Sorting Algorithms

Pool Game

Eight ball, also called stripes and solids, popular American pocket-billiards game in which 15 balls numbered consecutively and a white cue ball are used.

Pool Game: Problem

Our goal is to sort these balls in ascending order so that these can easily be placed on the table.

Pool Game: Problem

Let's consider the following balls for simplicity.

Pool Game: Problem

Since, we want to implement these algorithms on Computer and computers can only compare 2 elements in a unit time, therefore, we will only see/compare 2

elements at a time.

In-Place VS Out-Of-Place Sorting

A sorting algorithm is **In-place** if the algorithm does not use extra space for manipulating the input but may require a small though non-constant extra space for its operation.
Or we can say, a sorting algorithm sorts **in-place** if only a constant number of elements of the input array are ever stored outside the array.

Stable VS Unstable Sorting

A sorting algorithm is stable if it does not change the order of elements with the same value.

Stable VS Unstable Sorting

A sorting algorithm is unstable if it changes the order of elements with the same value.

Solution 1

Sorting: Solution 1

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order.

Sorting: Solution 1

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order.

Sorting: Solution 1

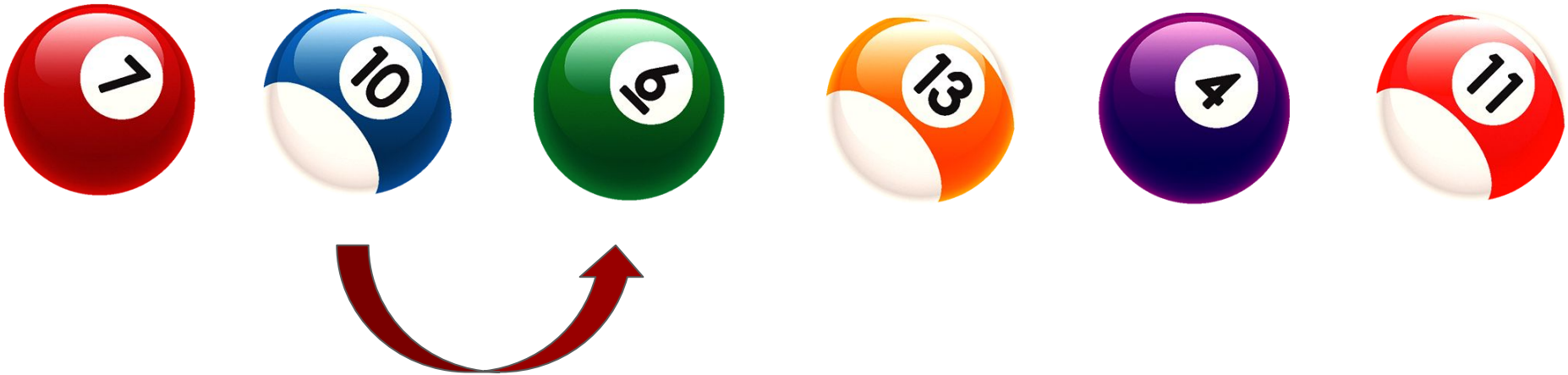
Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order.

Sorting: Solution 1

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order.

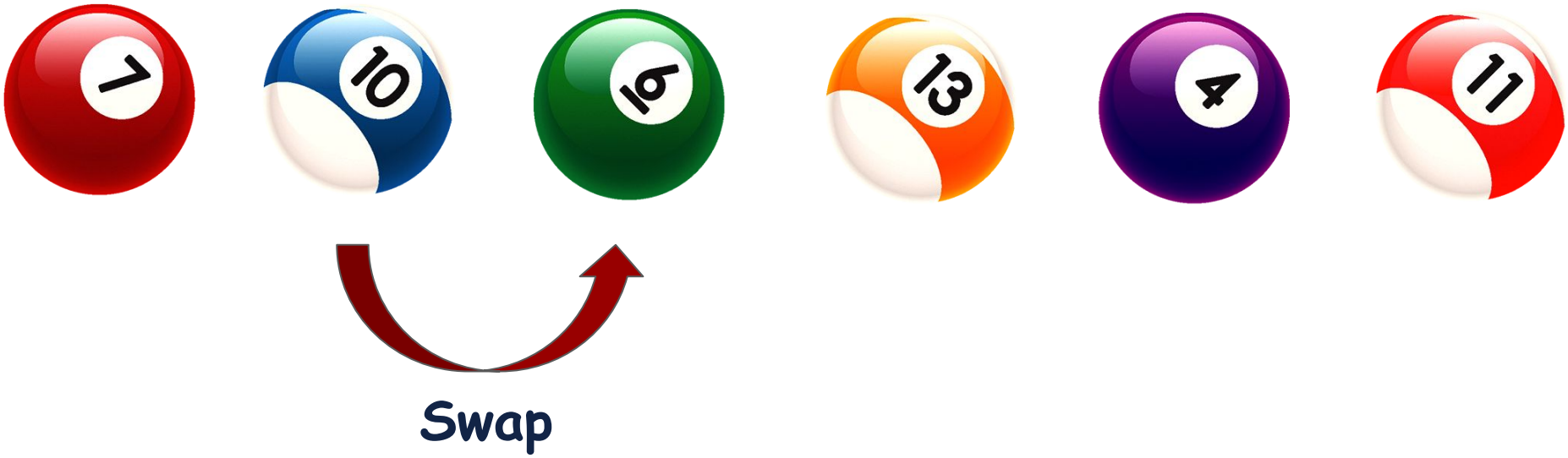
Sorting: Solution 1

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order.



Sorting: Solution 1

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order.

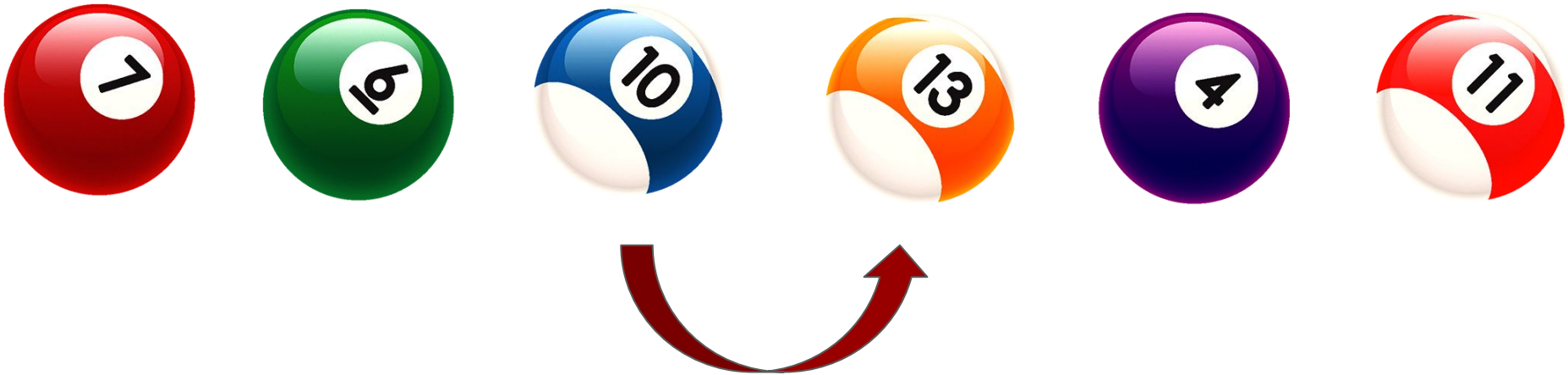


Sorting: Solution 1

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order.

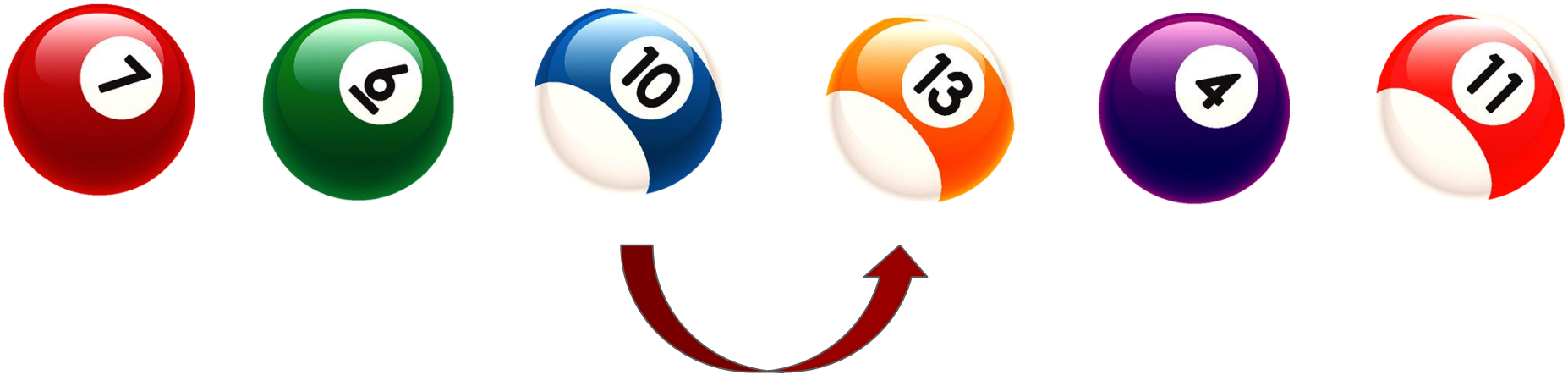
Sorting: Solution 1

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order.



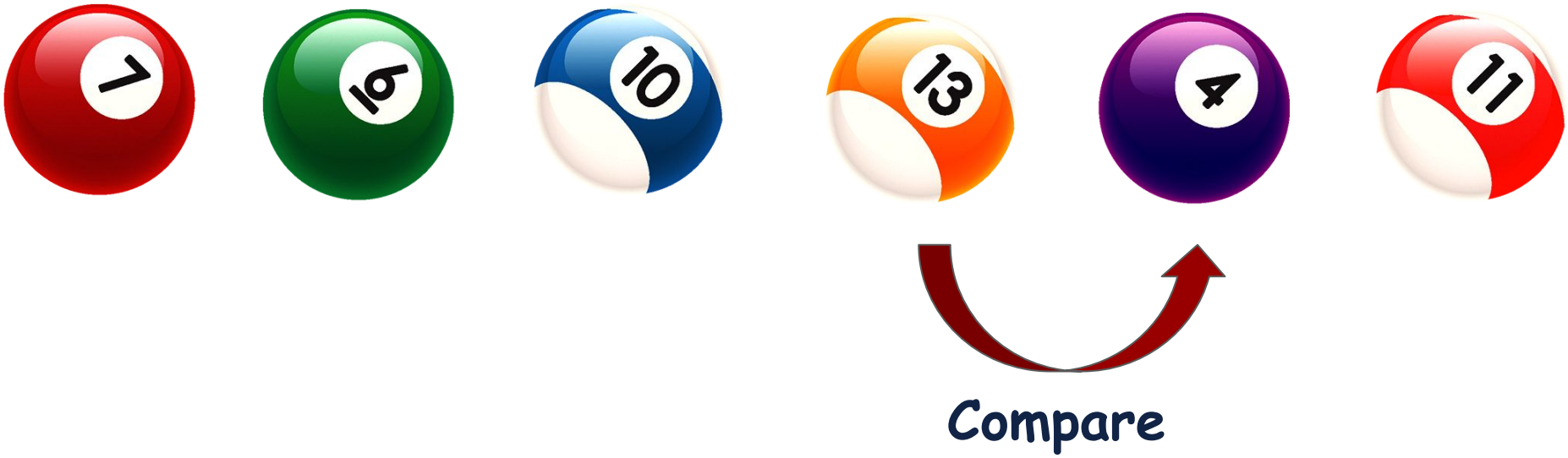
Sorting: Solution 1

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order.



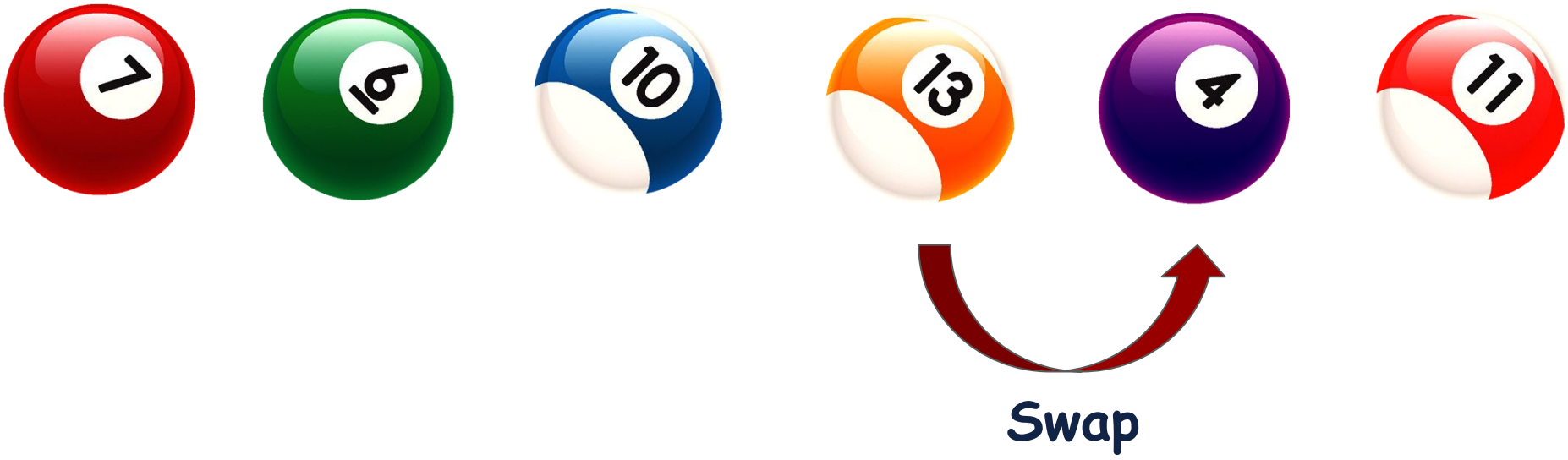
Sorting: Solution 1

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order.



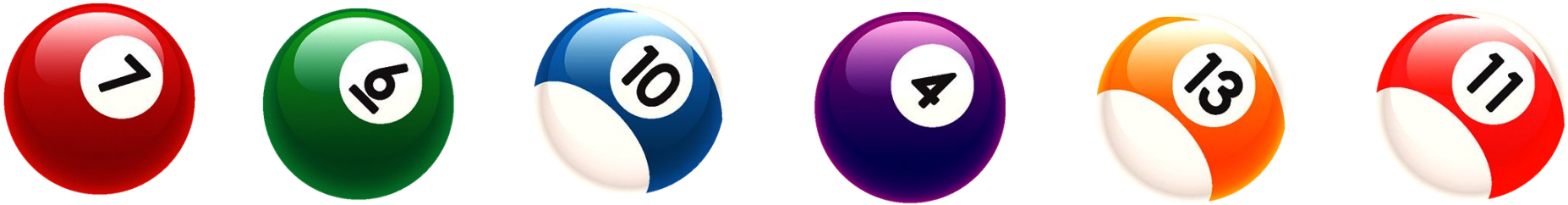
Sorting: Solution 1

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order.



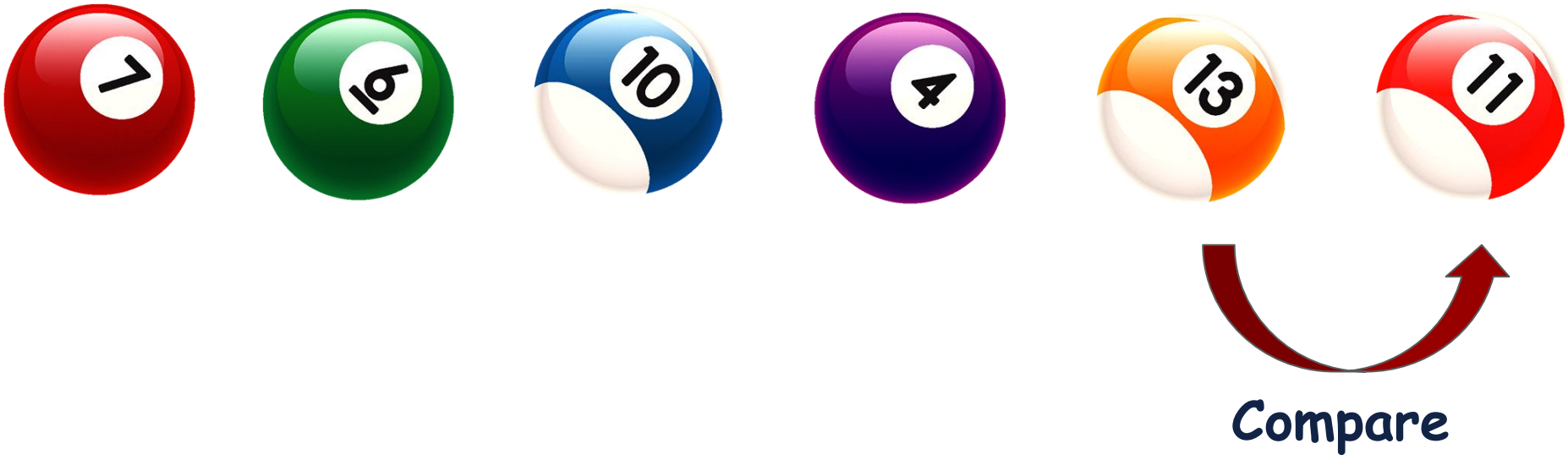
Sorting: Solution 1

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order.



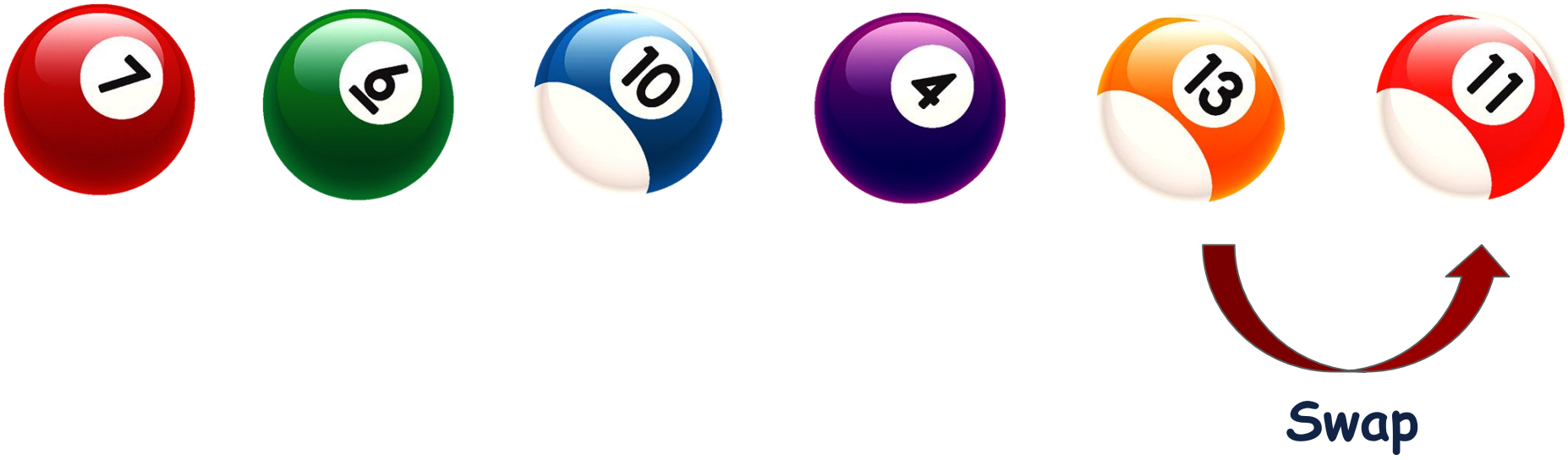
Sorting: Solution 1

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order.



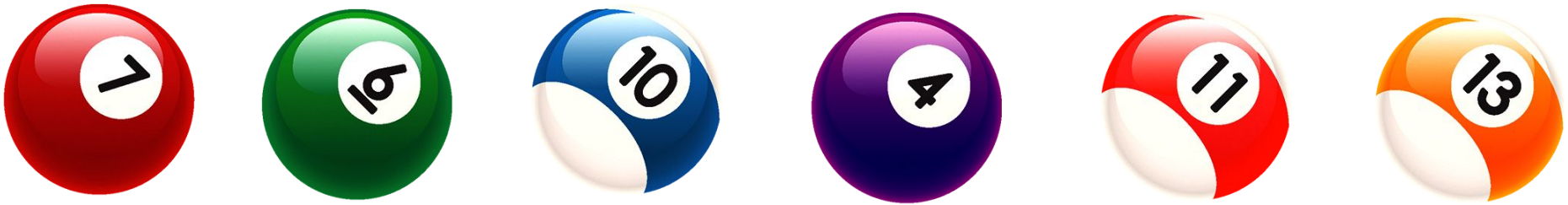
Sorting: Solution 1

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order.



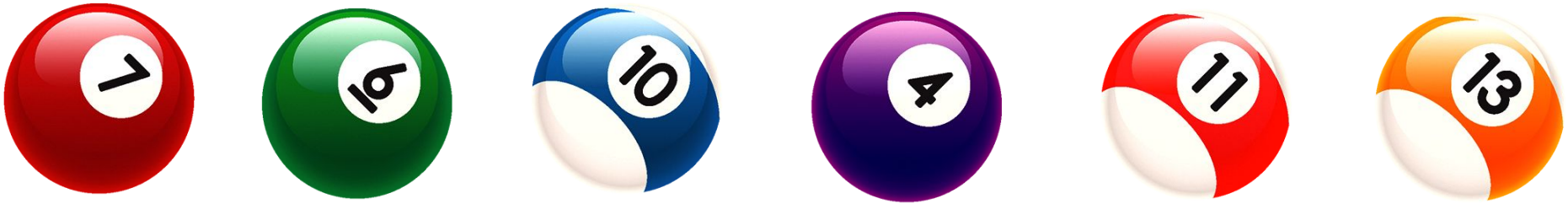
Sorting: Solution 1

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order.



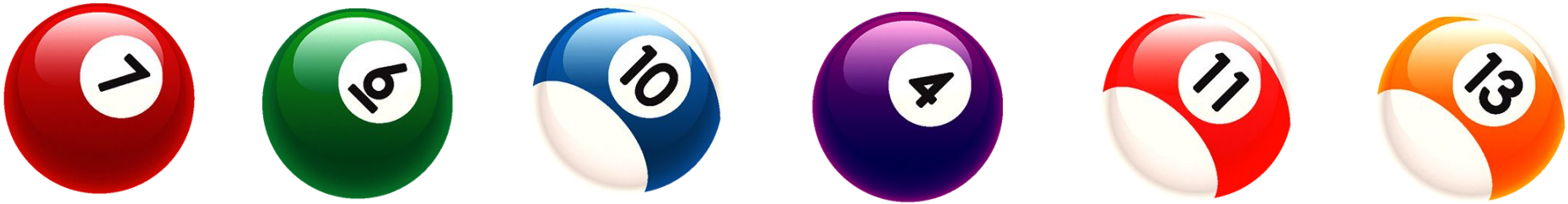
Sorting: Bubble Sort

After 1 pass of the loop, the largest element is bubbled at the end of the array. Therefore, this Algorithm of Sorting is called **Bubble Sort**.



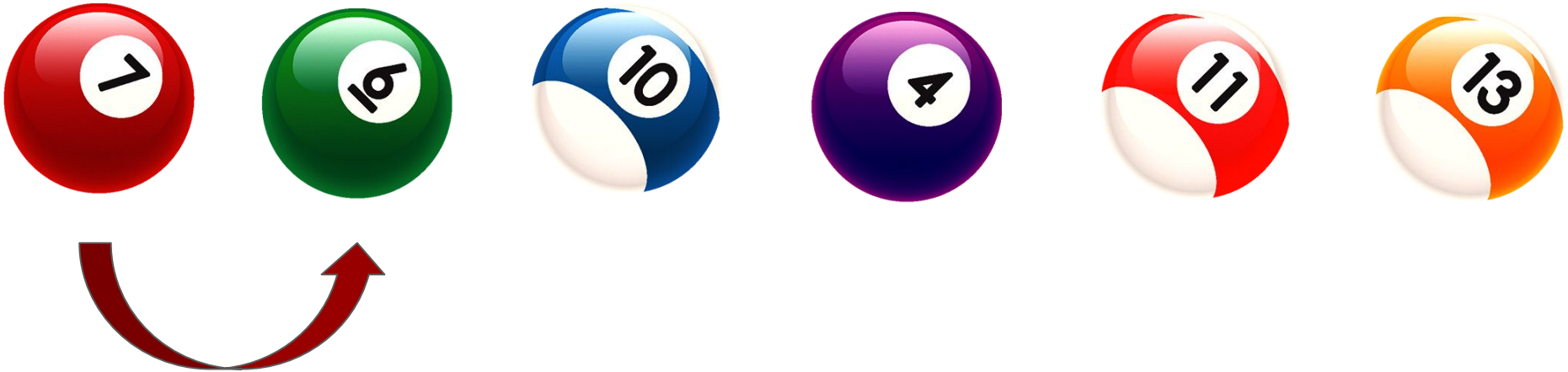
Bubble Sort: 2nd Pass

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order and don't compare with the last element now.



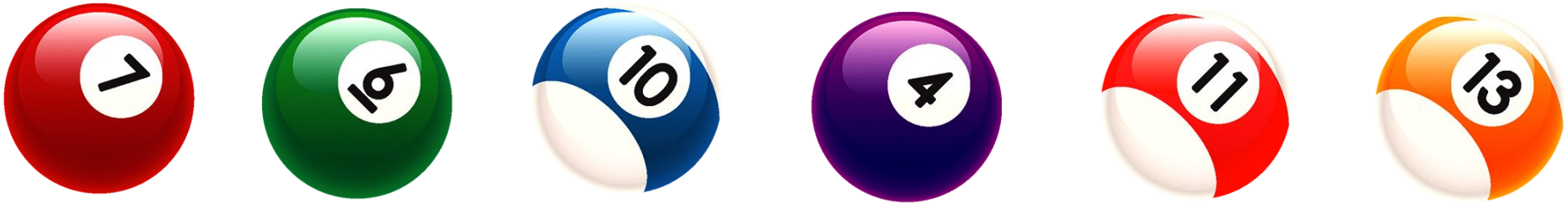
Bubble Sort: 2nd Pass

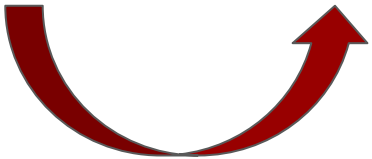
Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order and don't compare with the last element now.



Bubble Sort: 2nd Pass

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order and don't compare with the last element now.

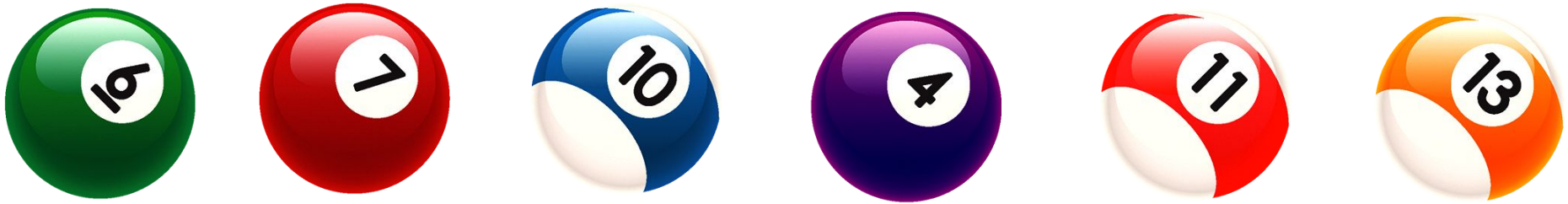




Swap

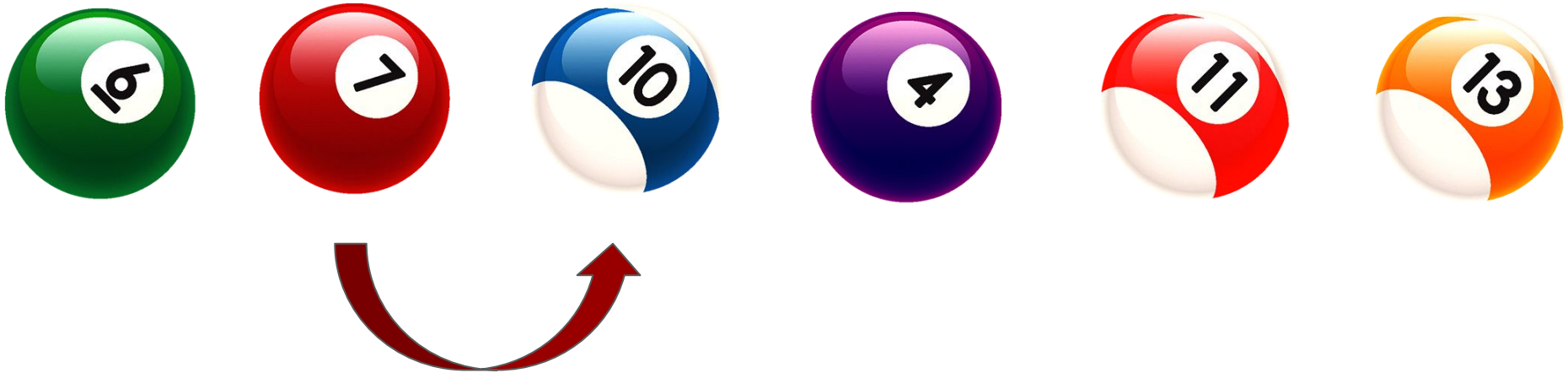
Bubble Sort: 2nd Pass

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order and don't compare with the last element now.



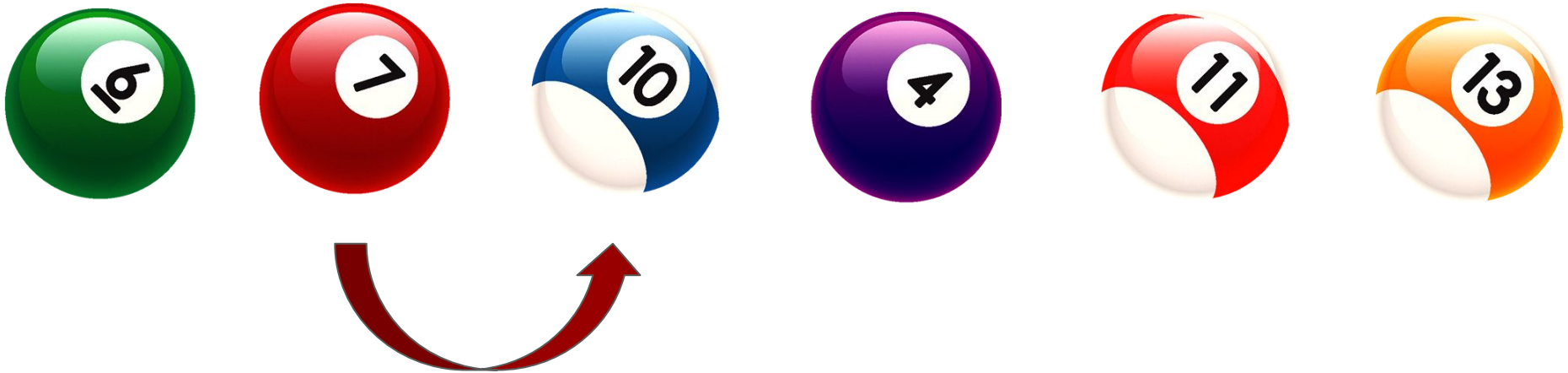
Bubble Sort: 2nd Pass

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order and don't compare with the last element now.



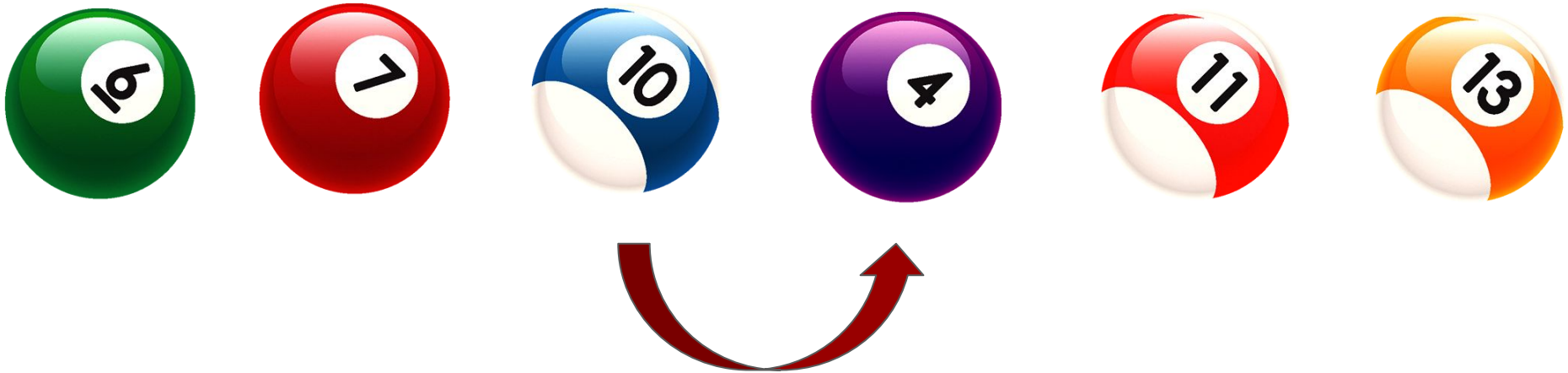
Bubble Sort: 2nd Pass

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order and don't compare with the last element now.



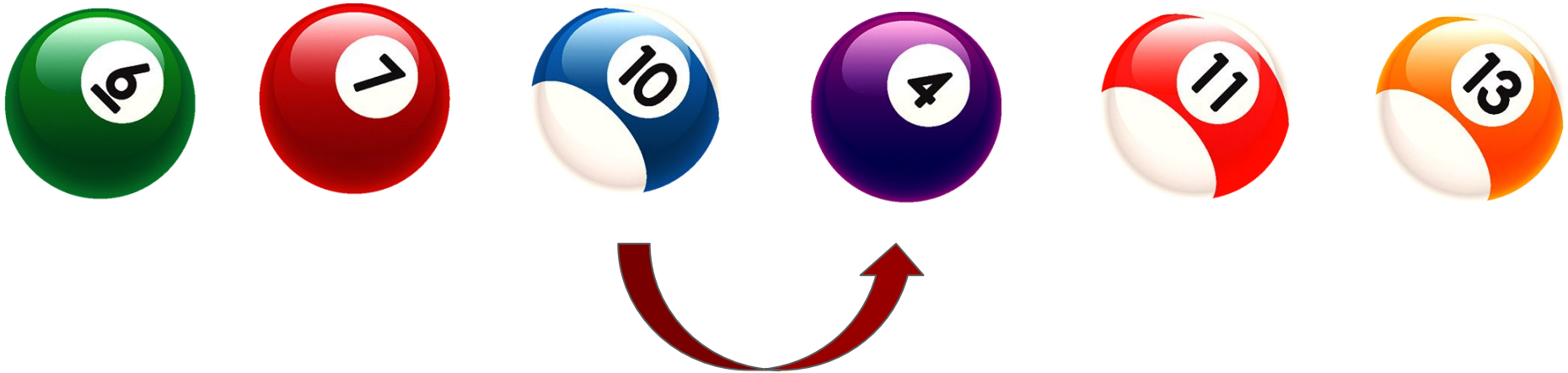
Bubble Sort: 2nd Pass

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order and don't compare with the last element now.



Bubble Sort: 2nd Pass

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order and don't compare with the last element now.

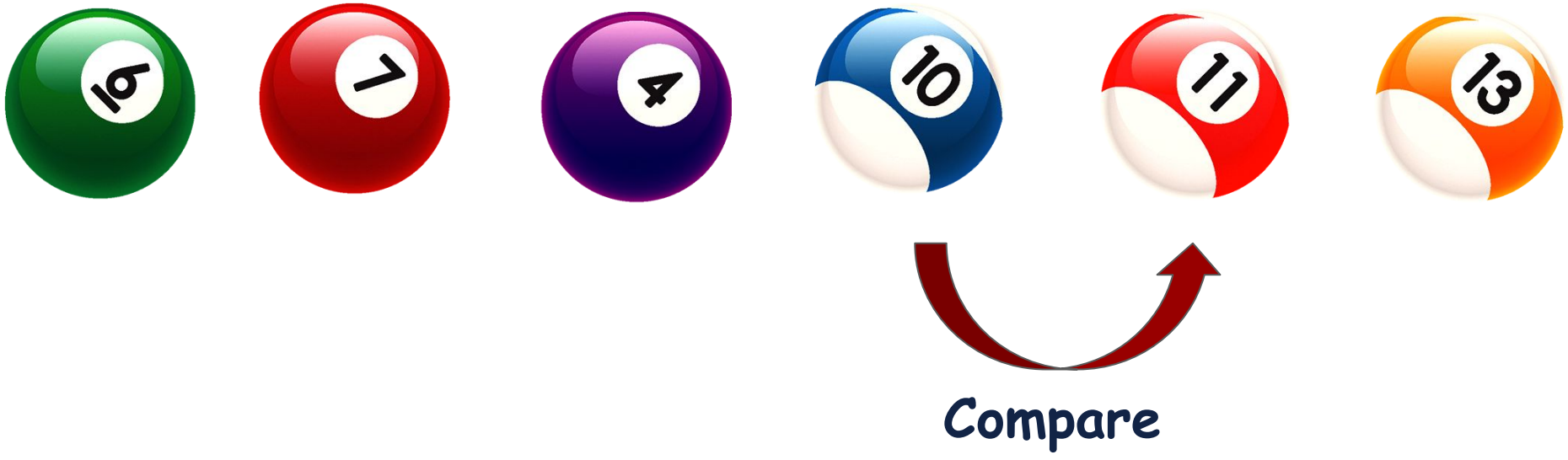


Bubble Sort: 2nd Pass

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order and don't compare with the last element now.

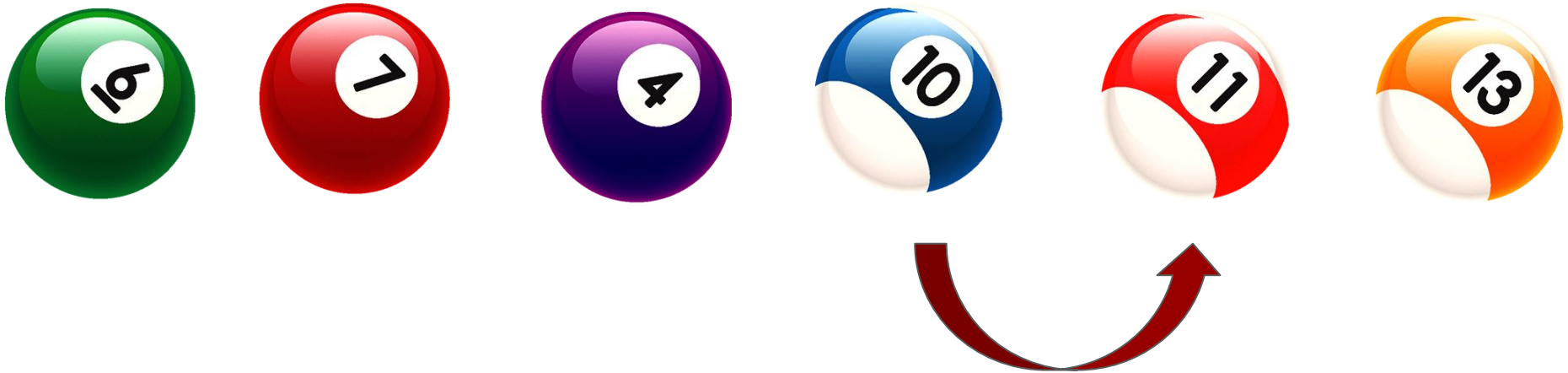
Bubble Sort: 2nd Pass

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order and don't compare with the last element now.



Bubble Sort: 2nd Pass

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order and don't compare with the last element now.

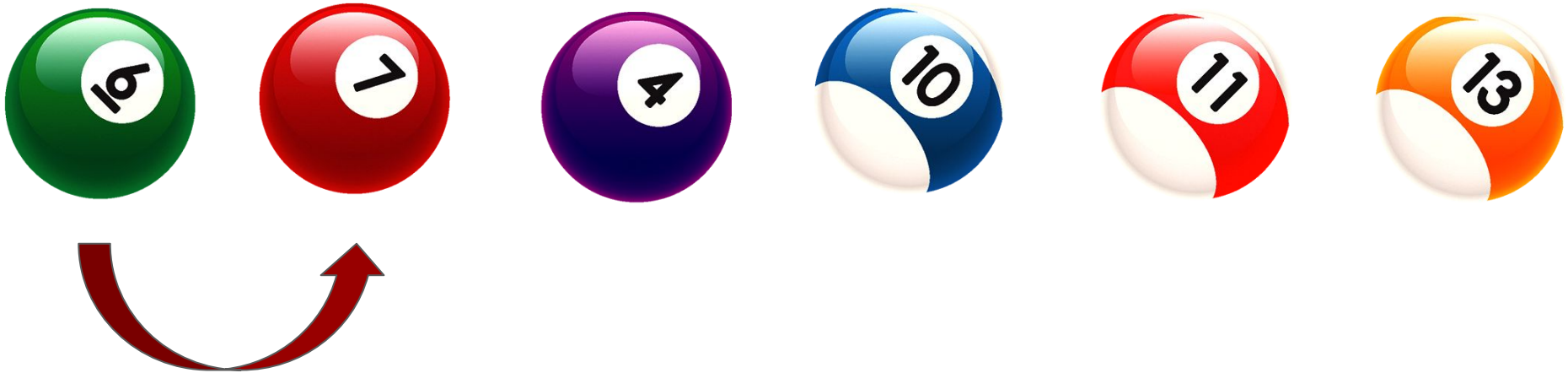


Bubble Sort: 3rd Pass

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order and don't compare with the last two elements now.

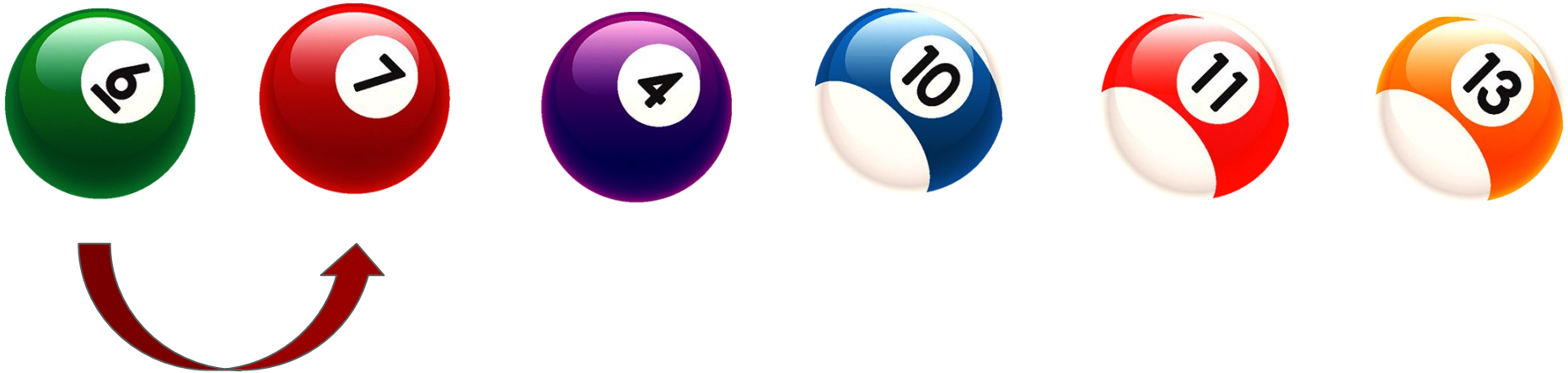
Bubble Sort: 3rd Pass

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order and don't compare with the last two elements now.



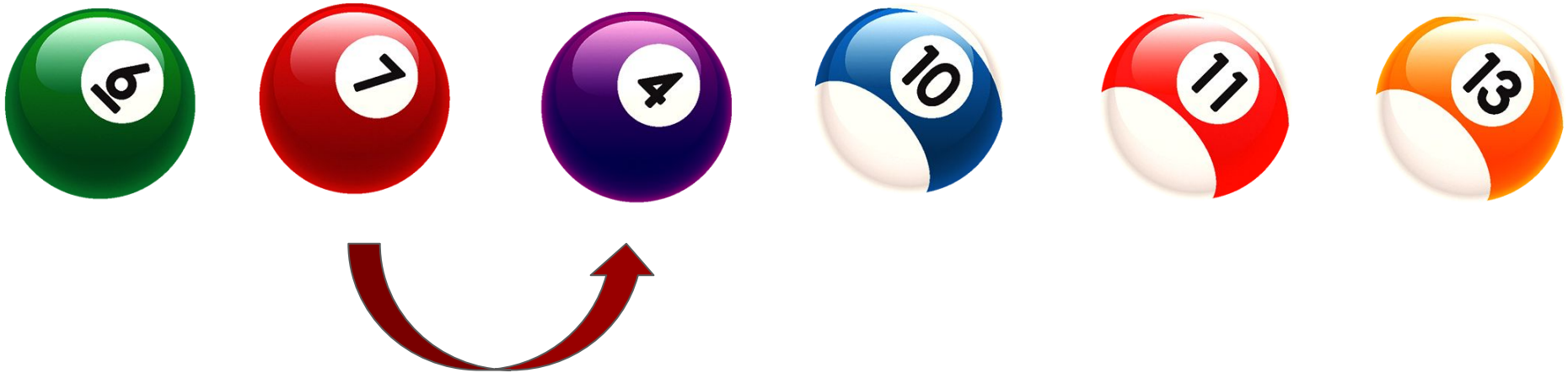
Bubble Sort: 3rd Pass

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order and don't compare with the last two elements now.



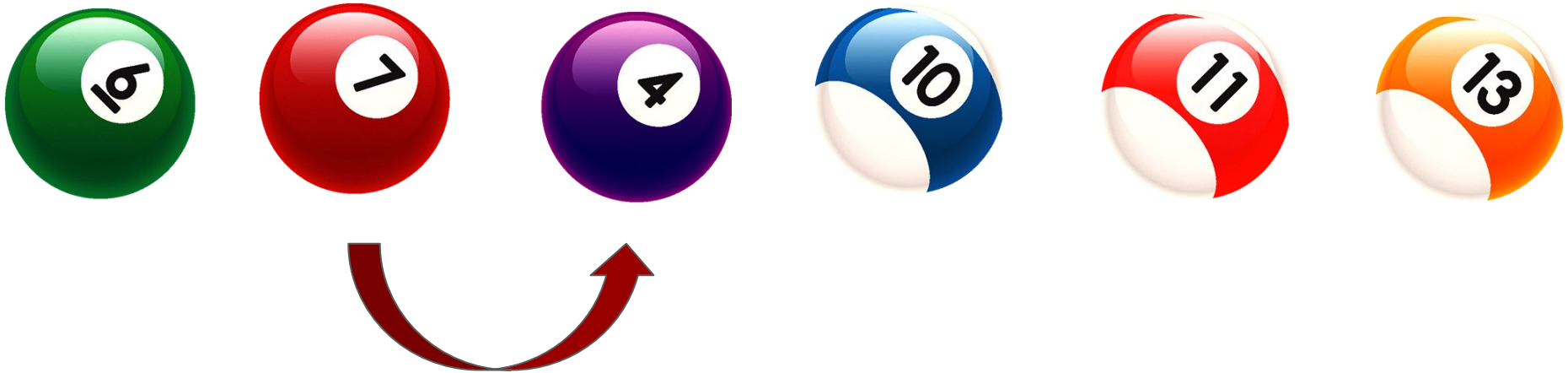
Bubble Sort: 3rd Pass

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order and don't compare with the last two elements now.



Bubble Sort: 3rd Pass

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order and don't compare with the last two elements now.

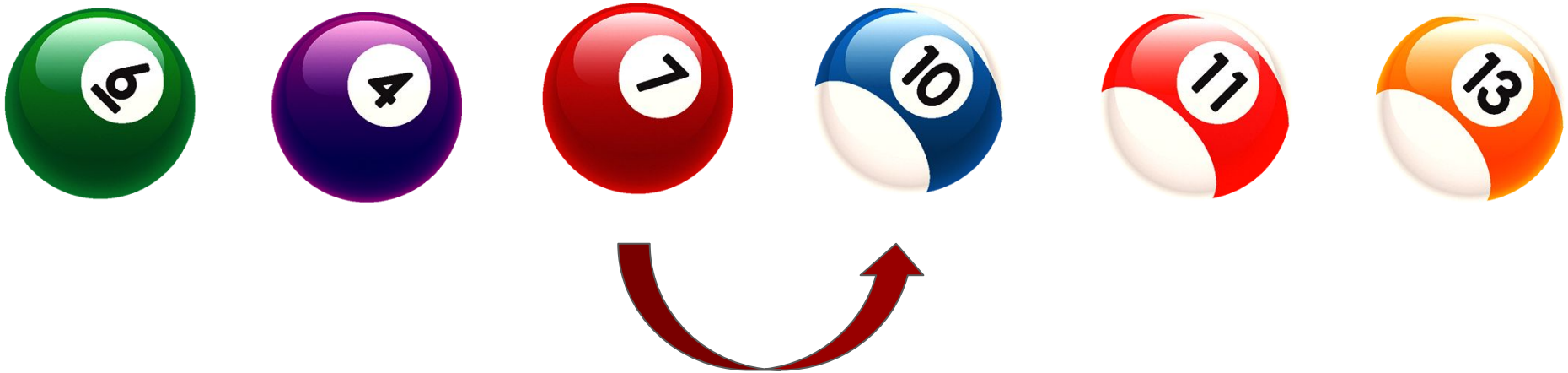


Bubble Sort: 3rd Pass

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order and don't compare with the last two elements now.

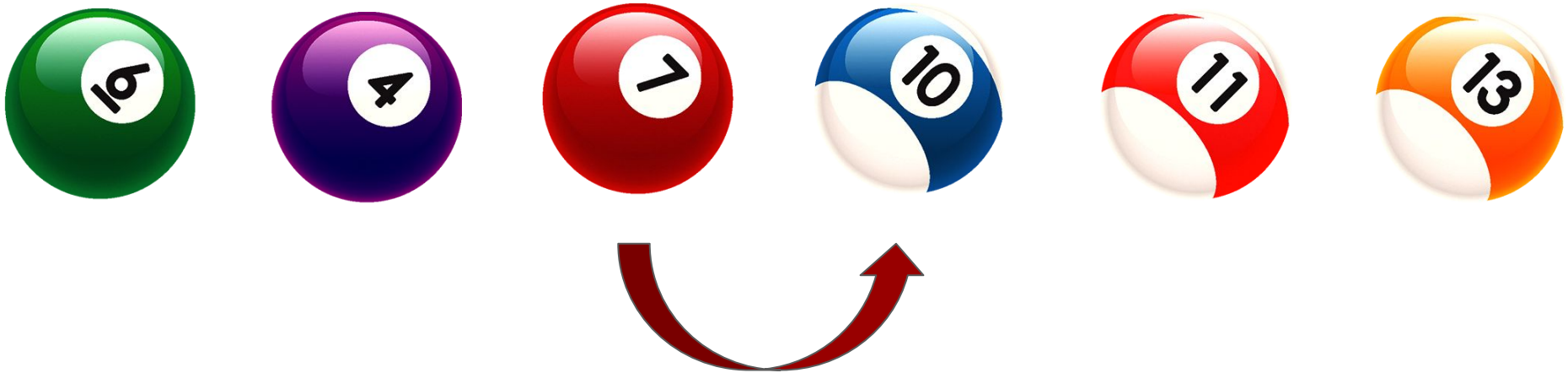
Bubble Sort: 3rd Pass

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order and don't compare with the last two elements now.



Bubble Sort: 3rd Pass

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order and don't compare with the last two elements now.

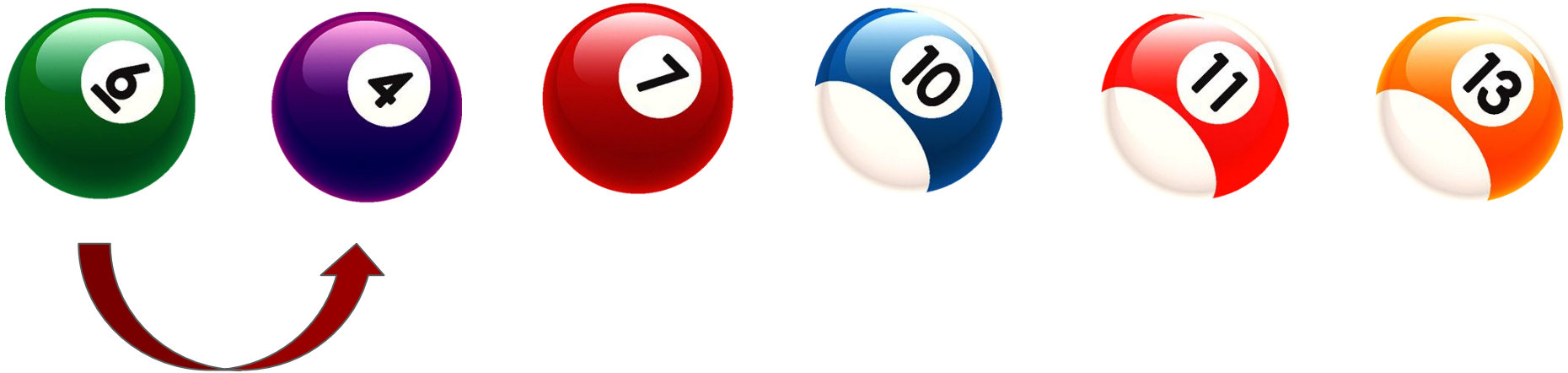


Bubble Sort: 4th Pass

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order and don't compare with the last three elements now.

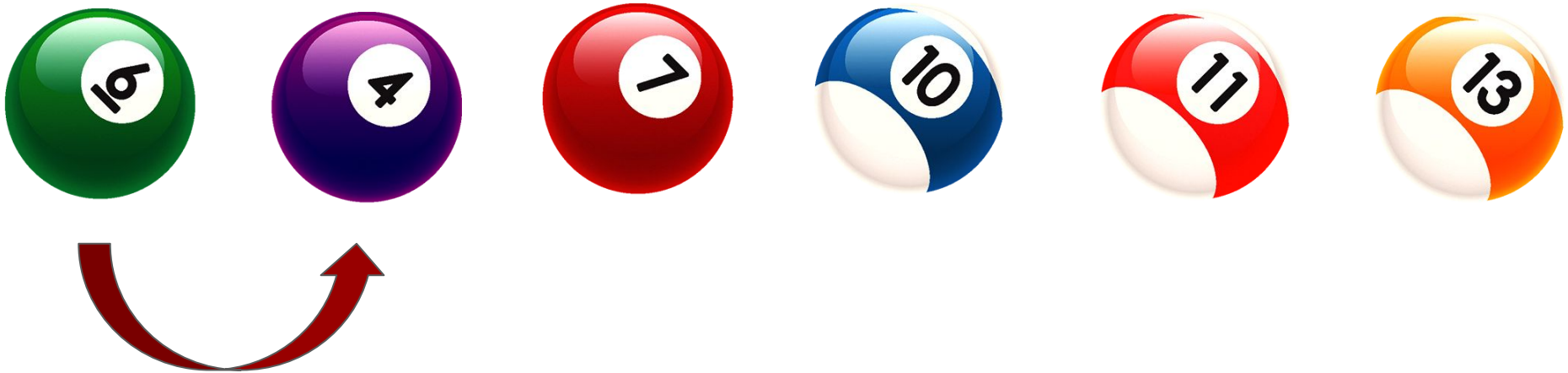
Bubble Sort: 4th Pass

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order and don't compare with the last three elements now.



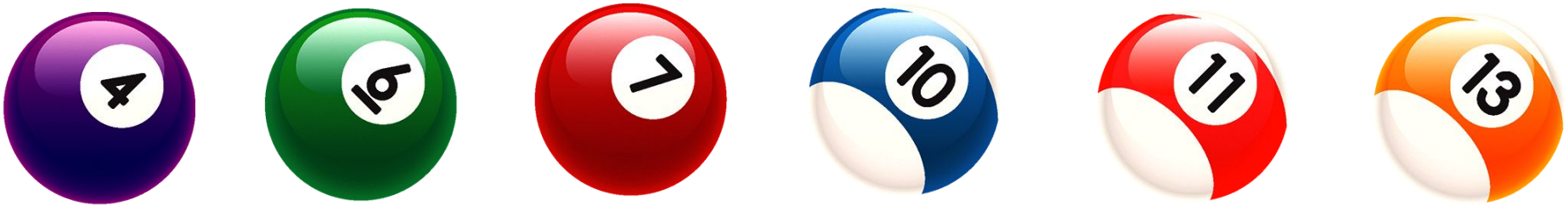
Bubble Sort: 4th Pass

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order and don't compare with the last three elements now.



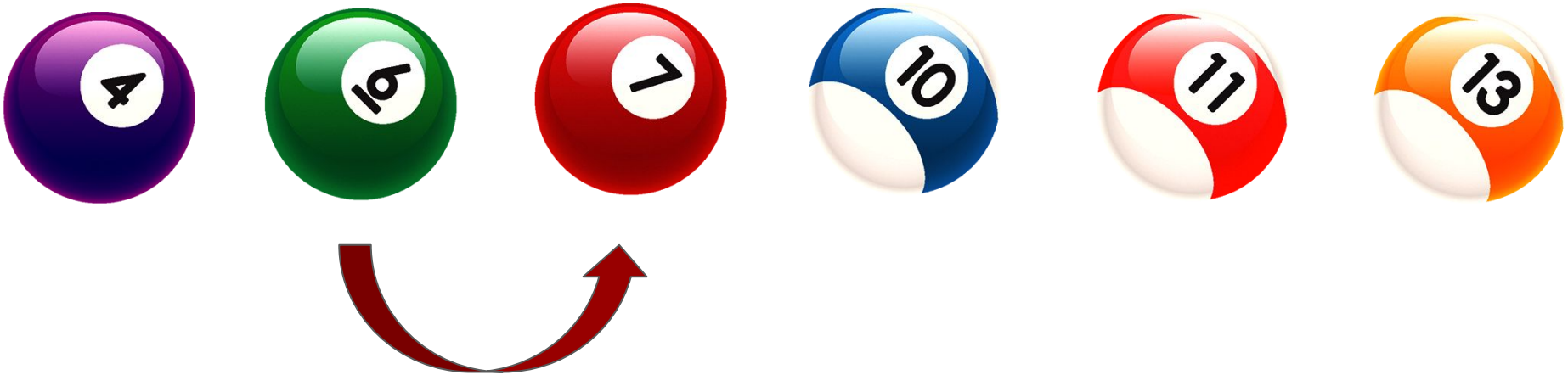
Bubble Sort: 4th Pass

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order and don't compare with the last three elements now.



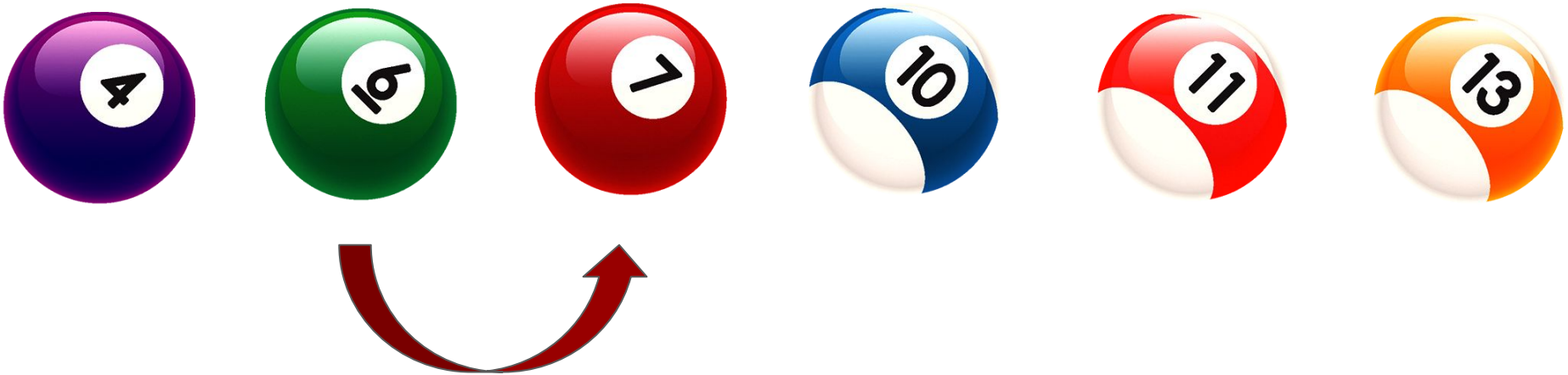
Bubble Sort: 4th Pass

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order and don't compare with the last three elements now.



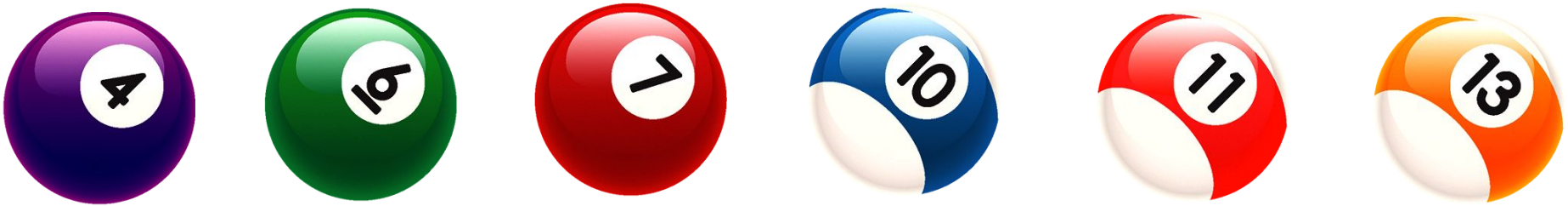
Bubble Sort: 4th Pass

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order and don't compare with the last three elements now.



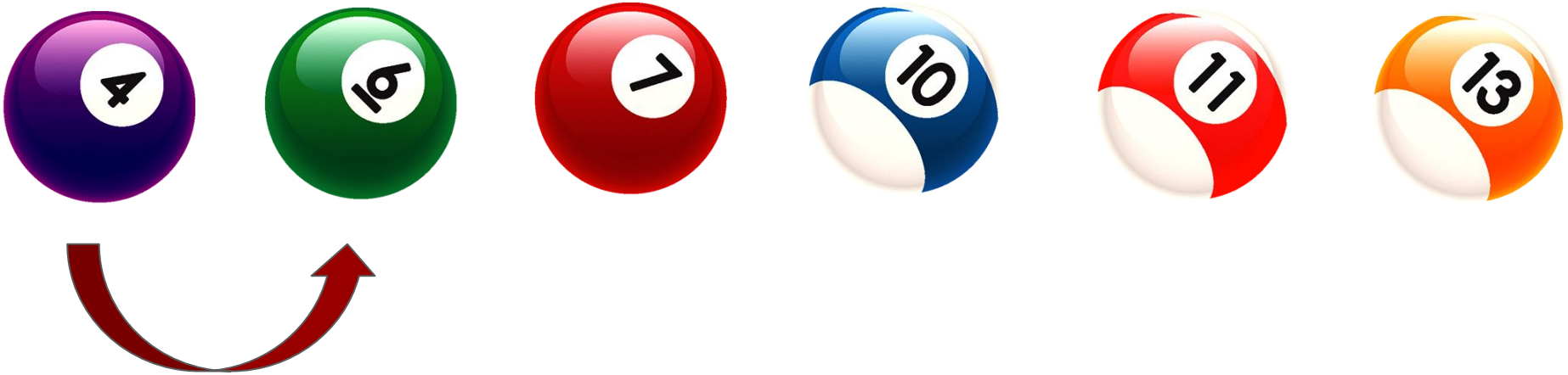
Bubble Sort: 5th Pass

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order and don't compare with the last four elements now.



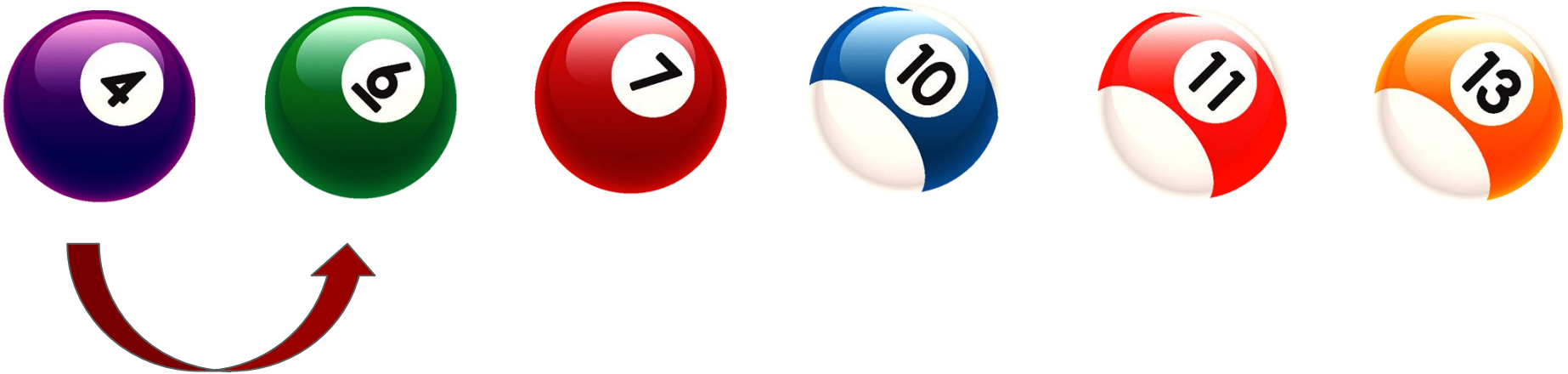
Bubble Sort: 5th Pass

Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order and don't compare with the last four elements now.



Bubble Sort: 5th Pass

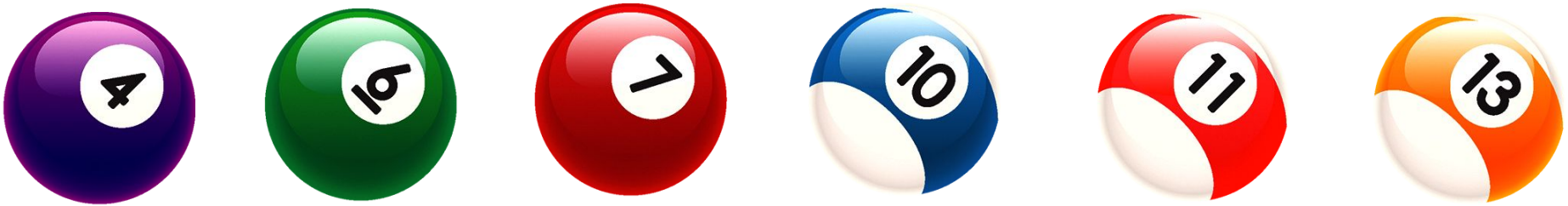
Lets compare the each pair of adjacent elements and swap the elements if they are not in ascending order and don't compare with the last four elements now.



No Swap

Bubble Sort: 5th Pass

After n-1 passes on the Array, the data is sorted.



Bubble Sort: Implementation

Let's implement the algorithm now.

Bubble Sort: Implementation

```
main(){
    int arr[6] = {10, 7, 6, 13, 4, 11};
    bubbleSort(arr, 6);
    for (int x = 0; x < 6; x++)
    {
        cout << arr[x] << " ";
    }
}
```

```
void swap(int &a, int &b){
    int c;
    c = a;
    a = b;
    b = c;
}
```

```
void bubbleSort(int arr[], int n)
{
    for (int x = 0; x < n - 1; x++)
    {
        for (int y = 0; y < n - x - 1; y++)
        {
            if (arr[y] > arr[y + 1])
            {
                swap(arr[y], arr[y + 1]);
            }
        }
    }
}
```

Bubble Sort: Time Complexity

What is the Time Complexity?

```
main(){
    int arr[6] = {10, 7, 6, 13, 4, 11};
    bubbleSort(arr, 6);
    for (int x = 0; x < 6; x++)
    {
        cout << arr[x] << " ";
    }
}
```

```
void swap(int &a, int &b){
    int c;
    c = a;
    a = b;
    b = c;
}
```

```
void bubbleSort(int arr[], int n)
{
    for (int x = 0; x < n - 1; x++)
    {
        for (int y = 0; y < n - x - 1; y++)
        {
            if (arr[y] > arr[y + 1])
            {
                swap(arr[y], arr[y + 1]);
            }
        }
    }
}
```

Bubble Sort: Time Complexity

What is the Time Complexity?

```
main(){
    int arr[6] = {10, 7, 6, 13, 4, 11};
    bubbleSort(arr, 6);
    for (int x = 0; x < 6; x++)
    {
        cout << arr[x] << " ";
    }
}
```

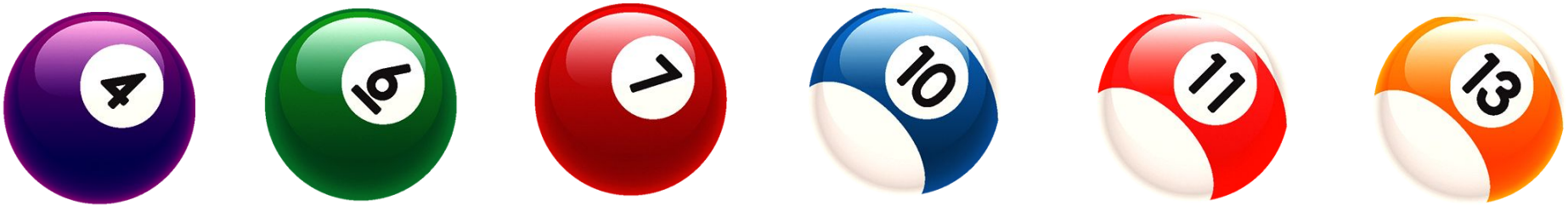
```
void swap(int &a, int &b){
    int c;
    c = a;
    a = b;
    b = c;
}
```

```
void bubbleSort(int arr[], int n)
{
    for (int x = 0; x < n - 1; x++)
    {
        for (int y = 0; y < n - x - 1; y++)
        {
            if (arr[y] > arr[y + 1])
            {
                swap(arr[y], arr[y + 1]);
            }
        }
    }
}
```

$O(n^2)$

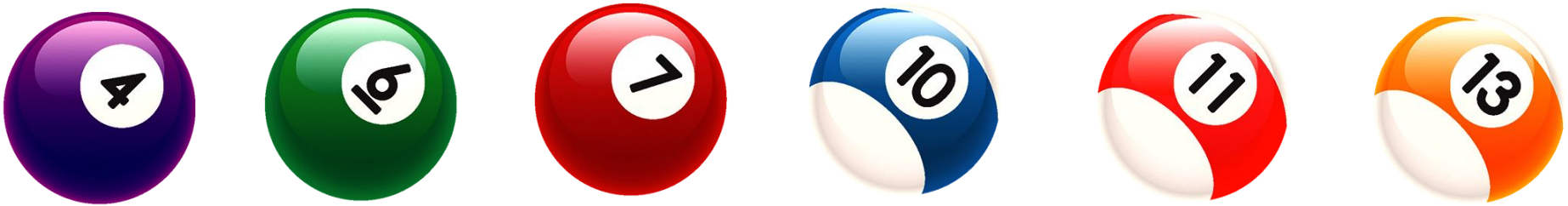
Bubble Sort: Point to Ponder

What if the data is already given in sorted order?
How many iterations will this algorithm perform?



Bubble Sort: Point to Ponder

What if the data is already given in sorted order?
How many iterations will this algorithm perform?

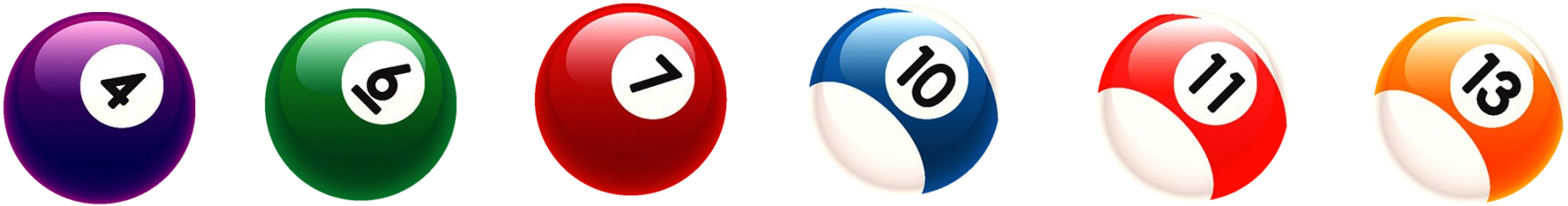


Both loops will run the same way. Just the swaps will not be made.

Best Case Time Complexity: $O(n^2)$

Bubble Sort: Point to Ponder

Can we update the previous Algorithm so that our time complexity for the best case improves from $O(n^2)$ to $O(n)$?



Bubble Sort: Can we improve?

```
main(){
    int arr[6] = {10, 7, 6, 13, 4, 11};
    bubbleSort(arr, 6);
    for (int x = 0; x < 6; x++)
    {
        cout << arr[x] << " ";
    }
}
```

```
void swap(int &a, int &b){
    int c;
    c = a;
    a = b;
    b = c;
}
```

```
void bubbleSort(int arr[], int n)
{
    for (int x = 0; x < n - 1; x++)
    {
        for (int y = 0; y < n - x - 1; y++)
        {
            if (arr[y] > arr[y + 1])
            {
                swap(arr[y], arr[y + 1]);
            }
        }
    }
}
```

Bubble Sort:

```
main(){
    int arr[6] = {10, 7, 6, 13, 4, 11};
    bubbleSort(arr, 6);
    for (int x = 0; x < 6; x++)
    {
        cout << arr[x] << " ";
    }
}
```

```
void swap(int &a, int &b){
    int c;
    c = a;
    a = b;
    b = c;
}
```

```
void bubbleSort(int arr[], int n)
{
    for (int x = 0; x < n - 1; x++)
    {
        bool isSwapped = false;
        for (int y = 0; y < n - x - 1; y++)
        {
            if (arr[y] > arr[y + 1])
            {
                swap(arr[y], arr[y + 1]);
                isSwapped = true;
            }
        }
        if (!isSwapped)
        {
            break;
        }
    }
}
```

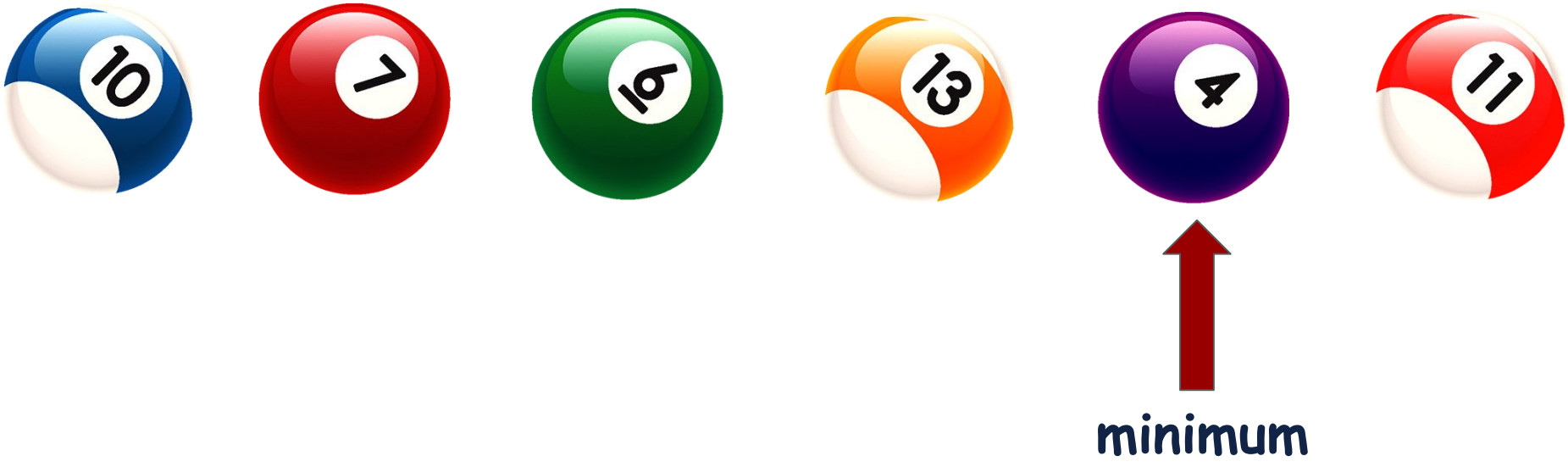
Solution 2

Sorting: Solution 2

Let's find the minimum element and swap it with the first element.

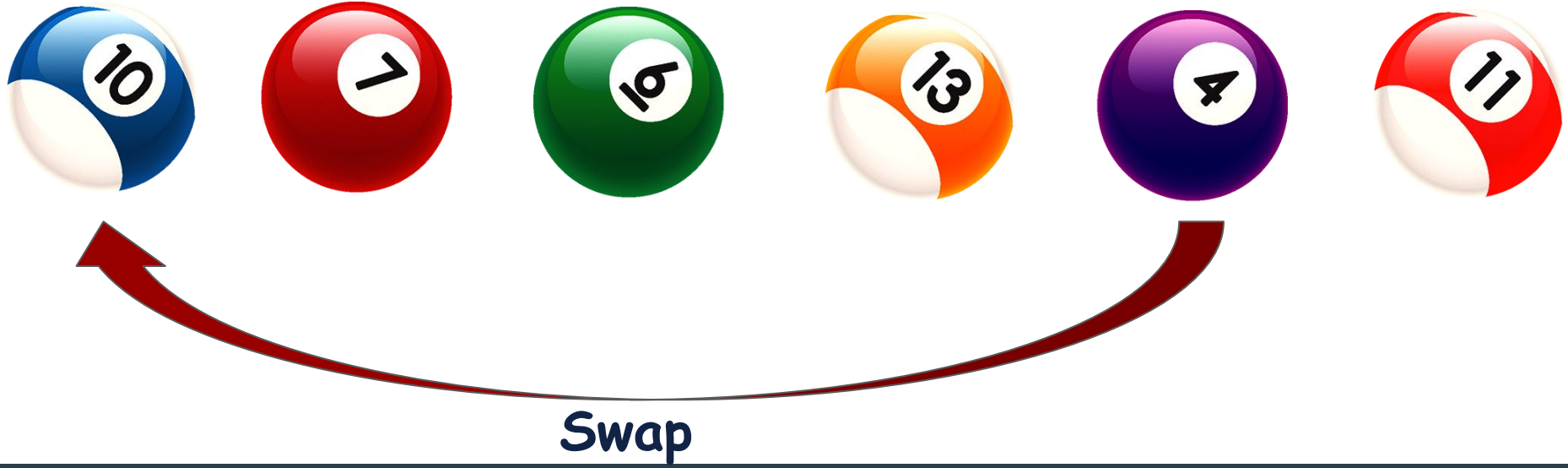
Sorting: Solution 2

Let's find the minimum element and swap it with the first element.



Sorting: Solution 2

Let's find the minimum element and swap it with the first element.



Sorting: Solution 2

Let's find the minimum element and swap it with the first element.

Sorting: Selection Sort

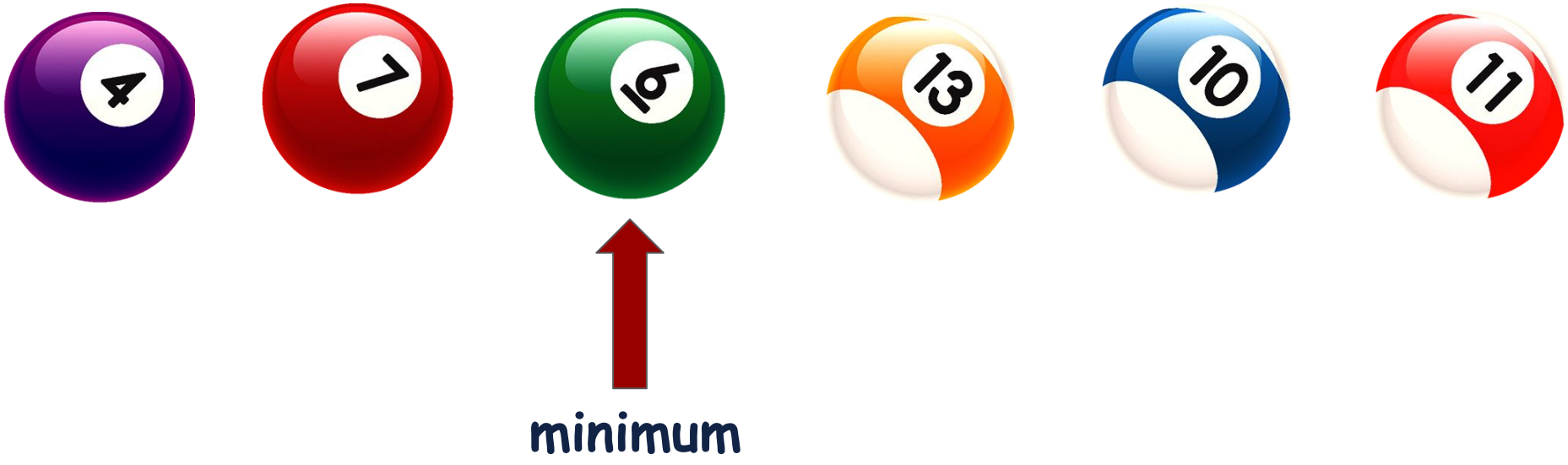
In this Algorithm we are selecting the minimum element and then swapping it with the element on the specific index. Therefore, it is called as **Selection Sort**.

Selection Sort: 2nd Pass

Let's find the minimum element starting from the second element and swap it with the second element.

Selection Sort: 2nd Pass

Let's find the minimum element starting from the second element and swap it with the second element.



Selection Sort: 2nd Pass

Let's find the minimum element starting from the second element and swap it with the second element.

Selection Sort: 2nd Pass

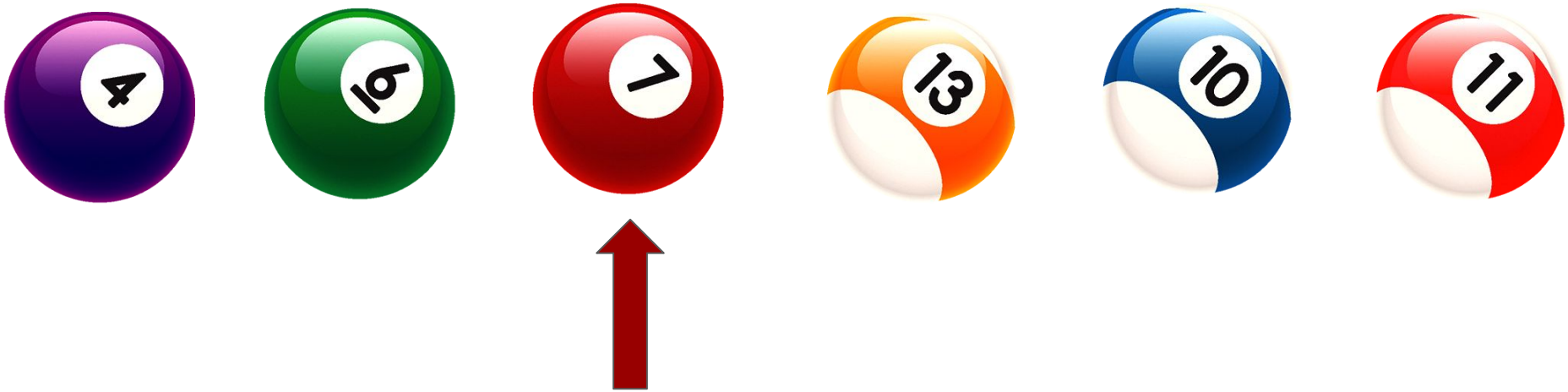
Let's find the minimum element starting from the second element and swap it with the second element.

Selection Sort: 3rd Pass

Let's find the minimum element starting from the third element and swap it with the third element.

Selection Sort: 3rd Pass

Let's find the minimum element starting from the third element and swap it with the third element.



Selection Sort: 3rd Pass

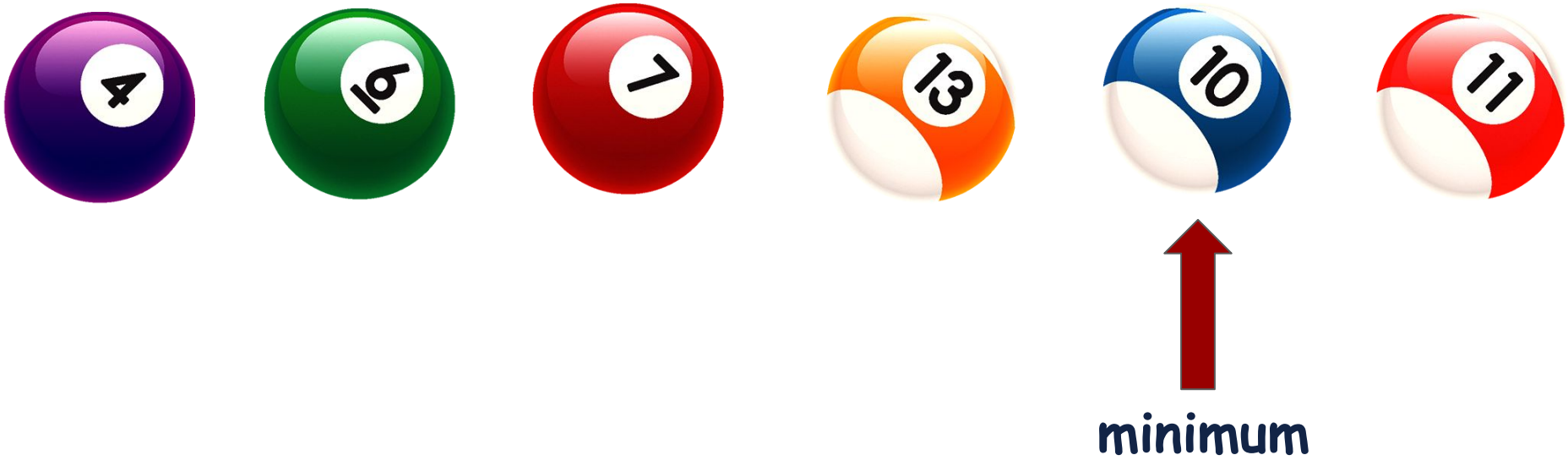
Let's find the minimum element starting from the third element and swap it with the third element.

Selection Sort: 4th Pass

Let's find the minimum element starting from the fourth element and swap it with the fourth element.

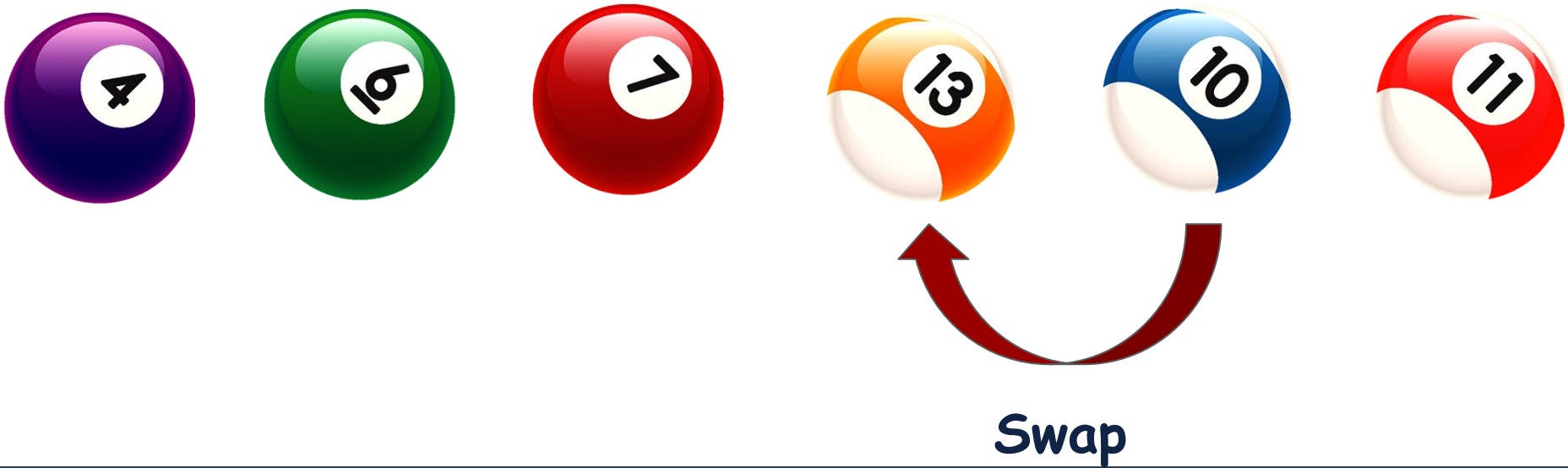
Selection Sort: 4th Pass

Let's find the minimum element starting from the fourth element and swap it with the fourth element.



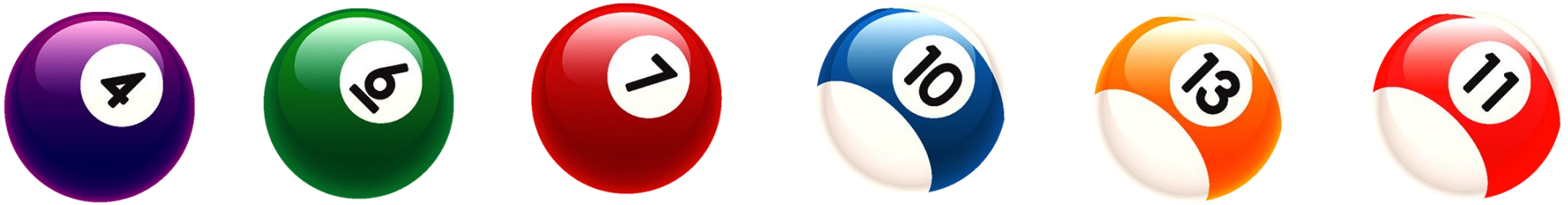
Selection Sort: 4th Pass

Let's find the minimum element starting from the fourth element and swap it with the fourth element.



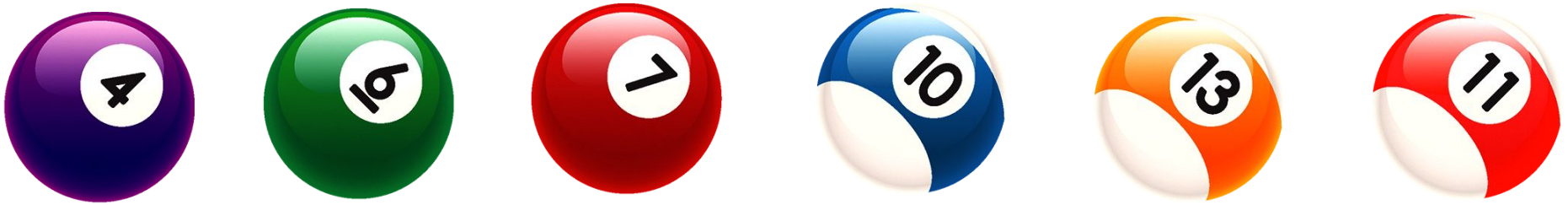
Selection Sort: 4th Pass

Let's find the minimum element starting from the fourth element and swap it with the fourth element.



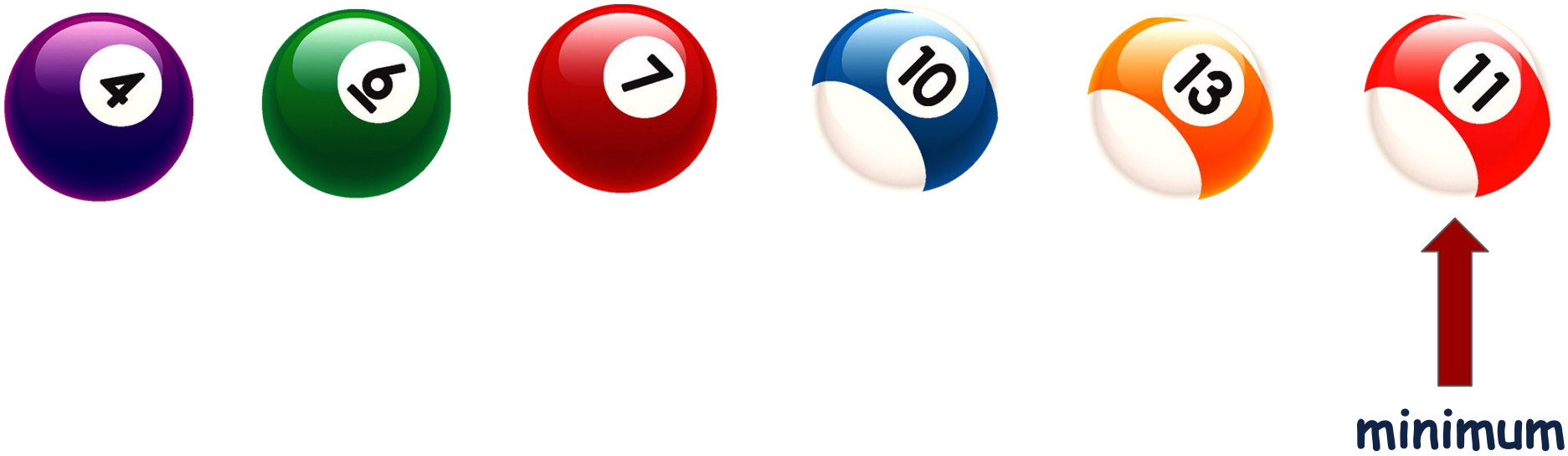
Selection Sort: 5th Pass

Let's find the minimum element starting from the fifth element and swap it with the fifth element.



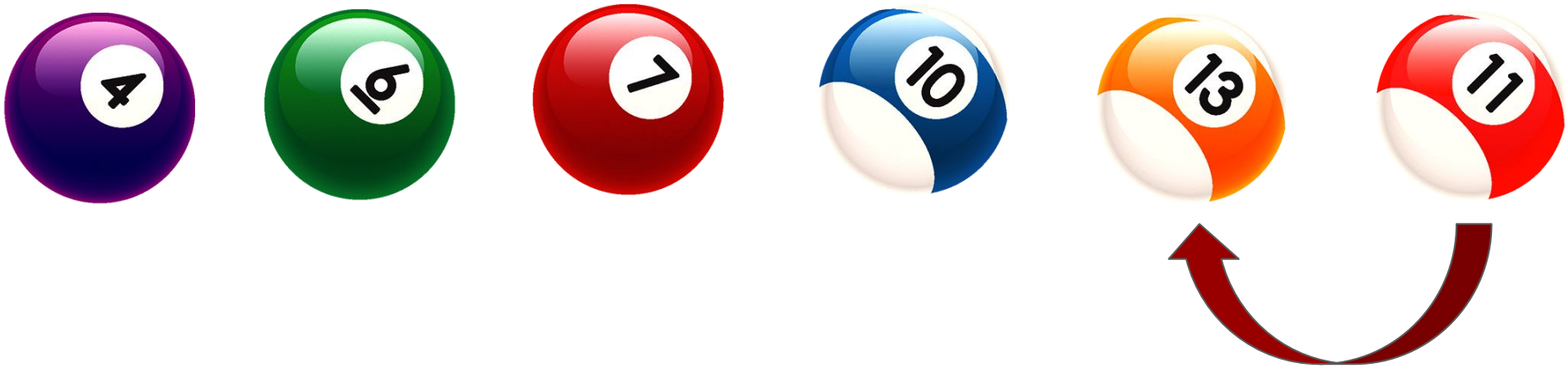
Selection Sort: 5th Pass

Let's find the minimum element starting from the fifth element and swap it with the fifth element.



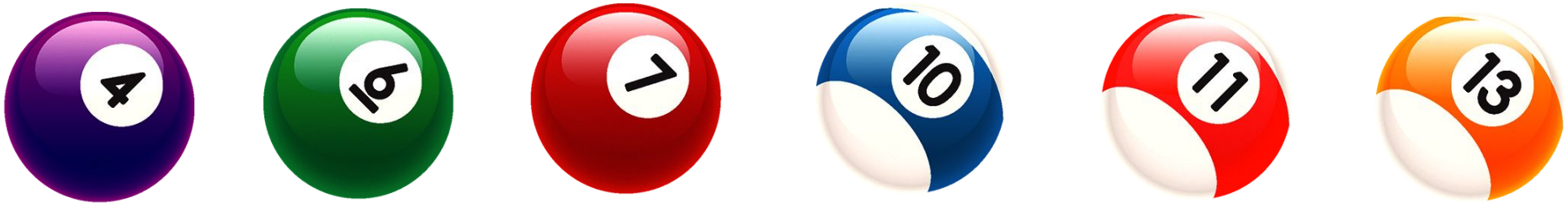
Selection Sort: 5th Pass

Let's find the minimum element starting from the fifth element and swap it with the fifth element.



Selection Sort: 5th Pass

Let's find the minimum element starting from the fifth element and swap it with the fifth element.



Selection Sort: Implementation

Let's implement the algorithm now.

Selection Sort: Implementation

```
main()
{
    int arr[6] = {10, 7, 6, 13, 4, 11};
    selectionSort(arr, 6);
    for (int x = 0; x < 6; x++)
    {
        cout << arr[x] << " ";
    }
}
```

```
void swap(int &a, int &b)
{
    int c;
    c = a;
    a = b;
    b = c;
}
```

Selection Sort

```
int findMinimum(int arr[], int start, int end)
{
    int min = arr[start];
    int minIndex = start;
    for(int x = start; x < end; x++)
    {
        if(min > arr[x])
        {
            min = arr[x];
            minIndex = x;
        }
    }
    return minIndex;
}
```

```
void selectionSort(int arr[], int n){
    for(int x = 0; x < n - 1; x++)
    {
        int minIndex = findMinimum(arr, x, n);
        swap(arr[x], arr[minIndex]);
    }
}
```

Selection Sort

What is the Time Complexity?

```
int findMinimum(int arr[], int start, int end)
{
    int min = arr[start];
    int minIndex = start;
    for(int x = start; x < end; x++)
    {
        if(min > arr[x])
        {
            min = arr[x];
            minIndex = x;
        }
    }
    return minIndex;
}
```

```
void selectionSort(int arr[], int n){
    for(int x = 0; x < n - 1; x++)
    {
        int minIndex = findMinimum(arr, x, n);
        swap(arr[x], arr[minIndex]);
    }
}
```

Selection Sort

What is the Time Complexity?

```
int findMinimum(int arr[], int start, int end)
{
    int min = arr[start];
    int minIndex = start;
    for(int x = start; x < end; x++)
    {
        if(min > arr[x])
        {
            min = arr[x];
            minIndex = x;
        }
    }
    return minIndex;
}
```

```
void selectionSort(int arr[], int n){
    for(int x = 0; x < n - 1; x++)
    {
        int minIndex = findMinimum(arr, x, n);
        swap(arr[x], arr[minIndex]);
    }
}
```

$O(n^2)$

Solution 3

Sorting: Solution 3

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.

Sorting: Solution 3

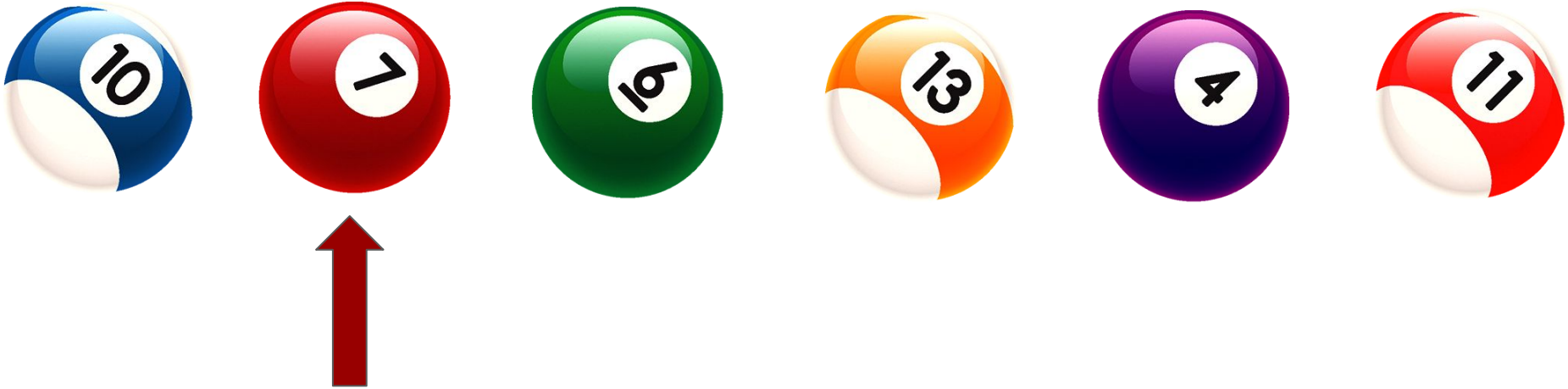
The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

sorted

unsorted

Sorting: Solution 3

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.



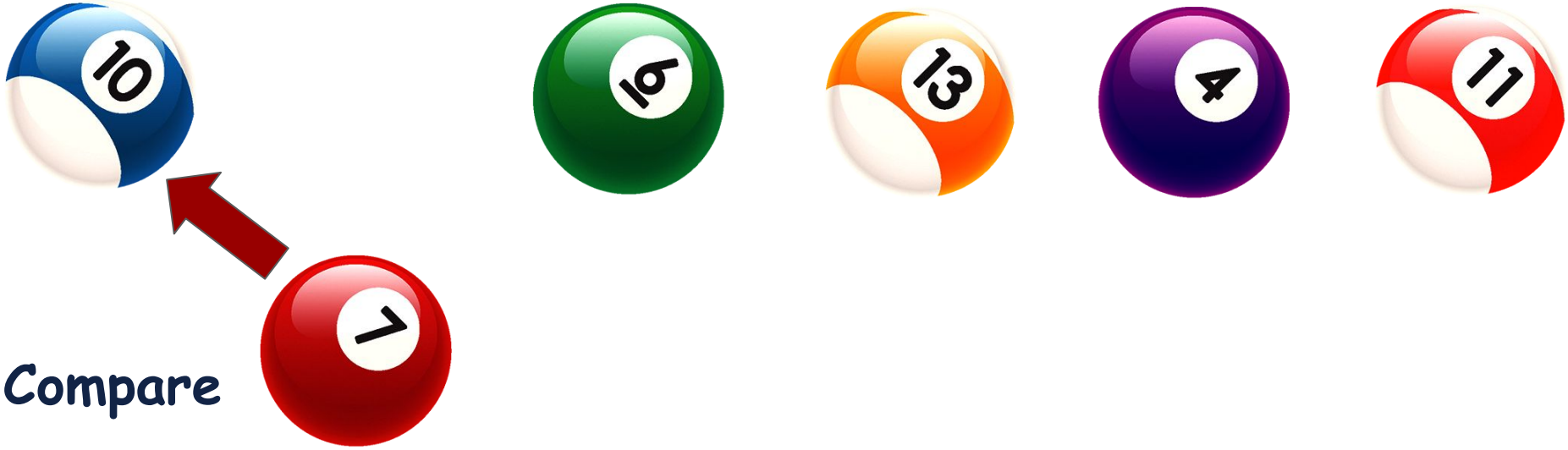
Sorting: Solution 3

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.

Store current element separately

Sorting: Solution 3

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.



Sorting: Solution 3

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.

Sorting: Solution 3

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.

Sorting: Solution 3

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.

Place at the correct location

Sorting: Solution 3

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.

Sorting: Solution 3

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.

sorted

unsorted

Insertion Sort

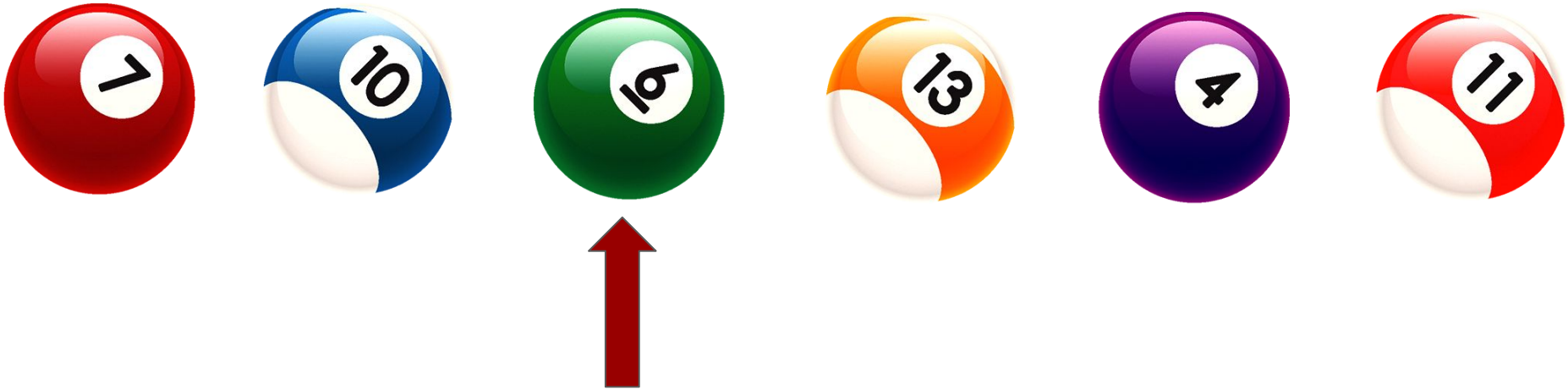
In this Algorithm, we are inserting the current element in the already sorted Array. Therefore, this algorithm is called as **Insertion Sort**.

sorted

unsorted

Insertion Sort: 2nd Pass

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.



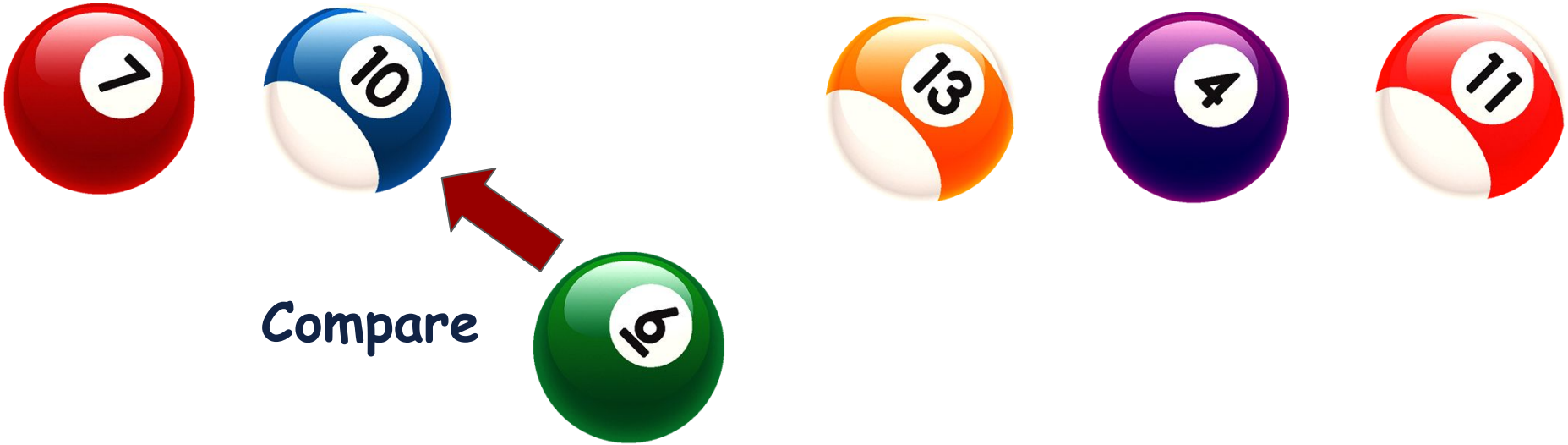
Insertion Sort: 2nd Pass

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.

Store current element separately

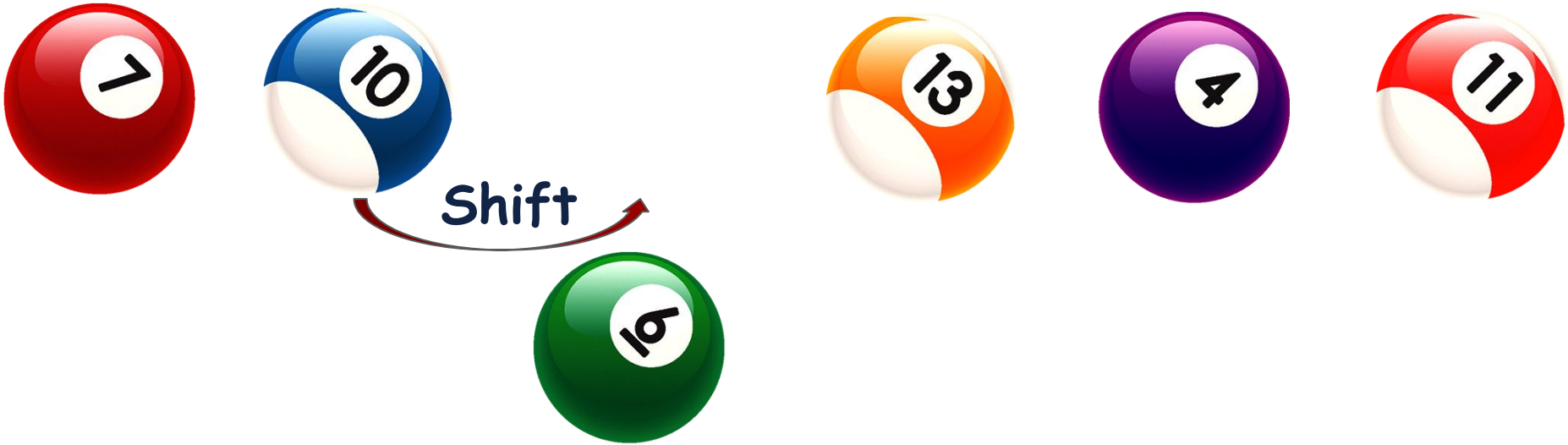
Insertion Sort: 2nd Pass

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.



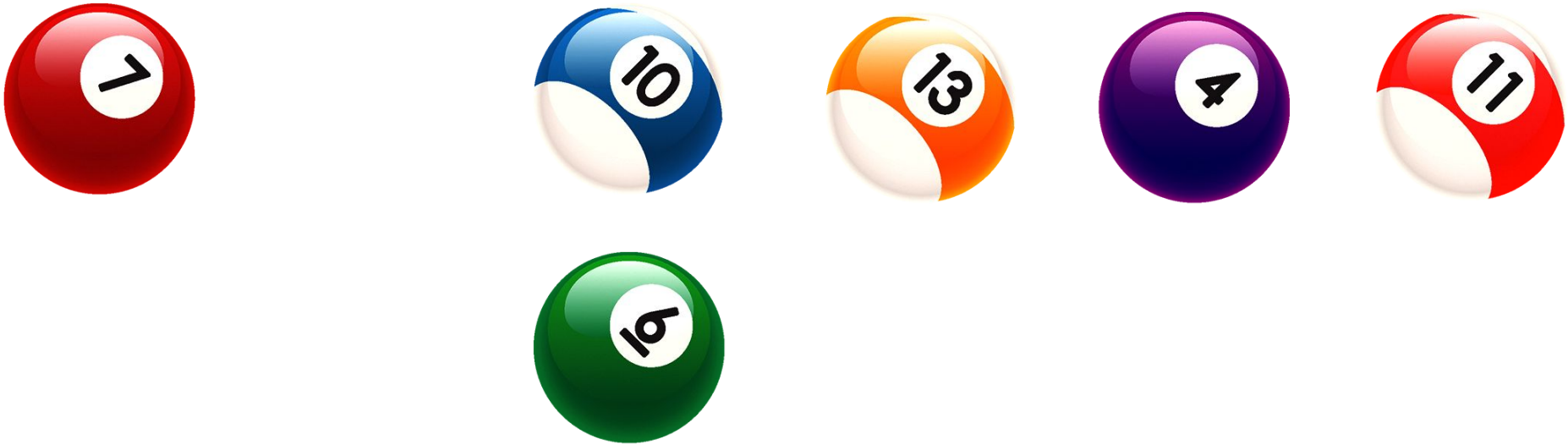
Insertion Sort: 2nd Pass

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.



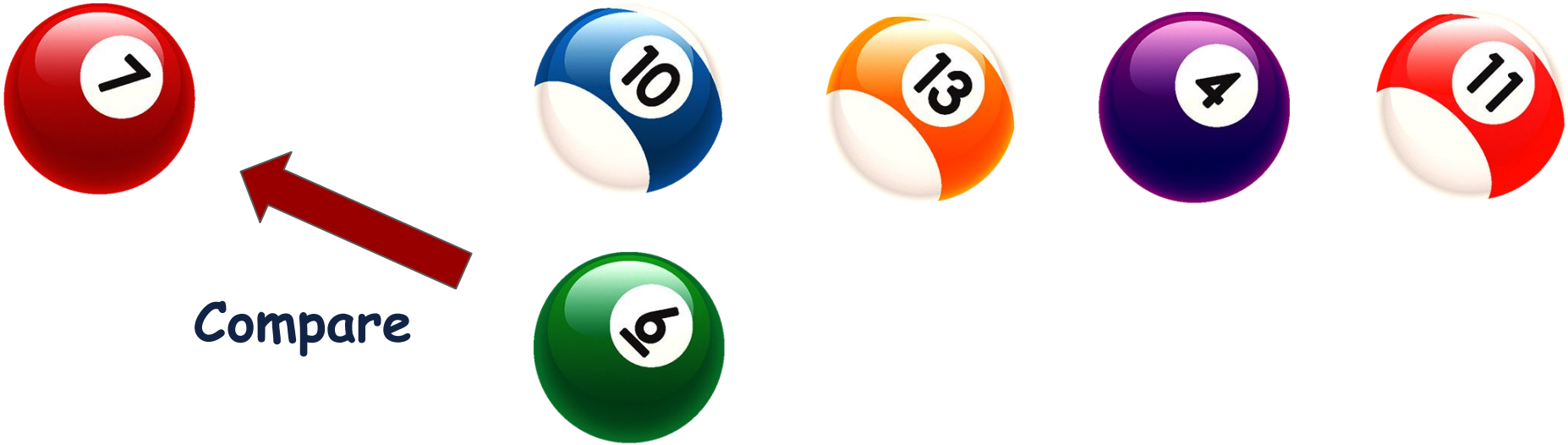
Insertion Sort: 2nd Pass

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.



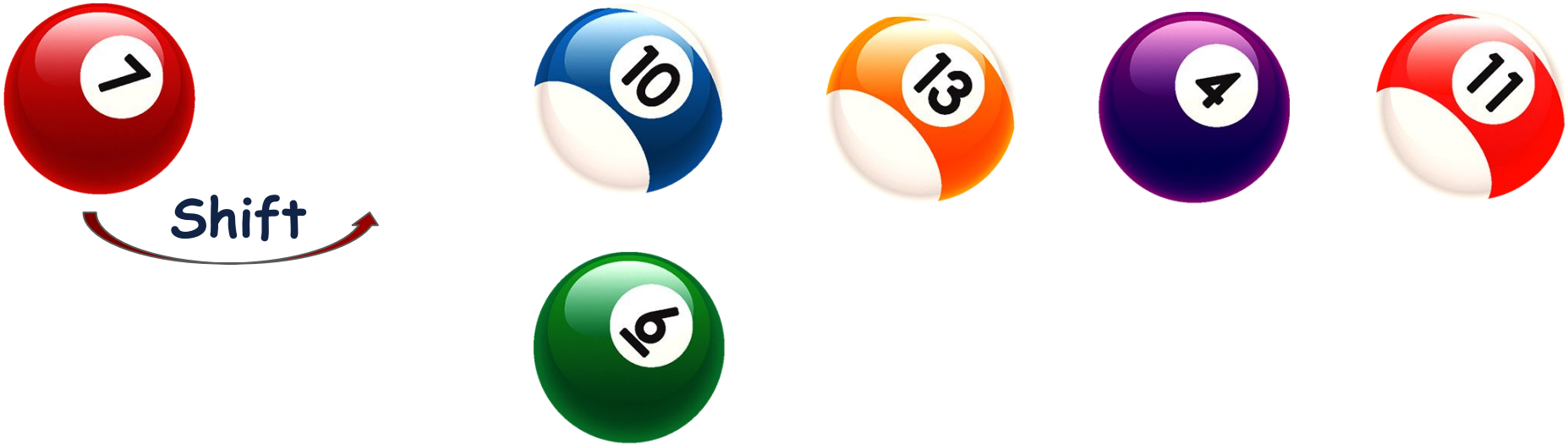
Insertion Sort: 2nd Pass

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.



Insertion Sort: 2nd Pass

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.



Insertion Sort: 2nd Pass

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.

Insertion Sort: 2nd Pass

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.

Place at the correct location

Insertion Sort: 2nd Pass

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.

Insertion Sort: 2nd Pass

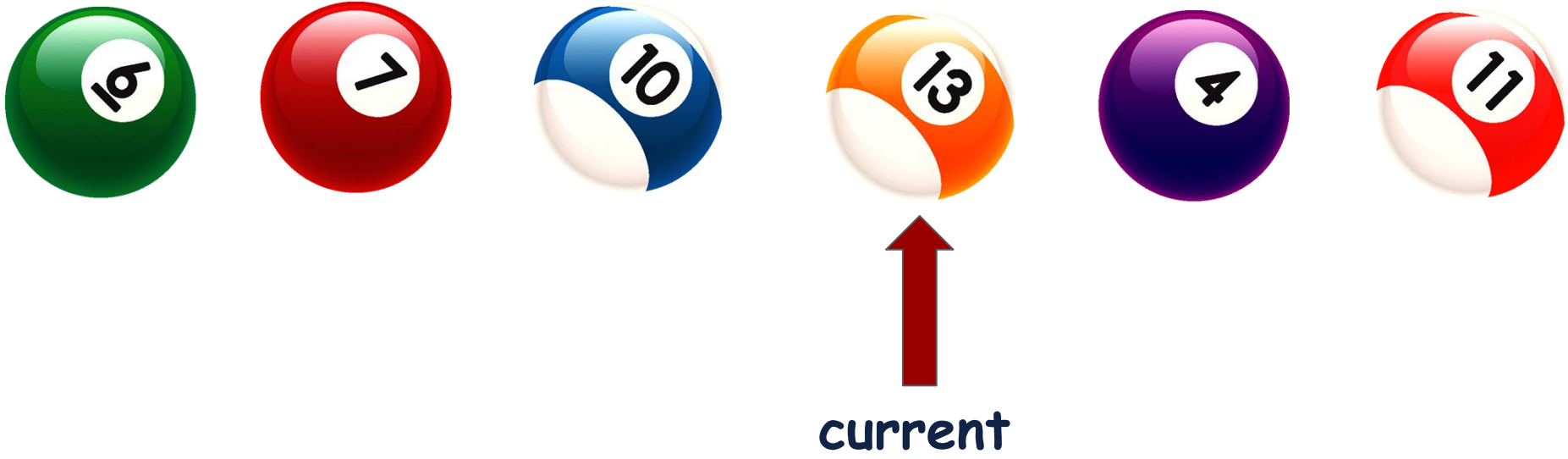
The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

sorted

unsorted

Insertion Sort: 3rd Pass

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.



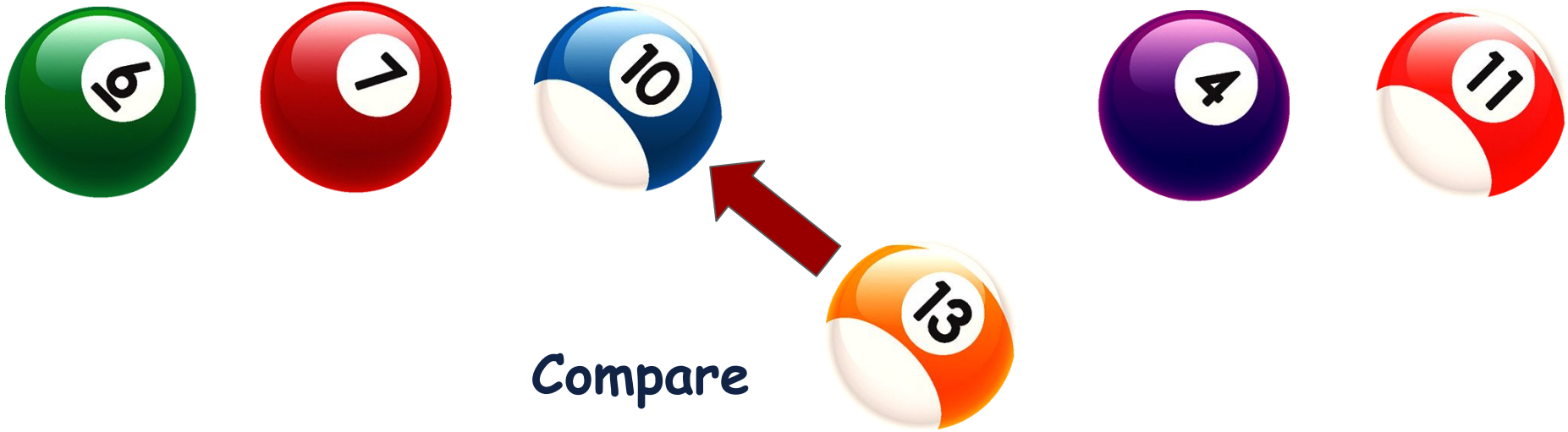
Insertion Sort: 3rd Pass

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.

Store current element separately

Insertion Sort: 3rd Pass

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.



Insertion Sort: 3rd Pass

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.

Place at the correct location

Insertion Sort: 3rd Pass

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.

Insertion Sort: 3rd Pass

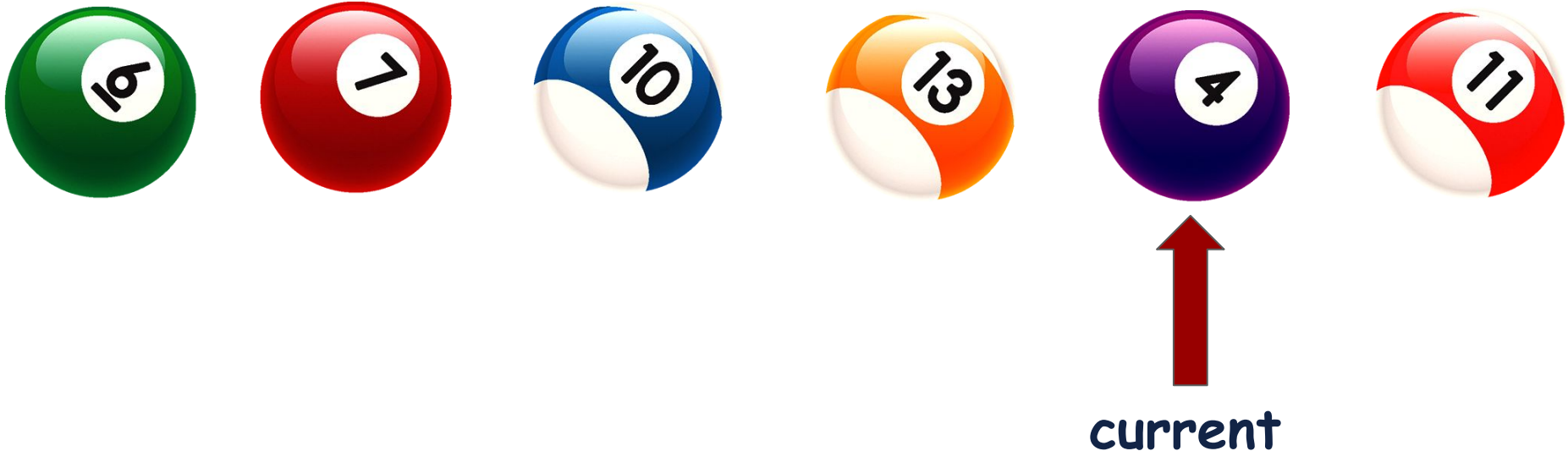
The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

sorted

unsorted

Insertion Sort: 4th Pass

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.



Insertion Sort: 4th Pass

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.

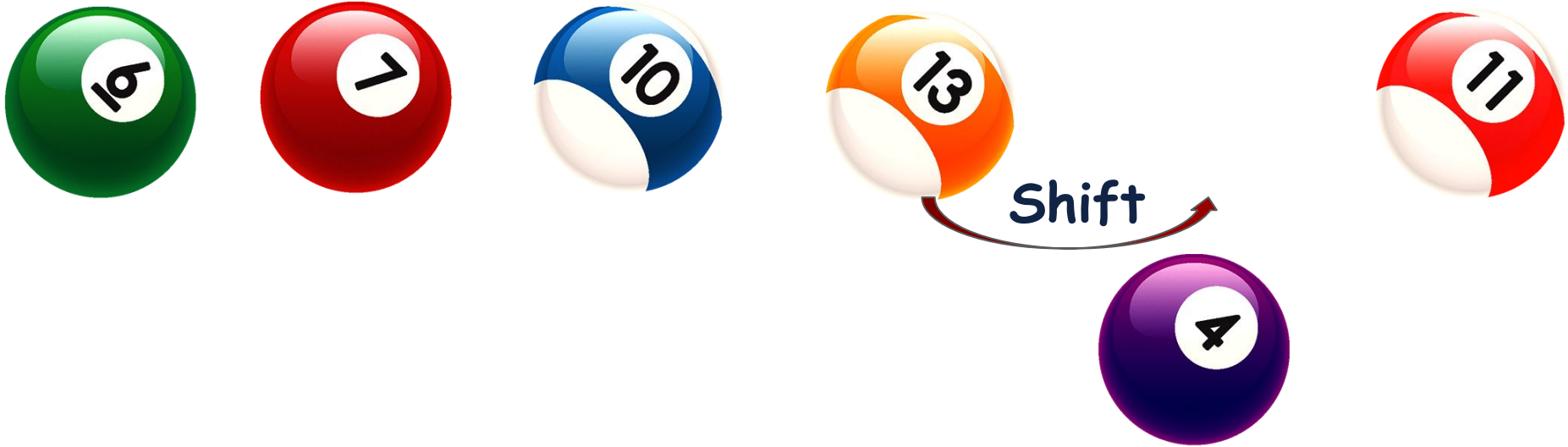
Store current element separately

Insertion Sort: 4th Pass

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.

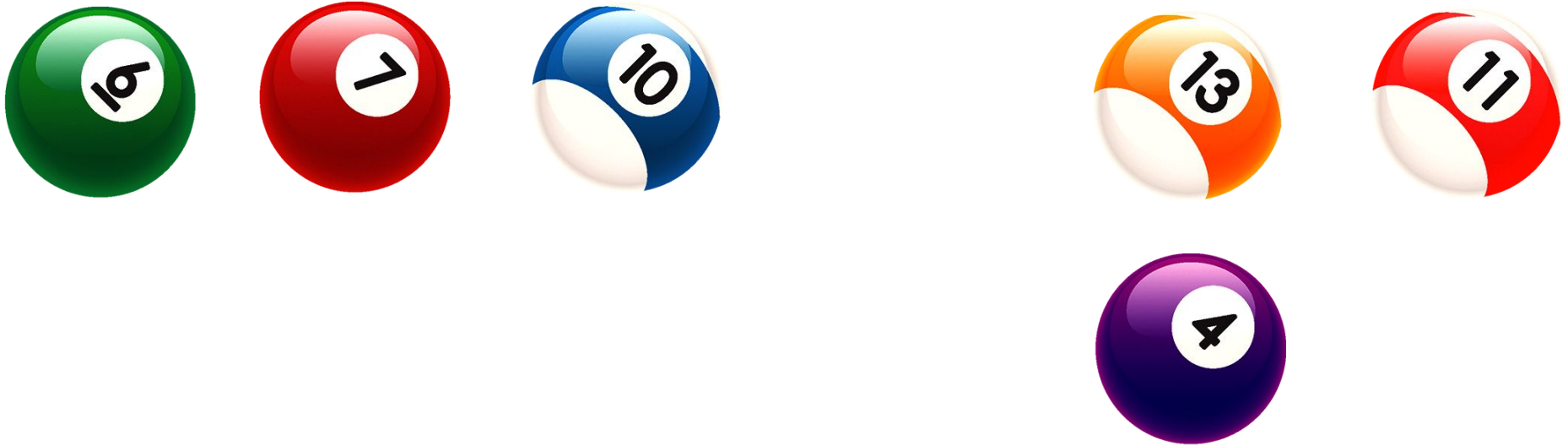
Insertion Sort: 4th Pass

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.



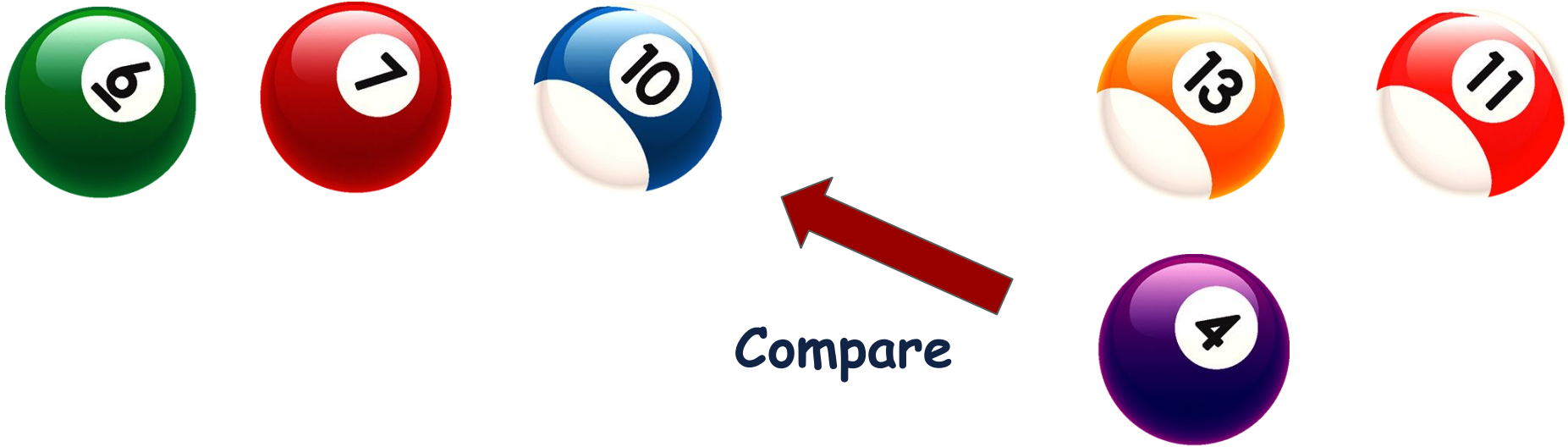
Insertion Sort: 4th Pass

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.



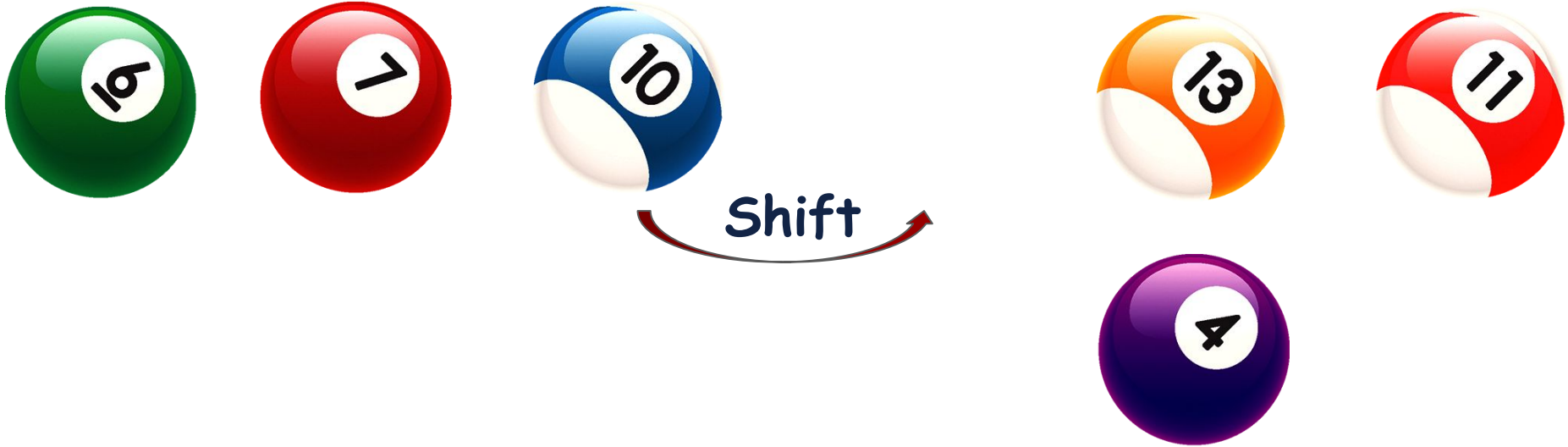
Insertion Sort: 4th Pass

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.



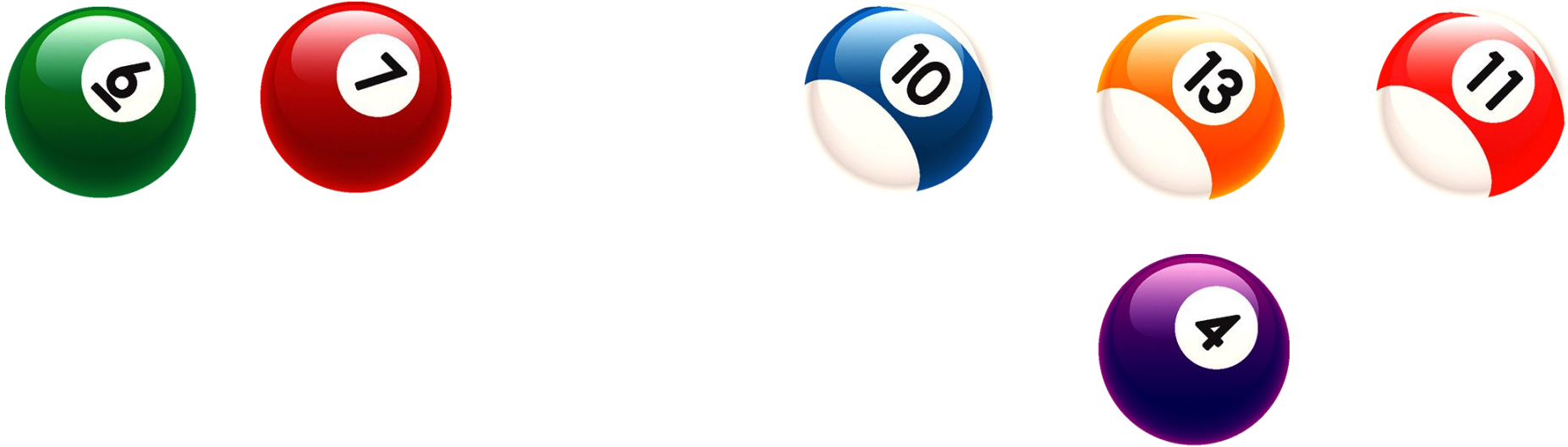
Insertion Sort: 4th Pass

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.



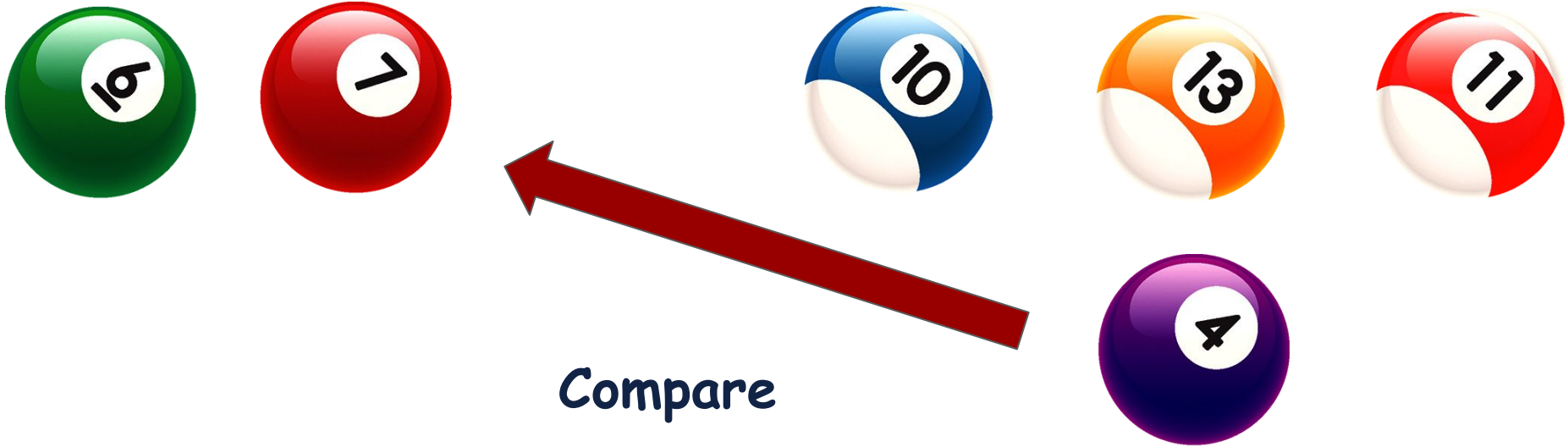
Insertion Sort: 4th Pass

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.



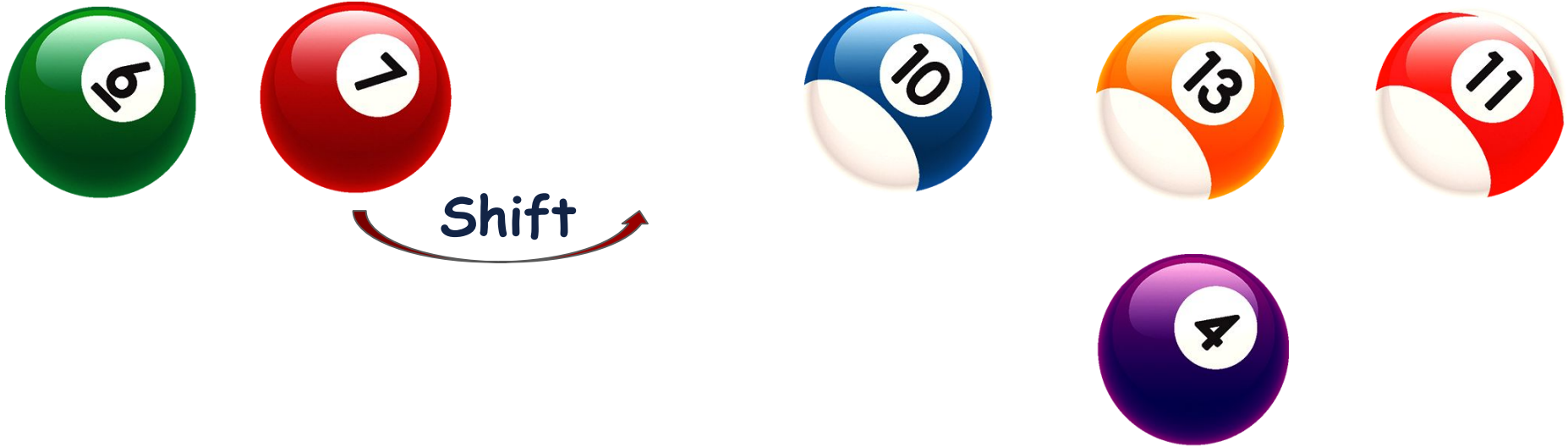
Insertion Sort: 4th Pass

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.



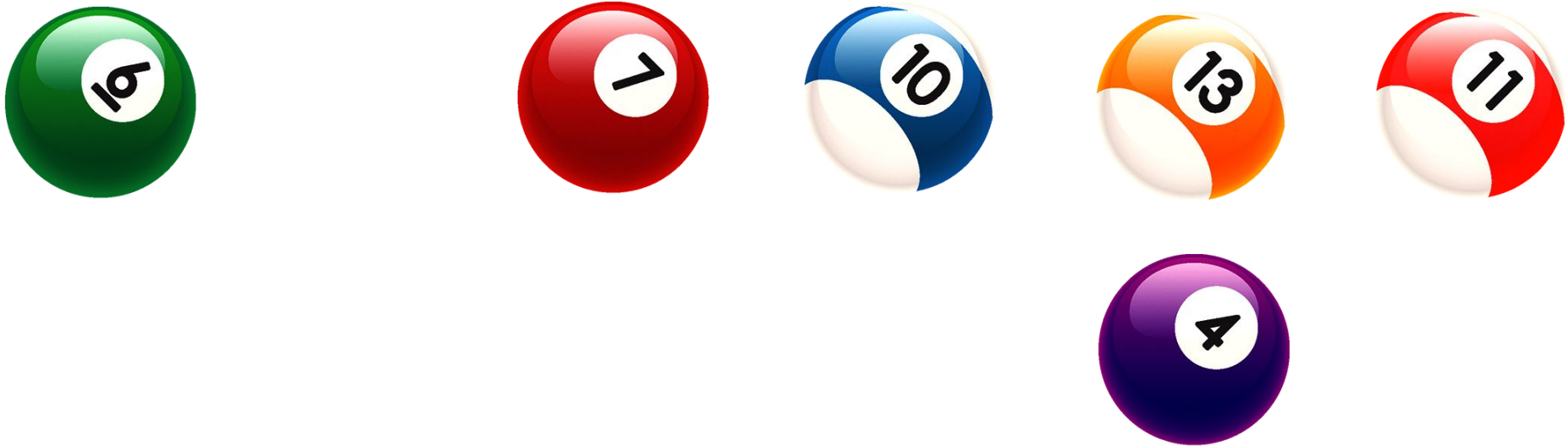
Insertion Sort: 4th Pass

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.



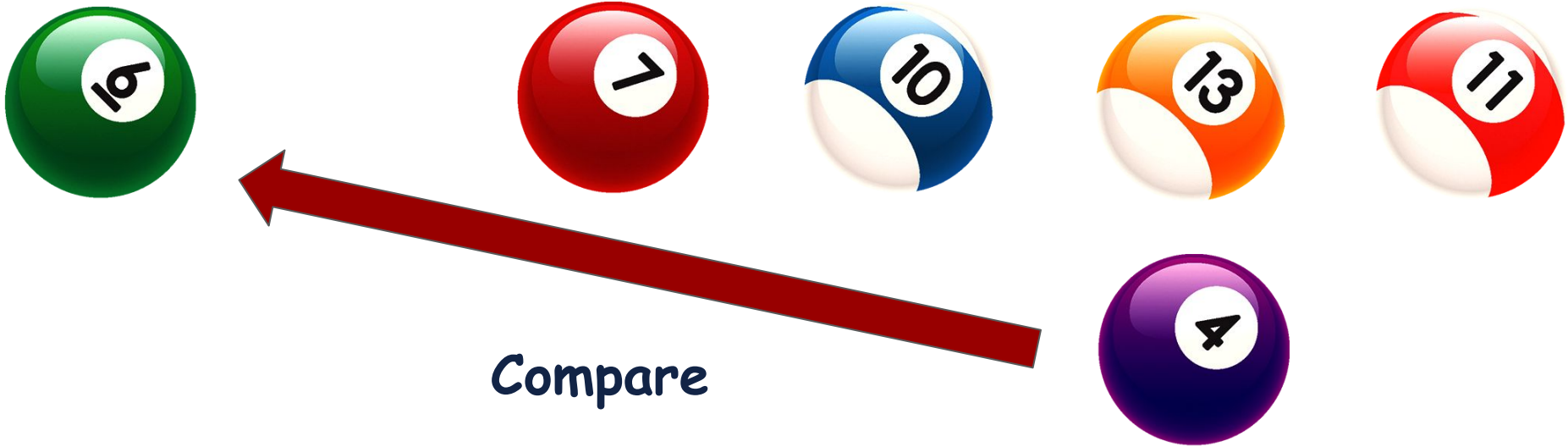
Insertion Sort: 4th Pass

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.



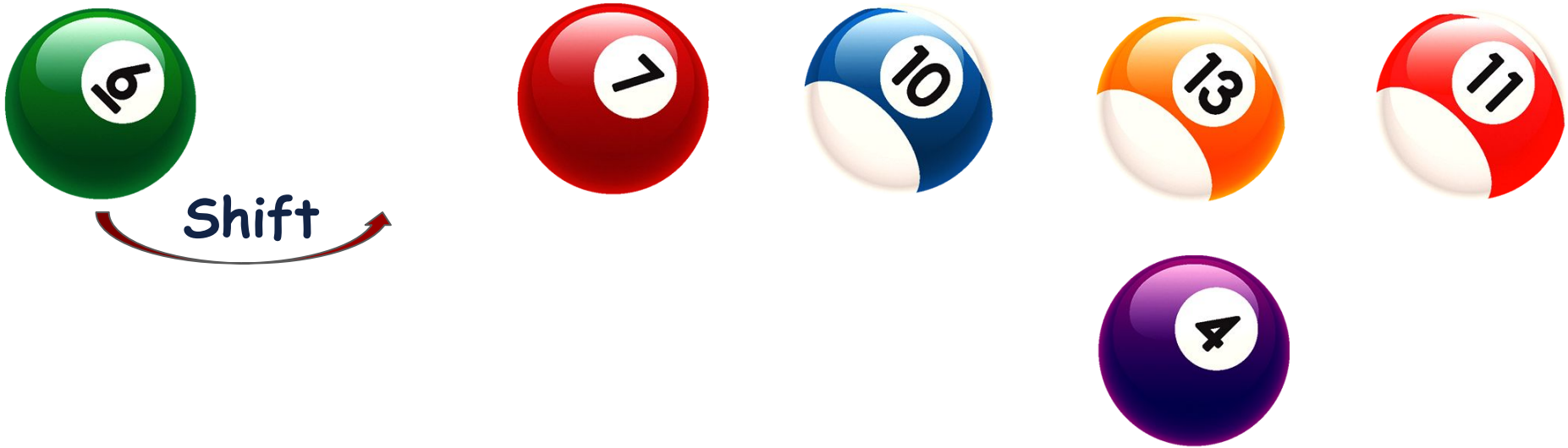
Insertion Sort: 4th Pass

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.



Insertion Sort: 4th Pass

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.



Insertion Sort: 4th Pass

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.

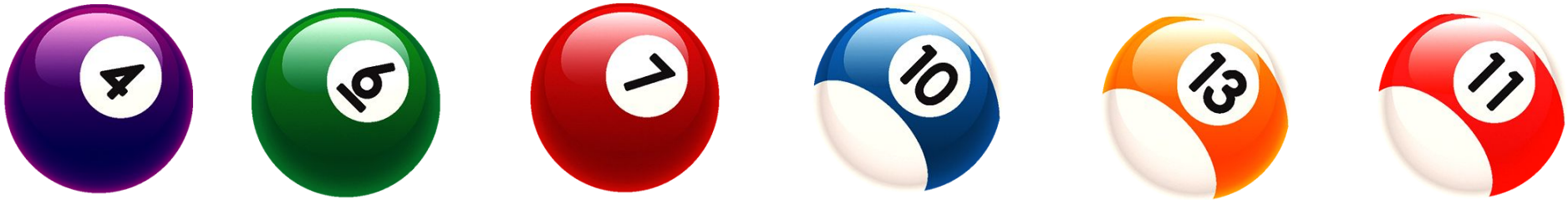
Insertion Sort: 4th Pass

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.

Place at the correct location

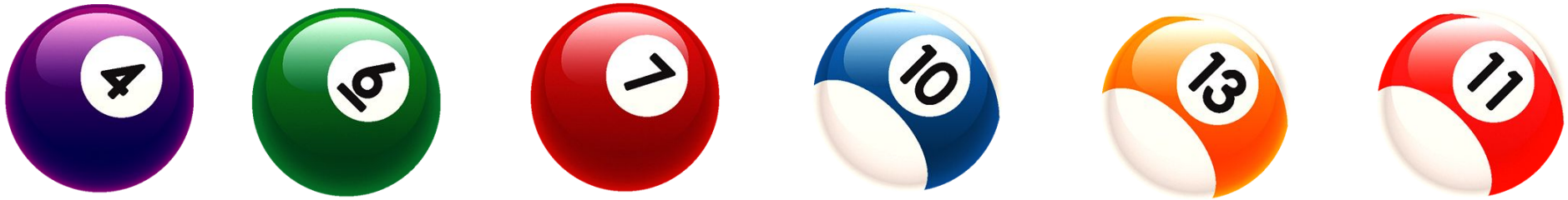
Insertion Sort: 4th Pass

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.



Insertion Sort: 4th Pass

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.

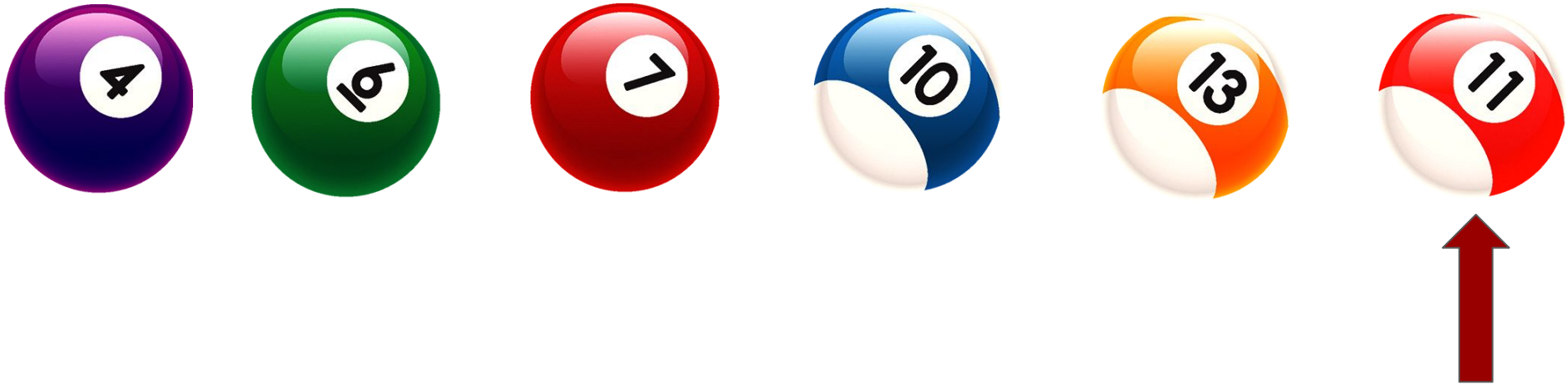


sorted

unsorted

Insertion Sort: 5th Pass

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.



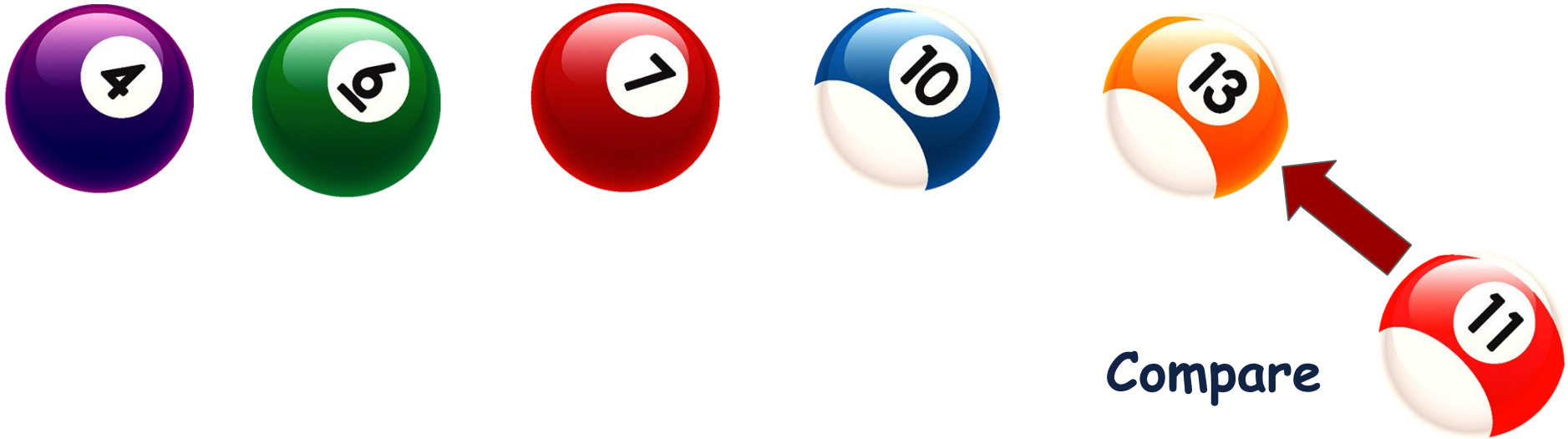
Insertion Sort: 5th Pass

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.

Store current element separately

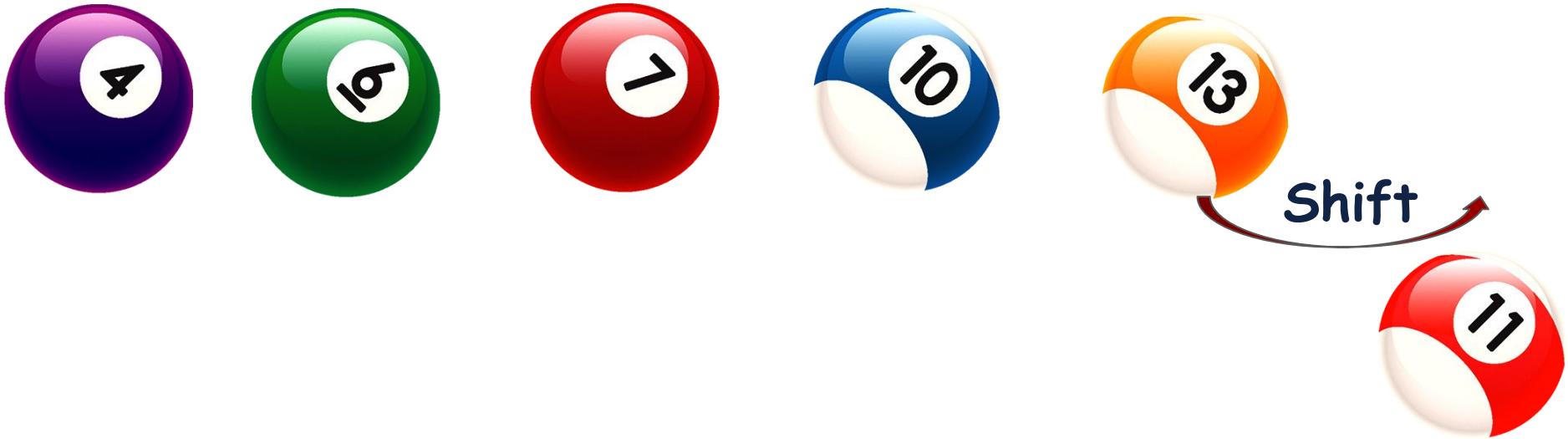
Insertion Sort: 5th Pass

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.



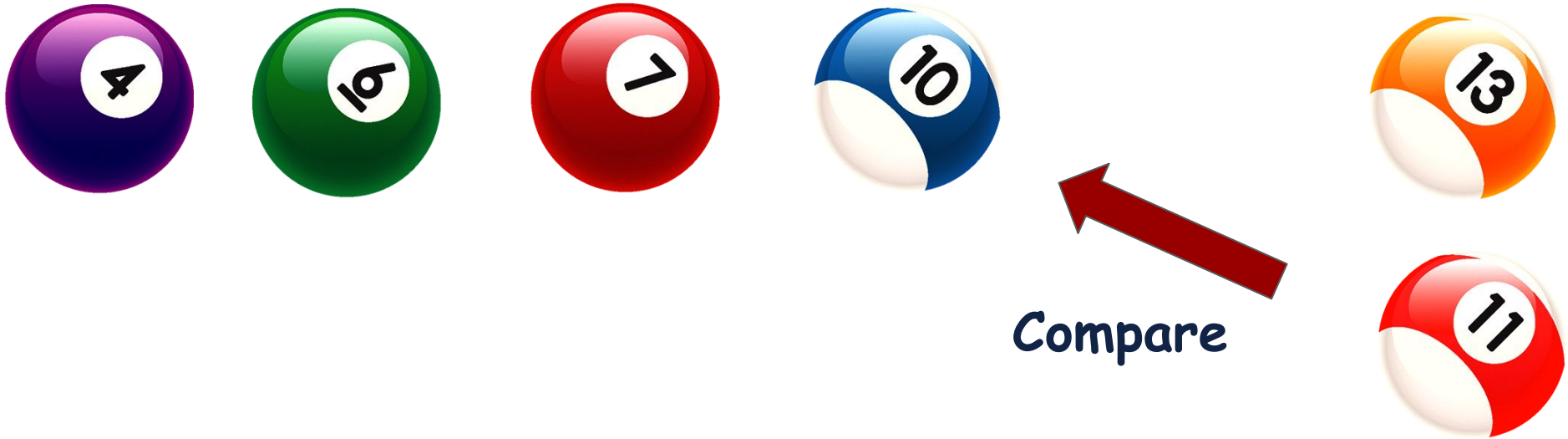
Insertion Sort: 5th Pass

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.



Insertion Sort: 5th Pass

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.



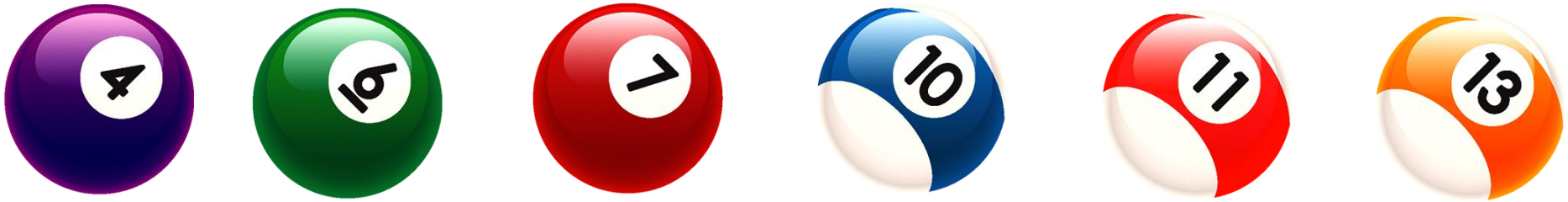
Insertion Sort: 5th Pass

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.

Place at the correct location

Insertion Sort: 5th Pass

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.



sorted

Insertion Sort: Implementation

Let's implement the algorithm now.

Insertion Sort: Implementation

```
main()
{
    int arr[6] = {10, 7, 6, 13, 4, 11};
    insertionSort(arr, 6);
    for (int x = 0; x < 6; x++)
    {
        cout << arr[x] << " ";
    }
}
```

```
void insertionSort(int arr[], int n)
{
    for (int x = 1; x < n; x++)
    {
        int key = arr[x];
        int y = x - 1;
        while (y >= 0 && arr[y] > key)
        {
            arr[y + 1] = arr[y];
            y--;
        }
        arr[y + 1] = key;
    }
}
```

Insertion Sort: Implementation

What is the Time Complexity?

```
main()
{
    int arr[6] = {10, 7, 6, 13, 4, 11};
    insertionSort(arr, 6);
    for (int x = 0; x < 6; x++)
    {
        cout << arr[x] << " ";
    }
}
```

```
void insertionSort(int arr[], int n)
{
    for (int x = 1; x < n; x++)
    {
        int key = arr[x];
        int y = x - 1;
        while (y >= 0 && arr[y] > key)
        {
            arr[y + 1] = arr[y];
            y--;
        }
        arr[y + 1] = key;
    }
}
```

Insertion Sort: Implementation

What is the Time Complexity?

```
main()
{
    int arr[6] = {10, 7, 6, 13, 4, 11};
    insertionSort(arr, 6);
    for (int x = 0; x < 6; x++)
    {
        cout << arr[x] << " ";
    }
}
```

$O(n^2)$

```
void insertionSort(int arr[], int n)
{
    for (int x = 1; x < n; x++)
    {
        int key = arr[x];
        int y = x - 1;
        while (y >= 0 && arr[y] > key)
        {
            arr[y + 1] = arr[y];
            y--;
        }
        arr[y + 1] = key;
    }
}
```

Bubble VS Selection VS Insertion

Sorting Algorithm	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	O(N)	$O(N^2)$	$O(N^2)$	O(1)
Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$	O(1)
Insertion Sort	O(N)	$O(N^2)$	$O(N^2)$	O(1)

Bubble VS Selection VS Insertion

Sorting Algorithm	In-Place	Stable
Bubble Sort	Yes	Yes
Selection Sort	Yes	No
Insertion Sort	Yes	Yes

Learning Objective

Students should be able to **apply** sorting using Bubble, Selection and Insertion Sort.

Self Assessment

To visually see the Algorithms Running

https://visualgo.net/en/sorting

Self Assessment

We have studied the Bubble, Selection and Insertion Sort algorithm. The worst time complexity of all of these algorithms are same. So, Why and in which cases they are useful. Why the scientists thought to invent these algorithms?