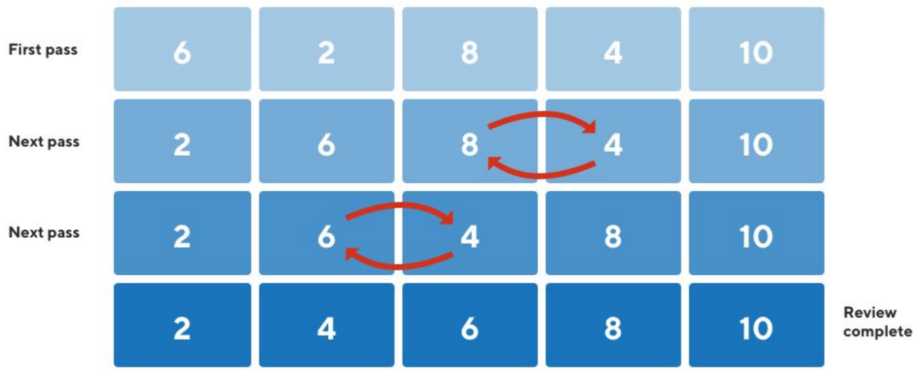
Worst case ? ____________

Best case ? ____________

Average case ? ____________

Stable ? ____________

In Place? ____________

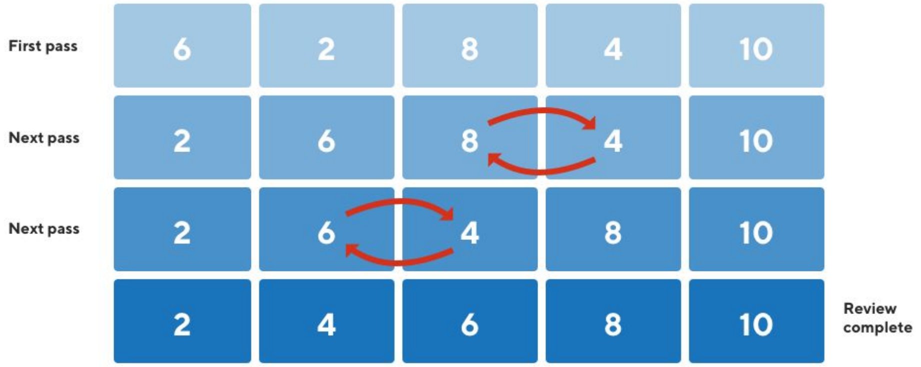


Time & Space Complexity

Time complexity: **$O(n^2)$**

Space complexity: **$O(1)$**

Worst case	**$O(n^2)$**
Best case	**$O(n^2)$**
Average case	**$O(n^2)$**
Stable ?	Yes
In Place?	Yes

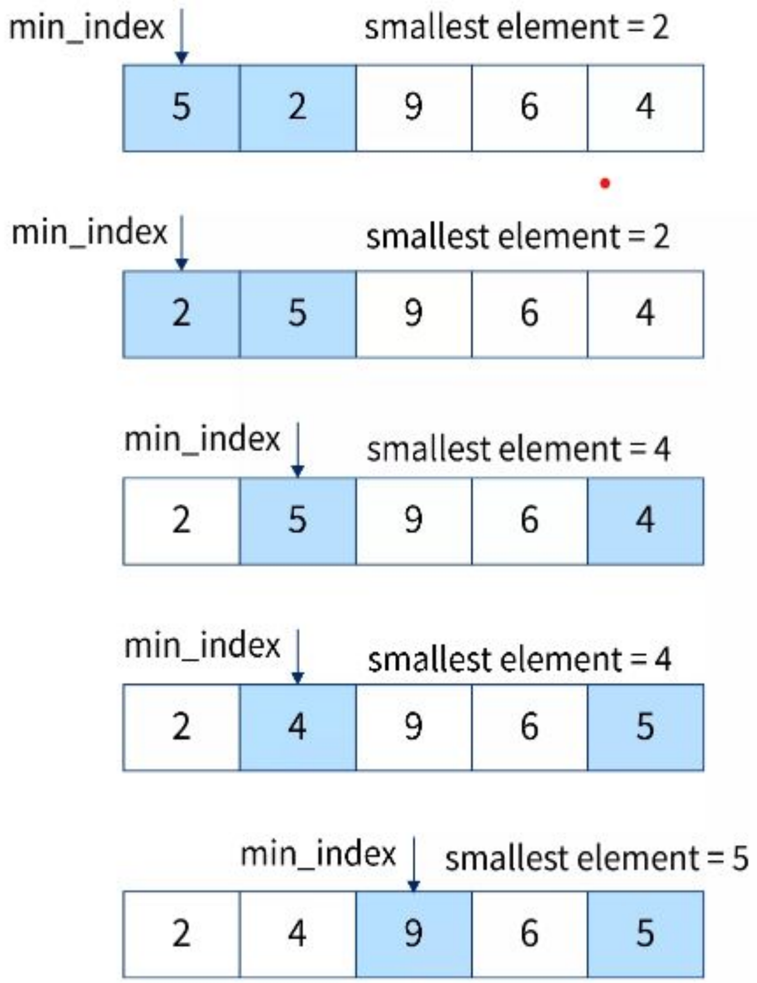


Solve by Using Bubble Sort

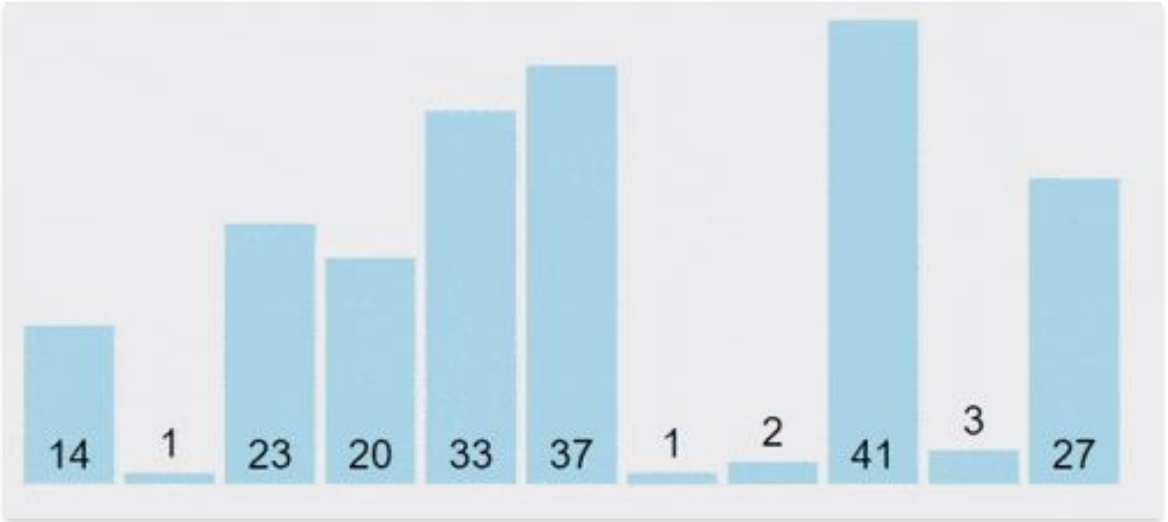
Implementation of Bubble sort

```
array = [64, 25, 12, 22, 11]
size = len(array)
for i in range(size):
    for j in range(size - i - 1):
        if array[j] > array[j + 1]:
            array[j], array[j + 1] = array[j + 1], array[j]
```

2. Selection Sort



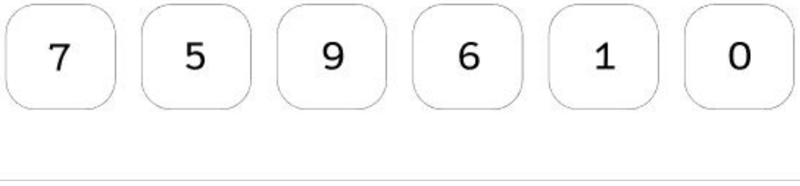
Selection Sort



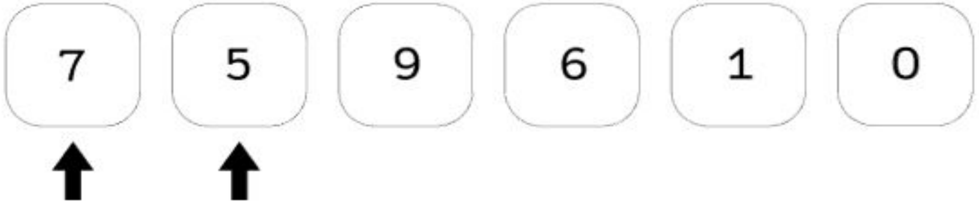
Selection sort selects the **smallest element** from an unsorted list in each iteration and places that element **at the beginning** of the unsorted list.

Selection Sort Simulation

Consider the following array as an example.

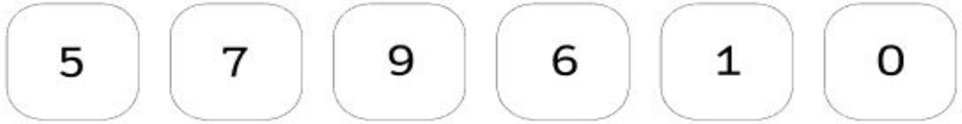


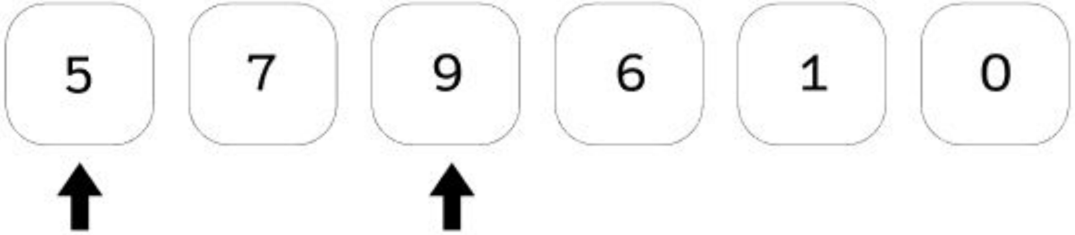
Selection Sort simulation



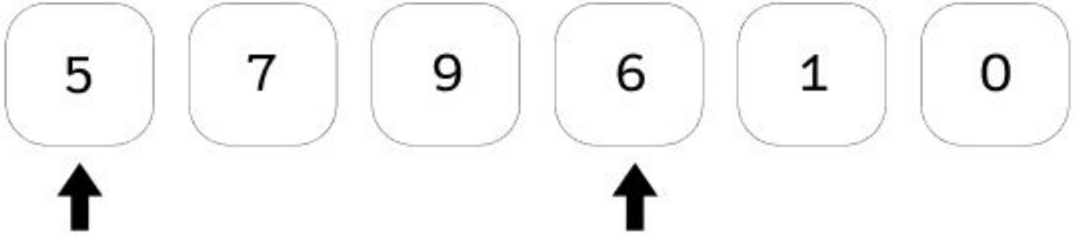
1st comparison

Swap

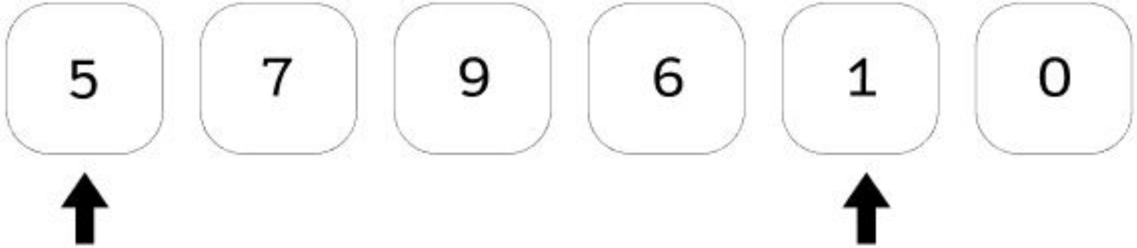




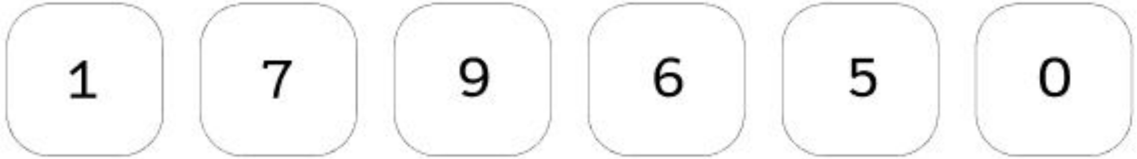
2nd comparison



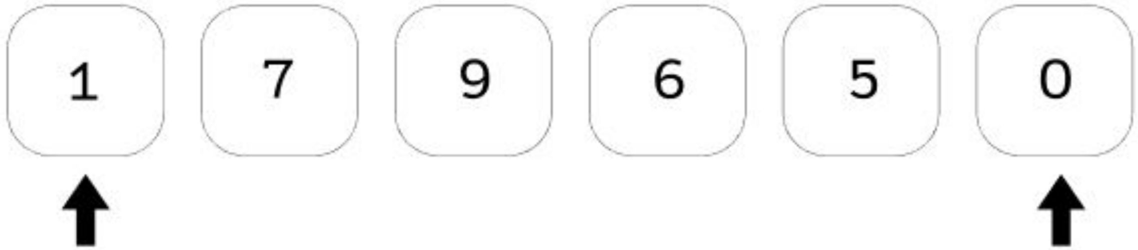
3rd comparison



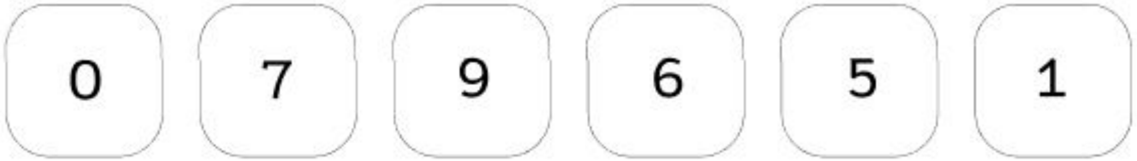
4th comparison



Swap

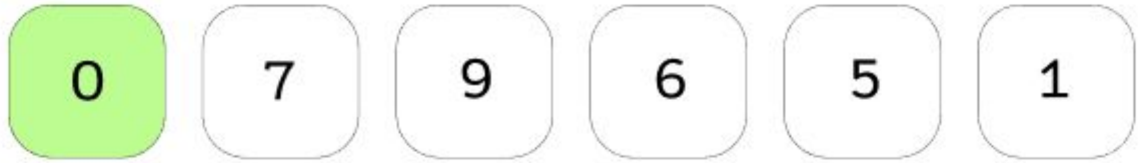


5th comparison

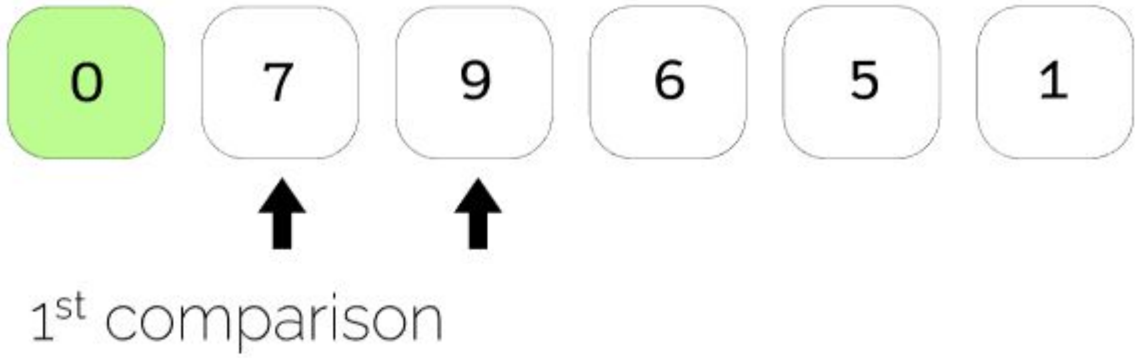


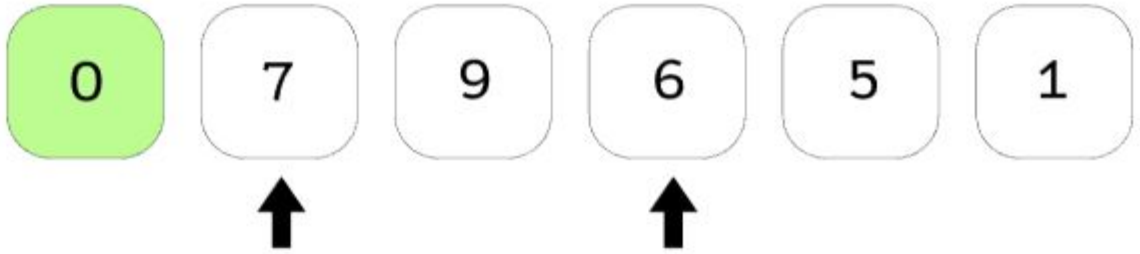
Swap

After one iterations, the least value is positioned at the beginning in a sorted manner.

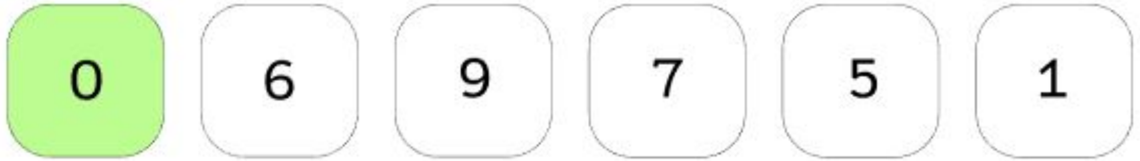


The same process is applied to the rest of the items in the array.

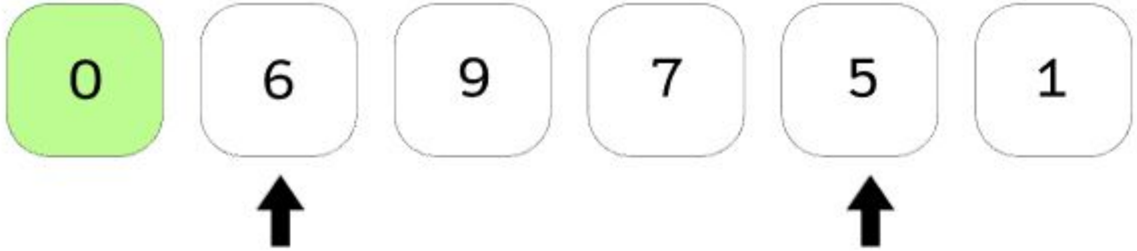




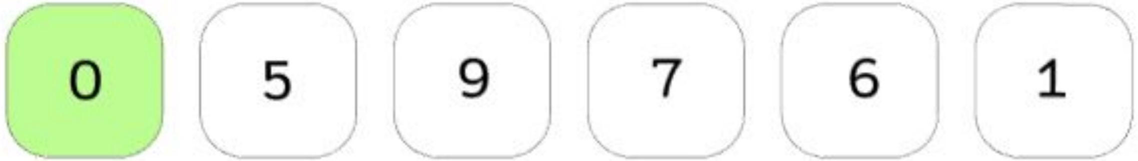
2nd comparison



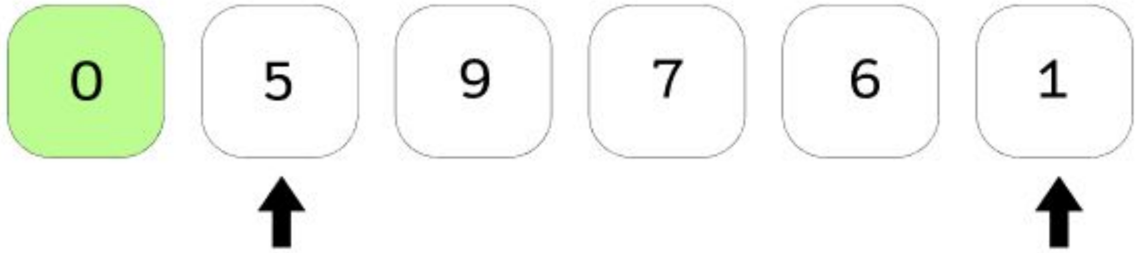
Swap



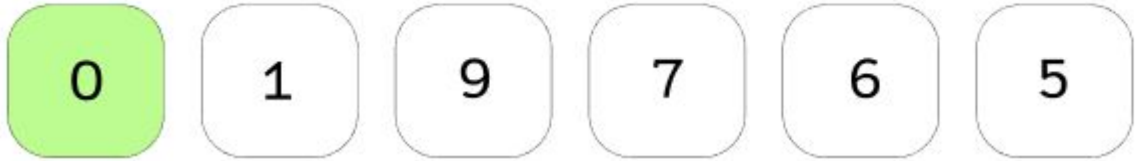
3rd comparison



Swap

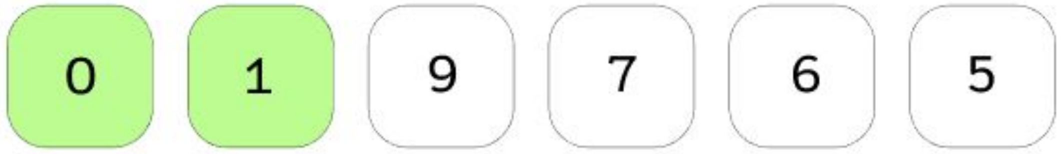


4th comparison

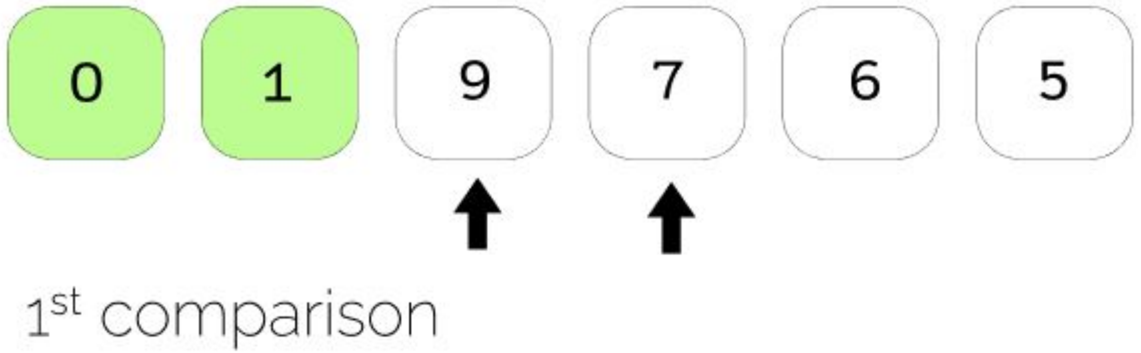


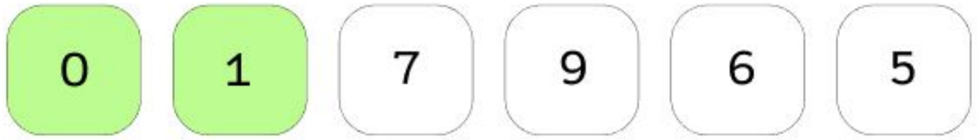
Swap

After two iterations, the two least value is positioned at the beginning in a sorted manner.

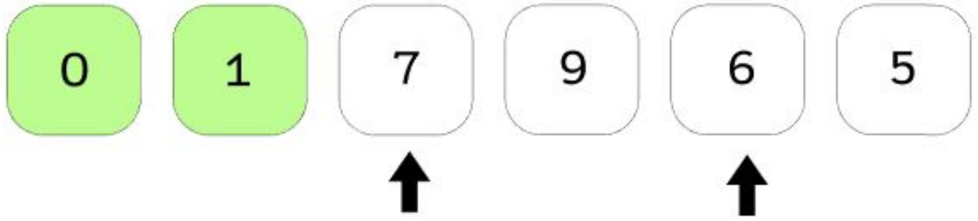


The same process is applied to the rest of the items in the array.

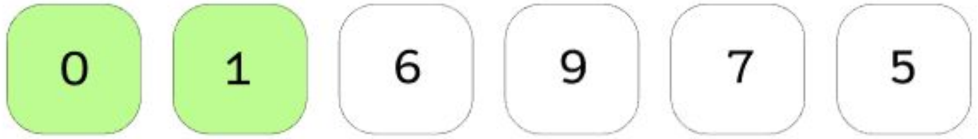




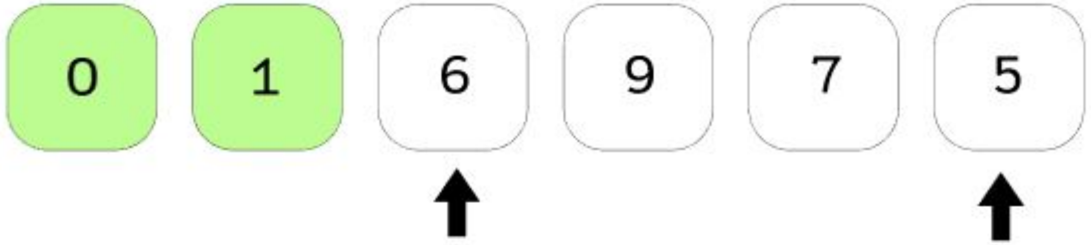
Swap



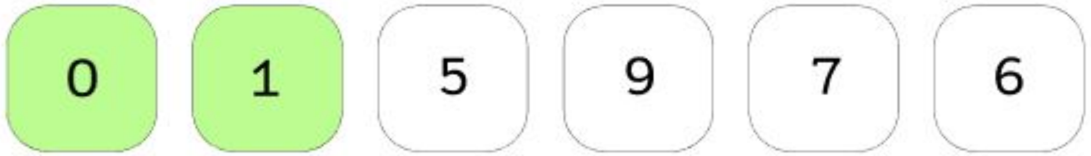
2nd comparison



Swap

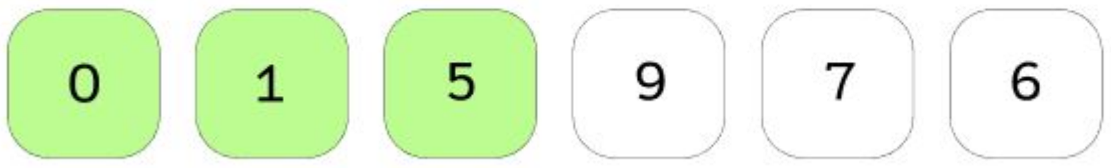


3rd comparison

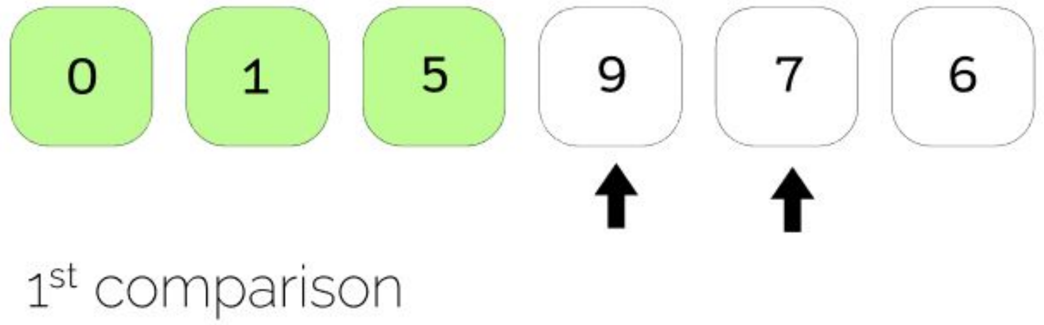


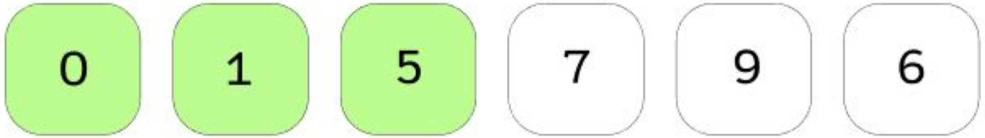
Swap

After three iterations, the three least value is positioned at the beginning in a sorted manner.

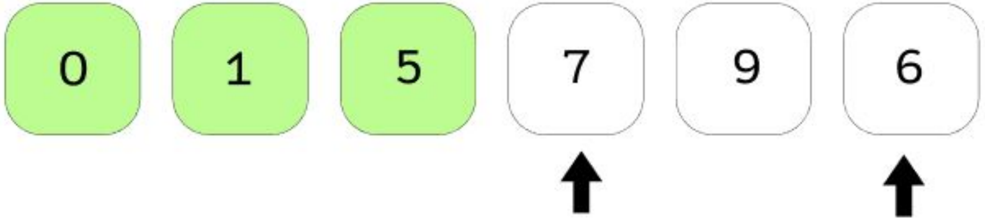


The same process is applied to the rest of the items in the array.

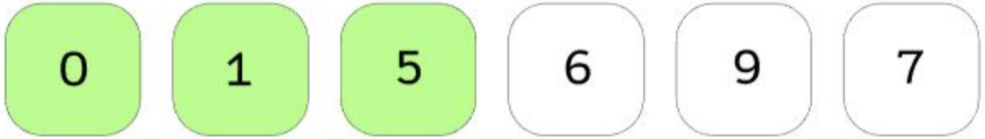




Swap

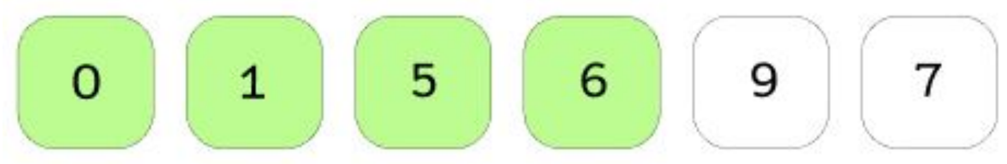


2nd comparison

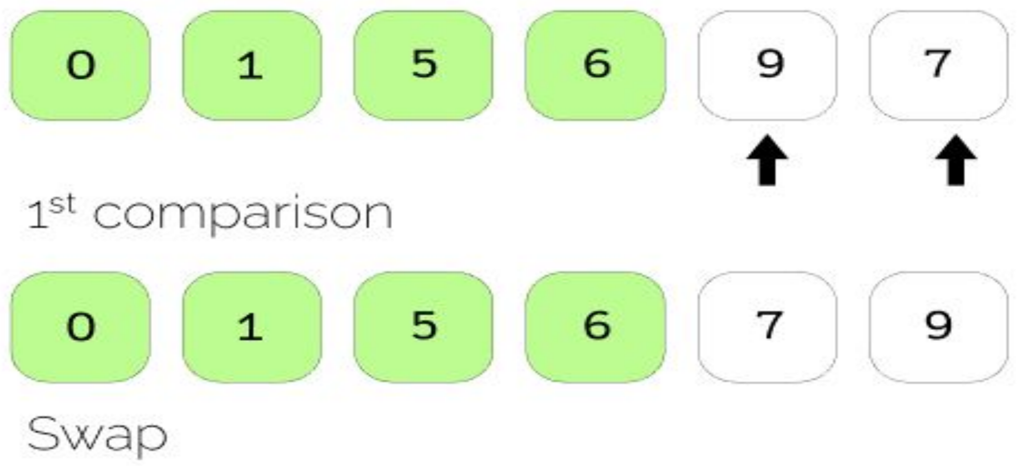


Swap

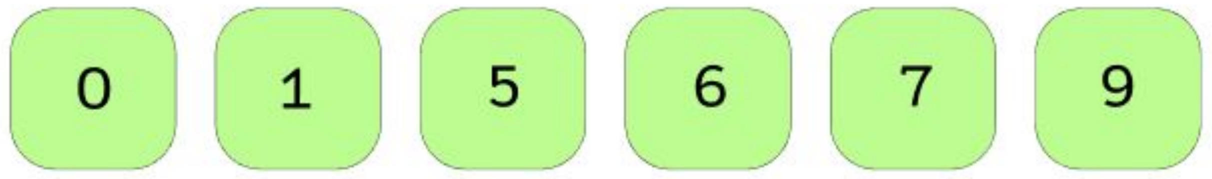
After four iterations, the four least value is positioned at the beginning in a sorted manner.



The same process is applied to the rest of the items in the array.



Done!



Q: What if we selected the largest one first and place it at the end?

Q: What happens when we put the smallest at the end?

?

Algorithm

- Set MIN_INDEX to location 0
- Repeat until list is sorted
    - Search the minimum element in the list
    - Swap with value at location MIN_INDEX
    - Increment MIN_INDEX to point to next element

Note: Every selection requires a search through the input list.

Time & Space Complexity

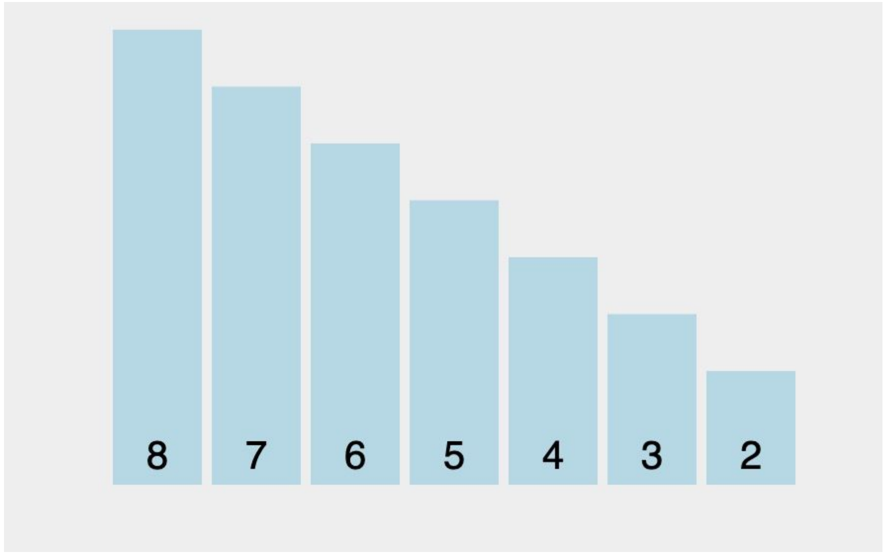
Worst case ? ____________

Best case ? ____________

Average case ? ____________

Stable ? ____________

In Place? ____________

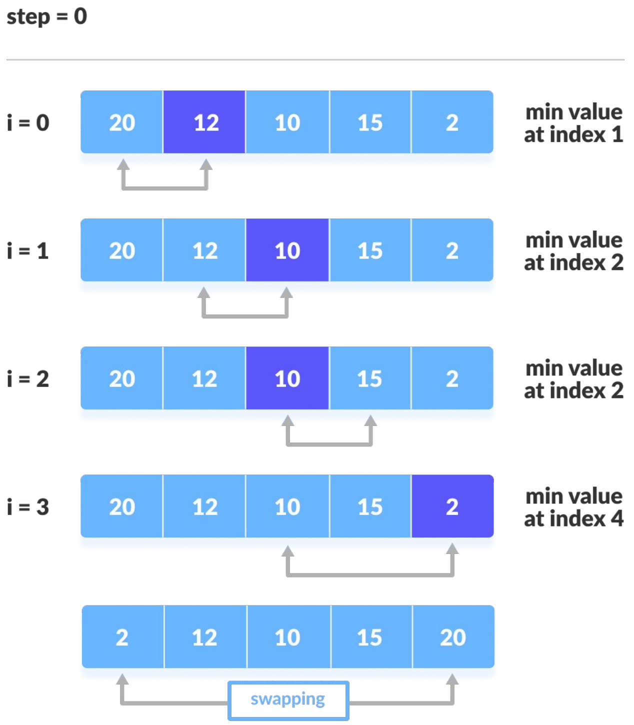


Time & Space Complexity

Time complexity: **O(n²)**

Space complexity: **O(1)**

Worst case	**O(n²)**
Best case	**O(n²)**
Average case	**O(n²)**
Stable ?	No
In Place?	Yes



Solve by Using Selection Sort

Implementation of Selection Sort

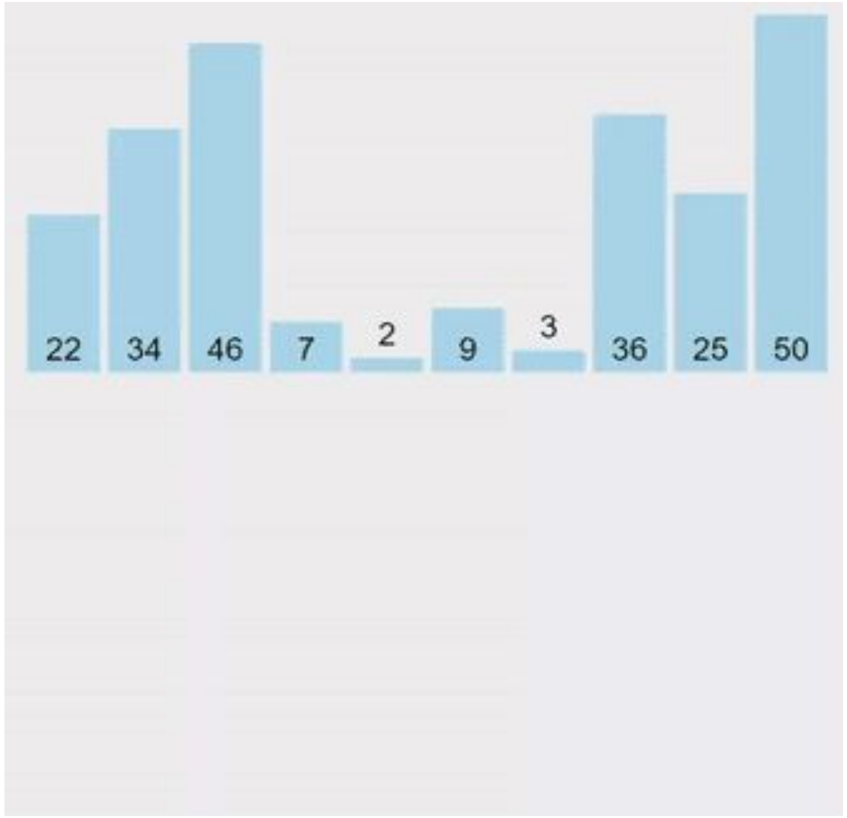
```
array = [64, 25, 12, 22, 11]
size = len(array)
for i in range(size):
    min_idx = i
    for j in range(i + 1, size):
        if array[min_idx] > array[j]:
            min_idx = j
    array[i], array[min_idx] = array[min_idx], array[i]
return  array
```

Africa To Silicon Valley

3. Insertion Sort

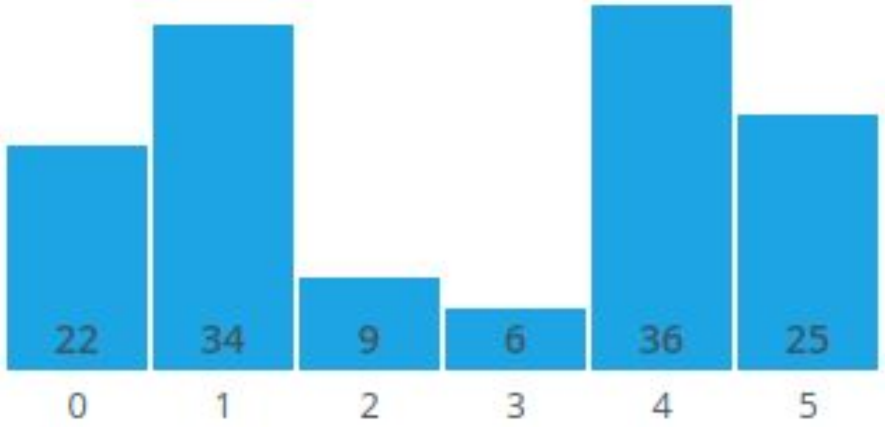
Africa To Silicon Valley

Insertion Sort



Let's sort the beginning of the list, and insert new elements to the sorted part one by one.

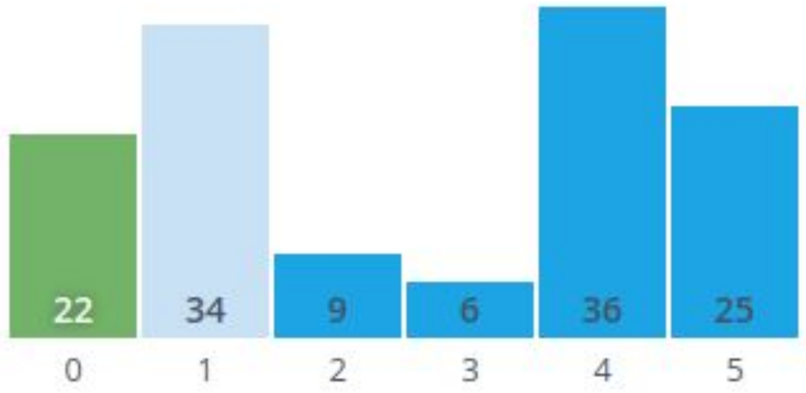
Insertion Sort Simulation



Insertion Sort Simulation

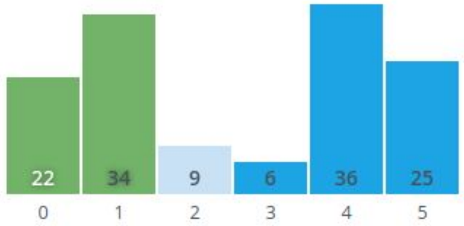
Green records to the left are always sorted.

We begin with the record in position 0 in the sorted portion, and we will be moving the record in position 1 (in blue) to the left until it is sorted.

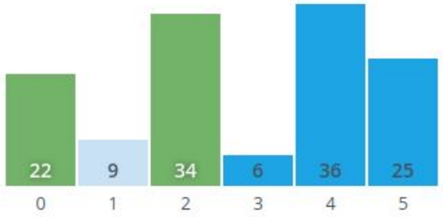


Insertion Sort Simulation

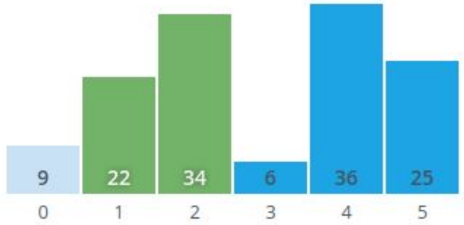
Move the blue record to the left until it reaches the correct position.



Swap

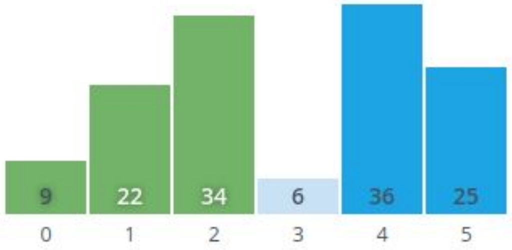


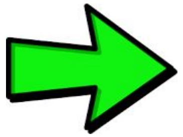
Swap

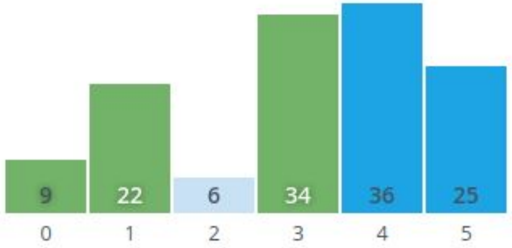


Insertion Sort Simulation

Move the blue record to the left until it reaches the correct position.

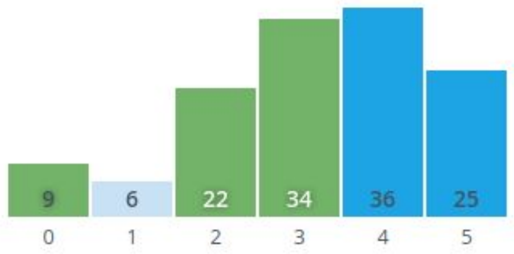


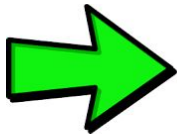




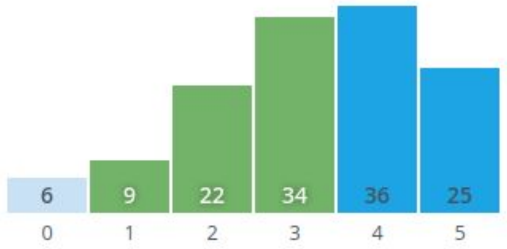
Insertion Sort Simulation

Move the blue record to the left until it reaches the correct position.



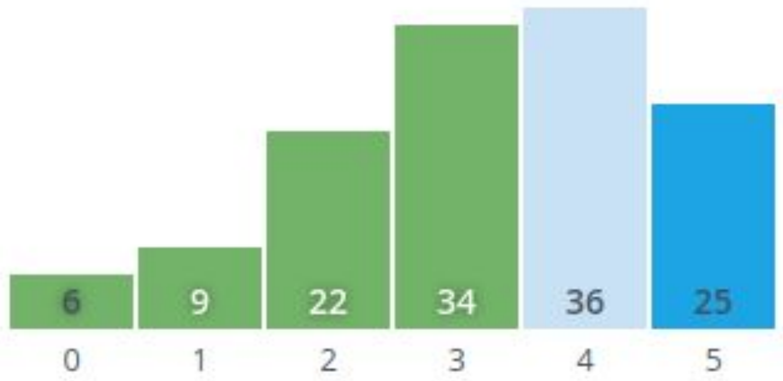


Swap



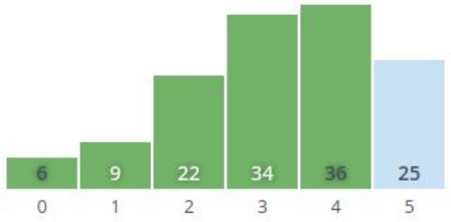
Insertion Sort Simulation

Move the blue record to the left until it reaches the correct position.

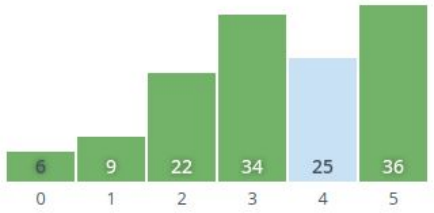


Insertion Sort Simulation

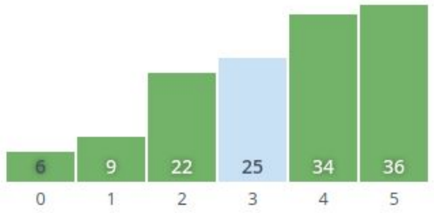
Move the blue record to the left until it reaches the correct position.



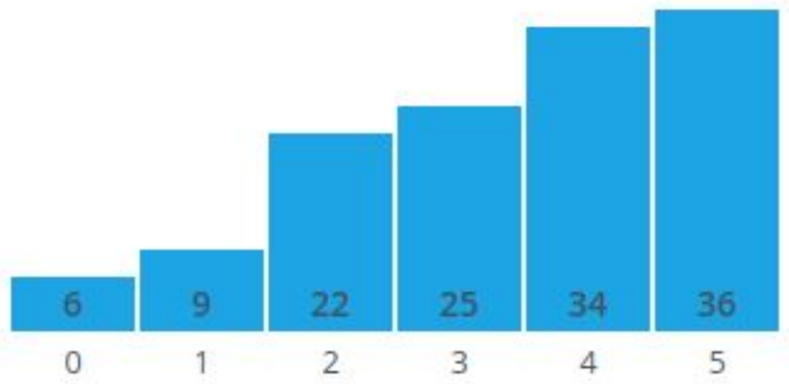
Swap



Swap

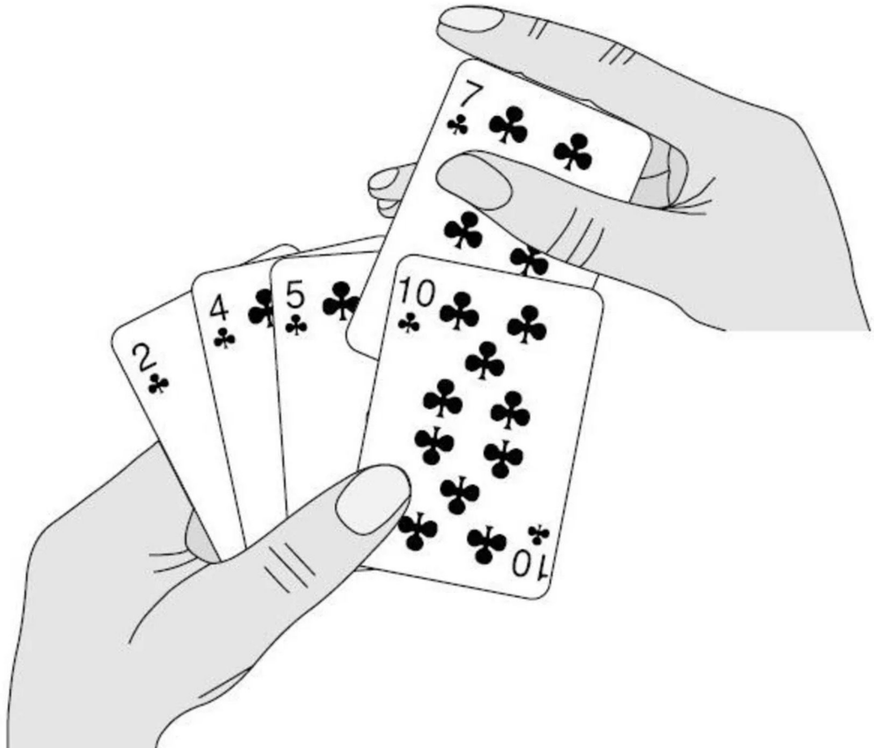


Final Sorted Output



Q: What would be a real life example of Insertion sort?

Insertion Sort



Algorithm

- If the element is the first one, it is already sorted.
- Move to next element
- Compare the current element with all elements in the sorted array
- Shift all the the elements in sorted sub-list that is greater than the value to be sorted.
- Insert the value at the correct position
- Repeat until the complete list is sorted

Time & Space Complexity

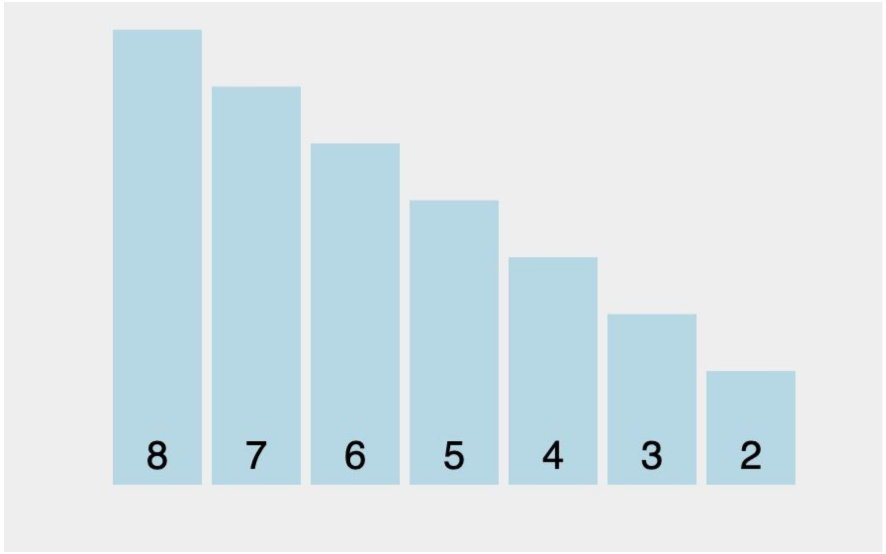
Worst case ? ____________

Best case ? ____________

Average case ? ____________

Stable ? ____________

In Place? ____________



Time & Space Complexity

Time complexity: **O(n^2)**

Space complexity: **O(1)**

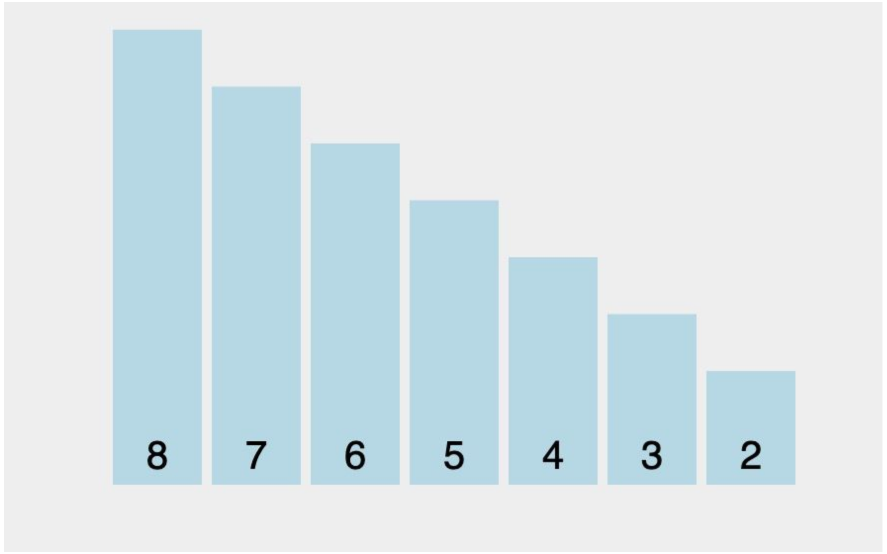
Worst case __**O(n^2)**___

Best case __**O(n)**___

Average case __**O(n^2)**___

Stable ? ____Yes________

In Place? ____Yes________



Solve by Using Insertion Sort

Implementation

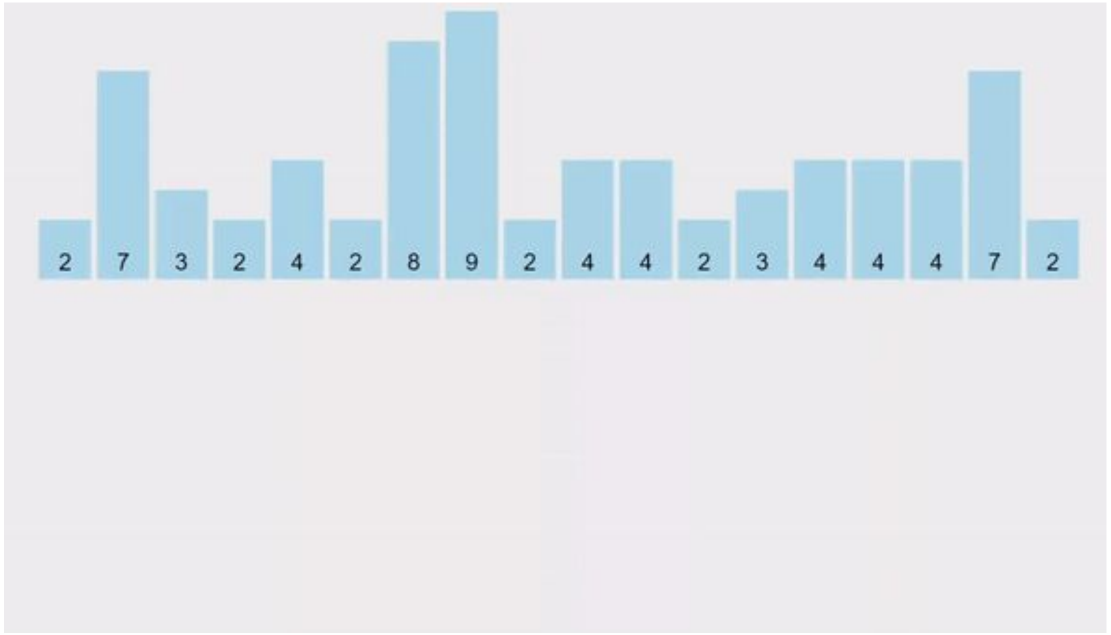
```
array = [64, 25, 12, 22, 11]
size = len(array)
for i in range(1, size):
    key = array[i]
    j = i - 1

    while j >= 0 and key < arr[j]:
        array[j + 1] = array[j]:
        j -= 1
    array[j + 1] = key
return  array
```

Distribution Sorting

1. Counting Sort

Definition



If the range of the numbers is small enough that can fit in memory;

Count the occurrence of items then generate output based on counts in counter array.

Illustration

Phase 1: **Counting**

First, a **storage array** is created whose length corresponds to the number range. Then you iterate once over the elements to be sorted, and, for each element, you **increment the value** in the array at the position corresponding to the element.

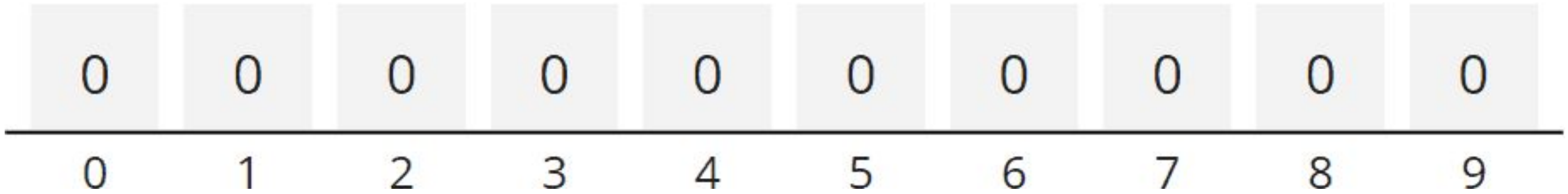
Consider the following array as an example.

3	7	4	6	6	3	0	3	8	6	6	9	6	2	8

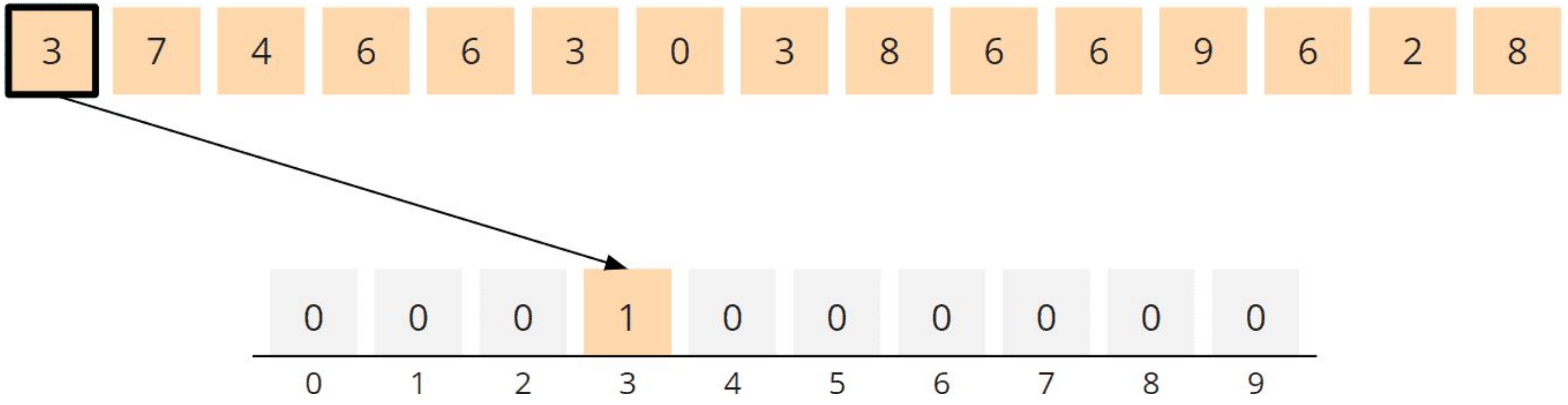
Find the maximum number

9
max

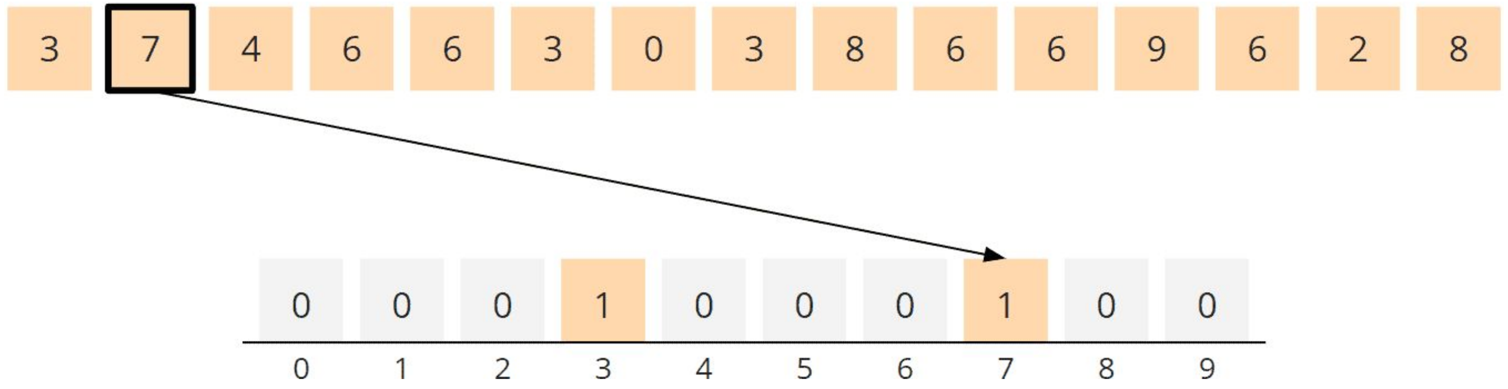
We create an additional array with the length of the maximum number +1. In our case we create an array with length of 10, initialized with zeros.



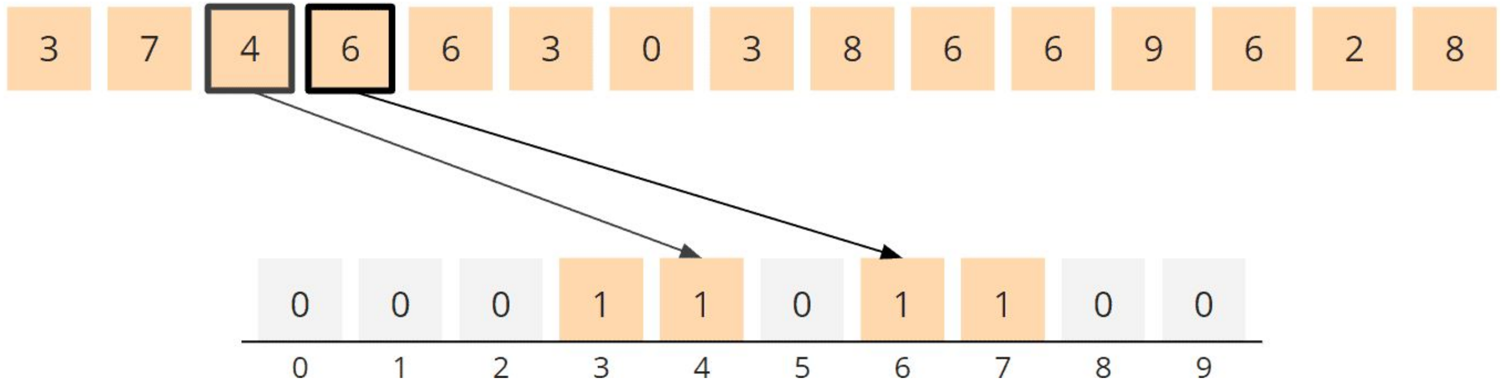
Now we iterate over the array to be sorted. The first element is a 3 – accordingly, we increase the value in the auxiliary array at position 3 by one:



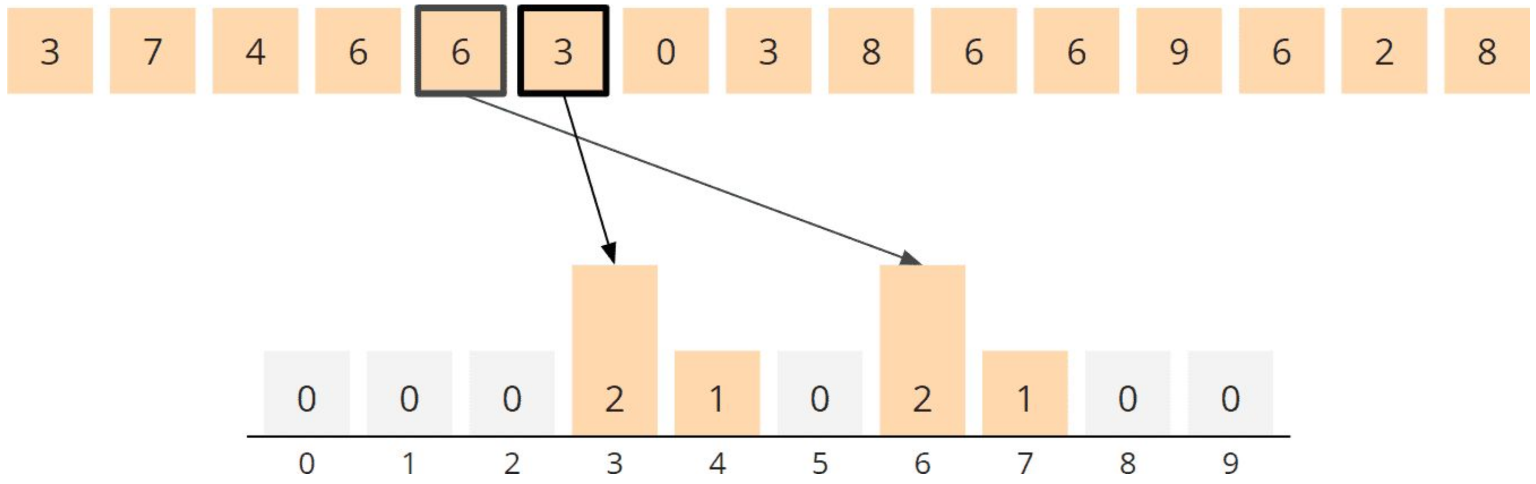
The second element is a 7. We increment the field at position 7 in the helper array



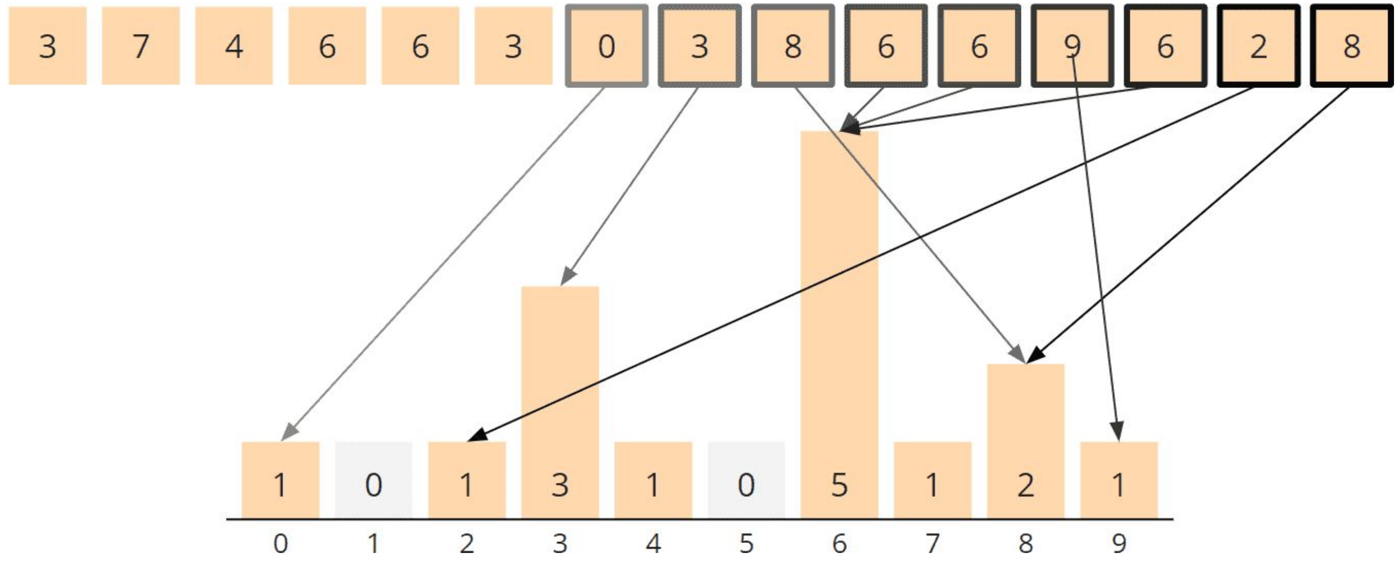
Elements 4 and 6 follow – thus, we increase the values at positions 4 and 6 by one each



The next two elements – the 6 and the 3 – are two elements that have already occurred before. Accordingly, the corresponding fields in the auxiliary array are increased from 1 to 2



The principle should be clear now. After also increasing the auxiliary array values for the remaining elements, the auxiliary array finally looks like this

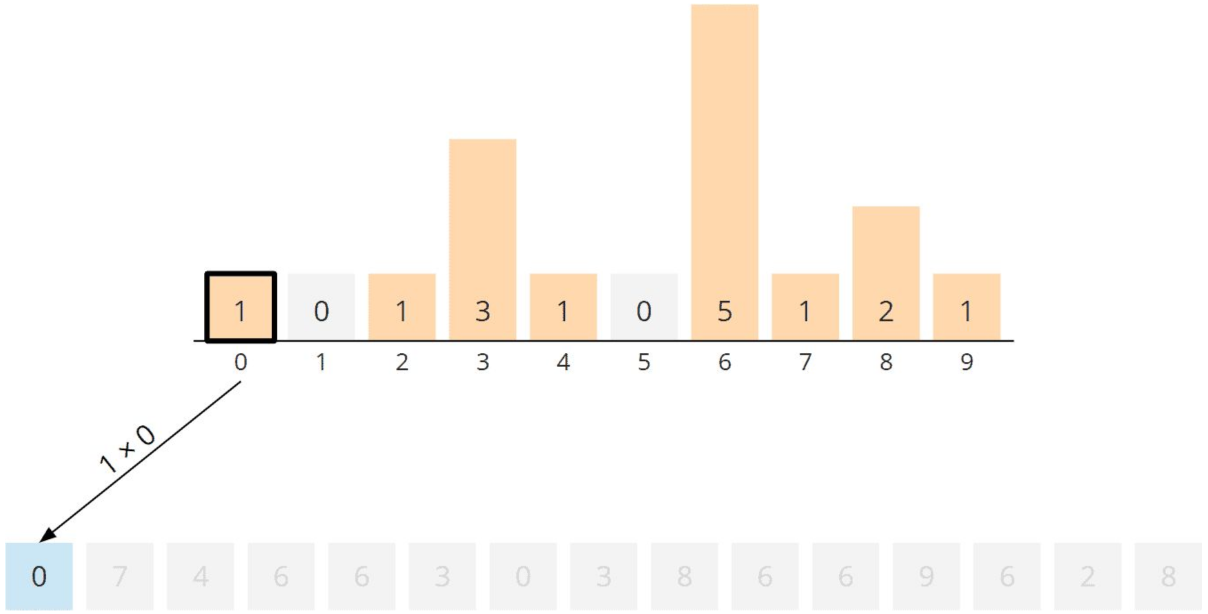


Illustration

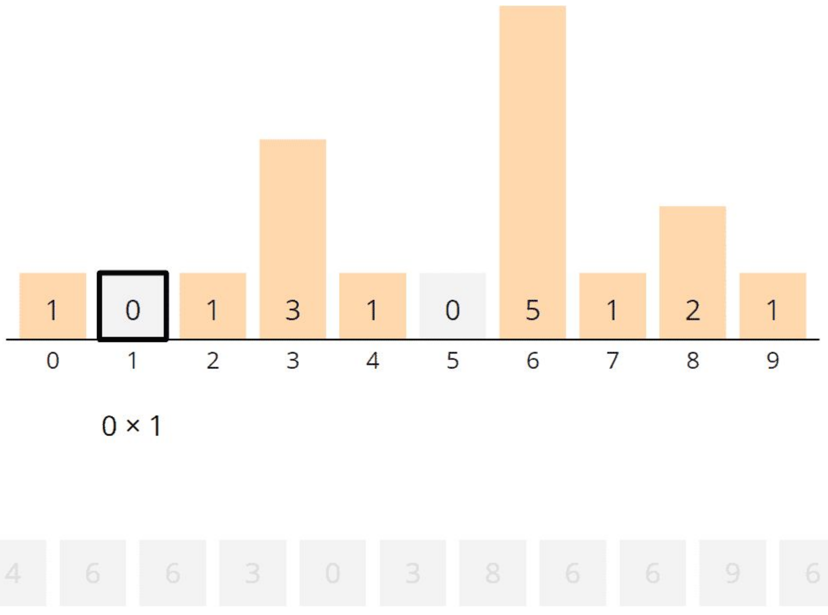
Phase 2: Rearranging

We iterate once over the histogram array. We write the respective array index into the array to be sorted as often as the histogram indicates at the corresponding position.

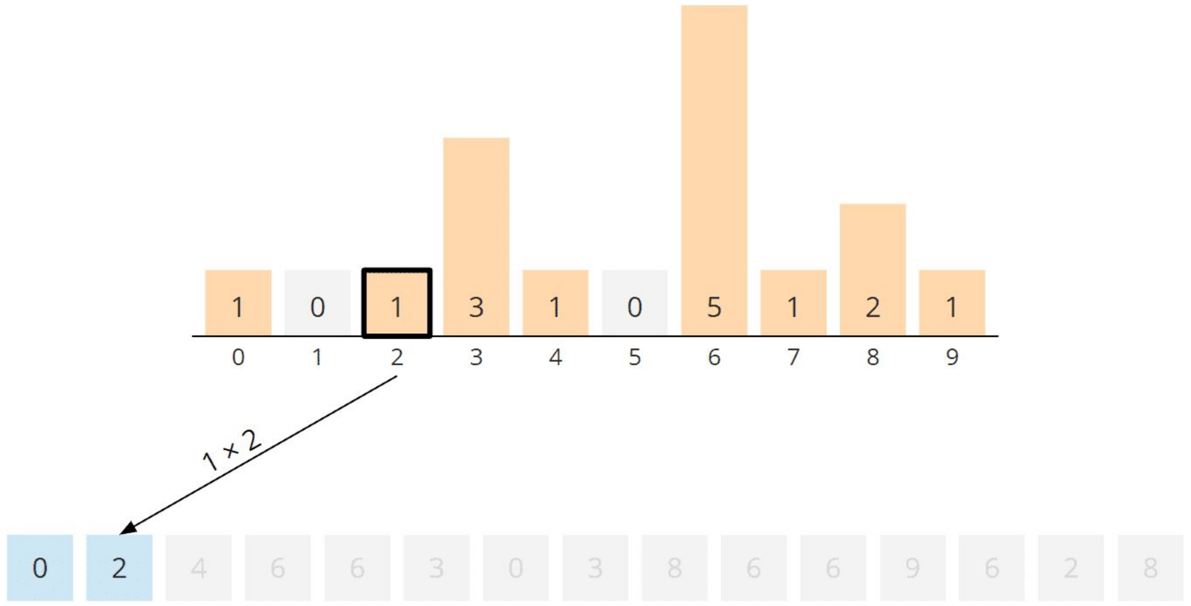
In the example, we start at position 0 in the auxiliary array. That field contains a 1, so we write the 0 exactly once into the array to be sorted.



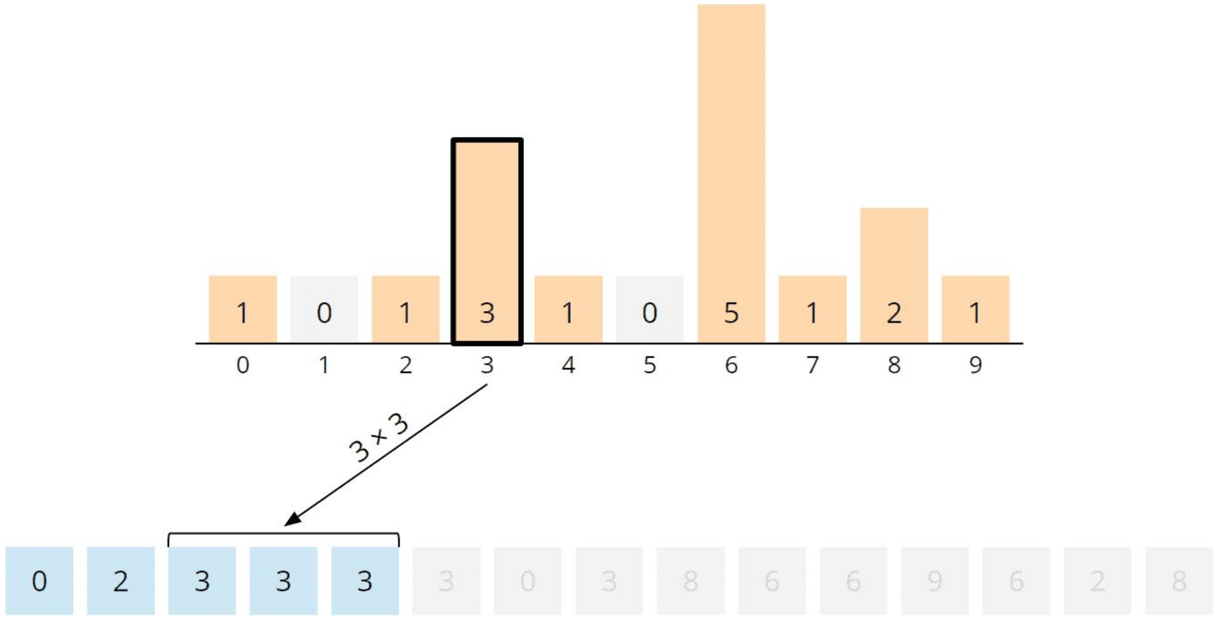
At position 1 in the histogram, there is a 0, meaning we skip this field – no 1 is written to the array to be sorted.



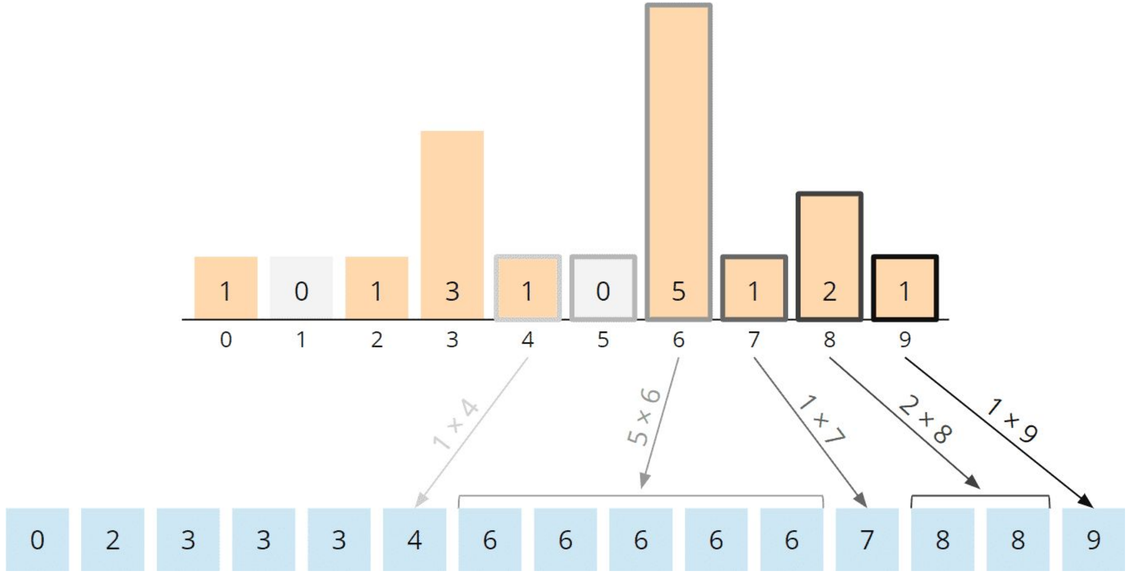
Position 2 of the histogram again contains a 1, so we write 2 once into the array to be sorted



We come to position 3, which contains a 3; so we write three times a 3 into the array



And so it goes on. We write 4 once, 6 five times, 7 once, 8 twice and finally 9 once into the array to be sorted



Solve by Using Counting Sort

Africa To Silicon Valley

Implementation

```
maximum = max(nums)
count = [0] * (maximum + 1)
for num in nums:
    count[num] += 1

target = 0
for index, value in enumerate(count):
    for i in range(value):
        nums[target] = index
        target += 1
```

Count Sorting with Negative Numbers

Q: Will the above implementation work when we include negative numbers?

Q: If NO, what can we do to make it work?

Count Sorting with Negative Numbers

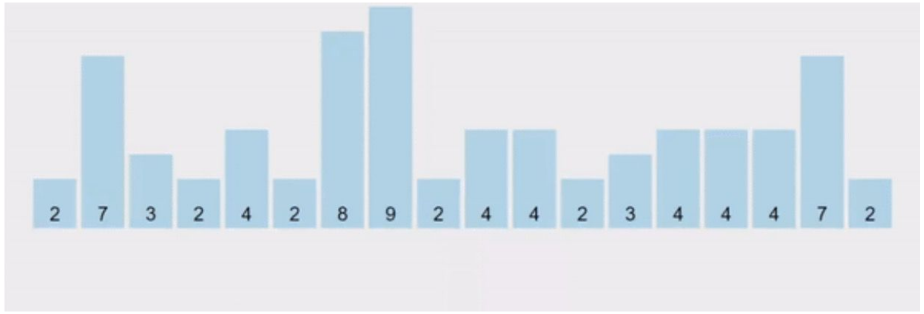
The problem with the previous counting sort was that we could not sort the elements if we have negative numbers in them. Because there are **no negative array indices**.

Implementation

```
maximum = max(nums)
minimum = abs(min(nums))
count = [0] * (maximum + minimum + 1)
for num in nums:
    count[num + minimum] += 1

target = 0
for index, value in enumerate(count):
    for i in range(value):
        nums[target] = index - minimum
        target += 1
```

Time complexity



Worst case ? ____________

Best case ? ____________

Average case ? ____________

Stable ? ____________

In Place?____________

Note: Counting sort is most efficient if the range of input values is not greater than the number of values to be sorted.

Time complexity

The time complexity of counting sort algorithm is O(n+k) where n is the number of elements in the array and k is the range of the elements.

Worst case	__**O(n+k)**___
Best case	__**O(n+k)**___
Average case	__**O(n+k)**___
Stable ?	______No______
In Place?	_____No_______

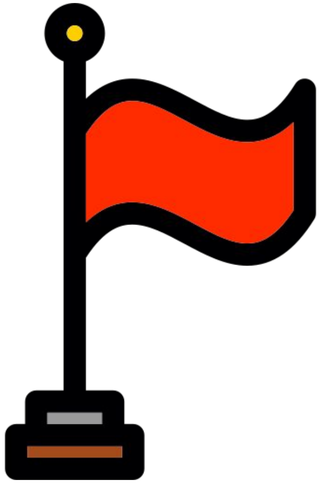
Note: Counting sort is most efficient if the range of input values is not greater than the number of values to be sorted.

- Any **improvement in sorting time** significantly affect the **overall efficiency** and saves a great deal of computer time.
- Space constraints are usually less important than time complexity, for most sorting algorithms the amount of space needed is closer to **O(n).**

Real Life Computing

Summary

	Time Complexity					
	Best Case	**Average Case**	**Worst Case**	**Space**	**Stable**	**In Place**
Bubble	O(n^2)	O(n^2)	O(n^2)	O(1)	Yes	Yes
Insertion	O(n)	O(n^2)	O(n^2)	O(1)	Yes	Yes
Selection	O(n^2)	O(n^2)	O(n^2)	O(1)	No	Yes
Counting	O(n + k)	O(n + k)	O(n + k)	O(n)	No	No



Checkpoint

Link

Using built-in functions to sort

Africa To Silicon Valley

Python libraries: sorted() and .sort()

```
array = [1,2,5,4,3,6]
array.sort()
print(array)
# 1 2 3 4 5 6
```

```
array = [1,2,5,4,3,6]
sorted_array = sorted(array)
print(sorted_array)
# 1 2 3 4 5 6
```

```
array = [1,2,5,4,3,6]
array.sort(reverse=True)
print(array)
# 6 5 4 3 2 1
```

Africa To Silicon Valley

Writing custom comparator

Default sorting sorts **based on the values**

What if you wanted to sort based on:

- In reverse
- Sorting based on some other criteria besides the values

Example: sort an array based on a cost

```
costs = [1,3,2,5,1,3]
students = [1,2,3,4,5,6]

studentToCost = {}
for idx in range(len(students)):
    studentToCost[students[idx]] = costs[idx]

def customComparator(item):
    return studentToCost[item]

students.sort(key = customComparator)
print(students)
# [1, 5, 3, 2, 6, 4]
```

Selecting a sorting algorithm

Language libraries often have sorting algorithms so we won't be coding our own sorting algorithms every time.

Q: When the range of numbers is small enough?

Q: When half of the array is already sorted?

Practice time

[Relative Sort Array]

Africa To Silicon Valley

Helpful Resources

- Big O Cheat Sheet
- Insertion Sort
- Selection Sort
- Counting Sort
- Stable vs Unstable Sort
- Visualizations

Practice Questions

Bubble Sort

Insertion Sort

Counting Sort

Selection Sort

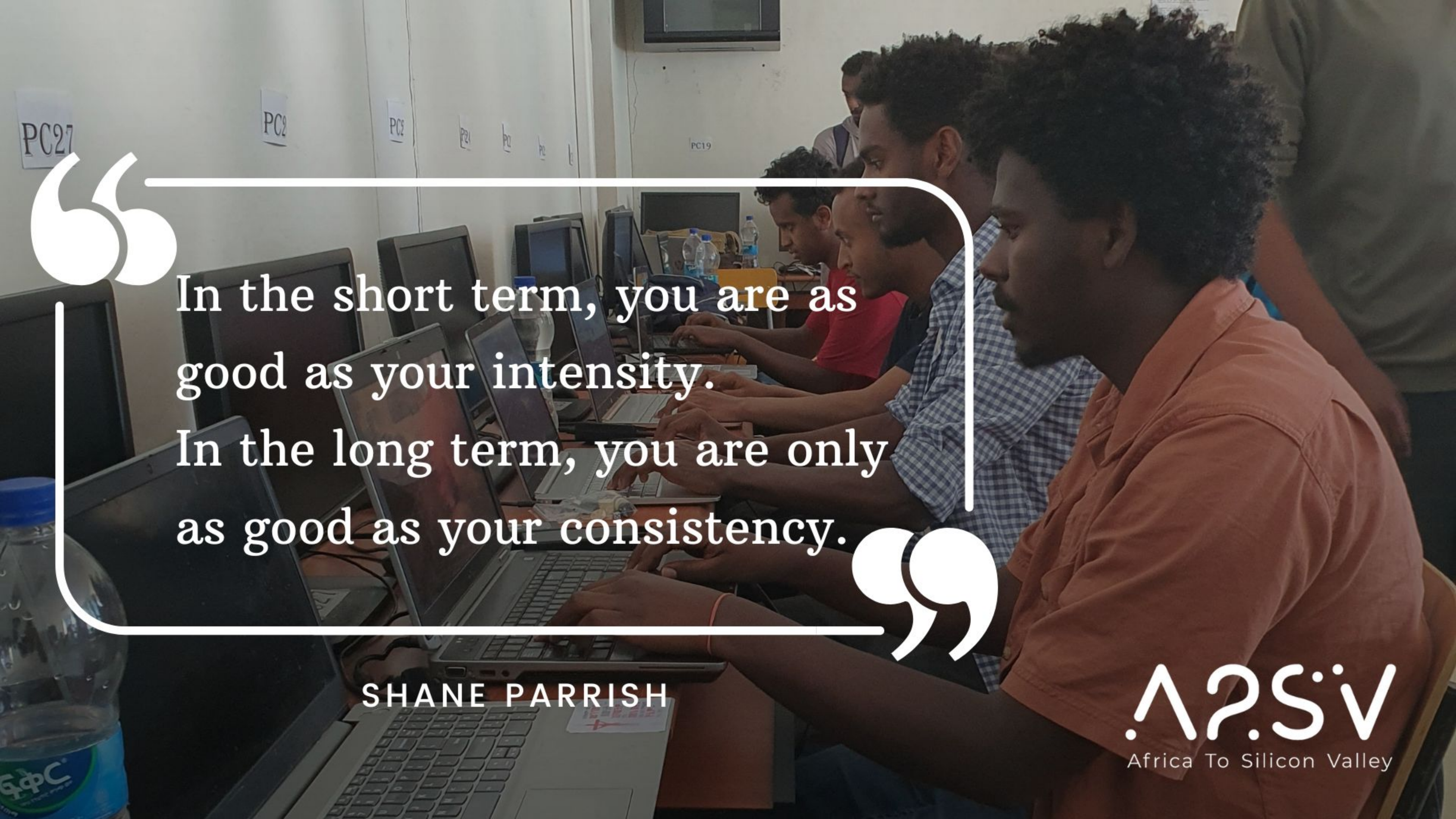
Sort Colors

Pancake Sorting

Find Target Indices After Sorting Array

Maximum Number Of Coins You Can Get

How Many Numbers Are Smaller Than The Current Number



In the short term, you are as good as your intensity.
In the long term, you are only as good as your consistency.

SHANE PARRISH

A2SV
Africa To Silicon Valley