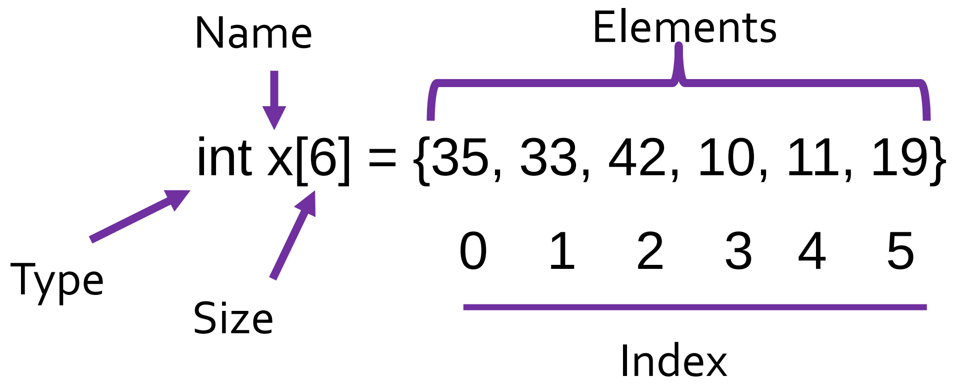
# DATA STRUCTURES & ALGORITHMS

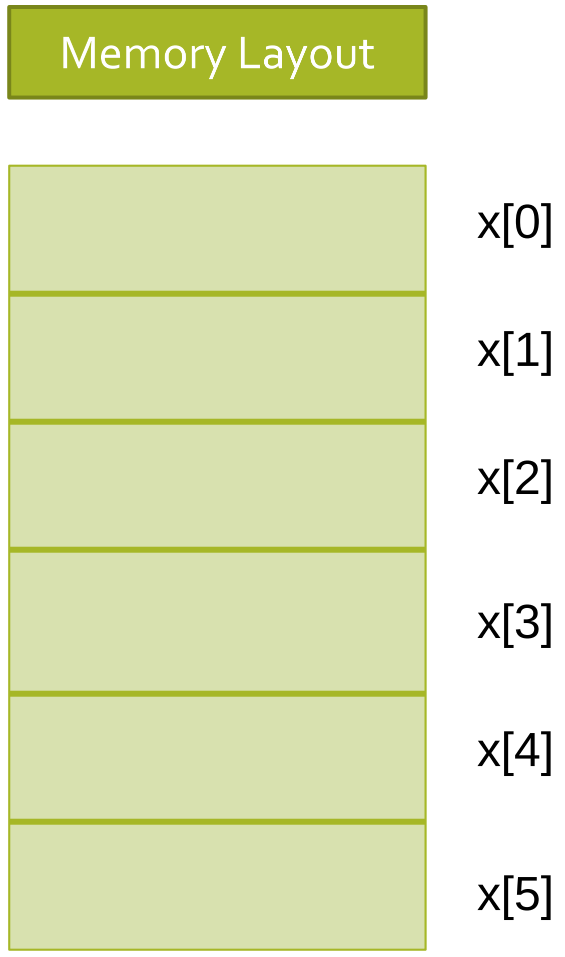
Arrays

Instructor: Engr. Laraib Siddiqui

## Arrays

- Collection of cells of the same type.
- Cells are numbered with consecutive integers
- Declaration:  int x[6];
- Access: use the array name and an index:
  x[0], x[1], x[2], x[3], x[4], x[5]





# Declaring array

Syntax: *data_type array_name [array_size];*

- If we are declaring 'x' is an array name, 'x' is not an value.
- For example,

```
            int a, b;
```

  we can write

```
            b = 2;
            a = b;
```

  But we cannot write

```
            2 = a; // not allowed
```

```
int x[6];
int n;

x[0] = 5;
x[1] = 2;

x = 3;        // not allowed
x = a + b;    // not allowed
x = &n;       // not allowed
```

What's special about arrays?

Constant time access

## Traversing an array

LA is a linear array with lower bound LB and upper bound UB.

```
1. [Initialize counter] Set K := LB.
2. Repeat steps 3 and 4 while K <= UB.
3.    [Visit element] Apply PROCESS to LA[K].
4.    [Increment counter] Set K := K + 1.
      [End of Step 2 loop.]
5. Exit.
```

## Traversing an array

$\longrightarrow$

e.g Array {8,9,5,12}

```
for (i=0; i<4; i++)
{
        printf ("%d\t", a[i]);
}
```

```
8    9    5    12
```

# Insertion in arrays

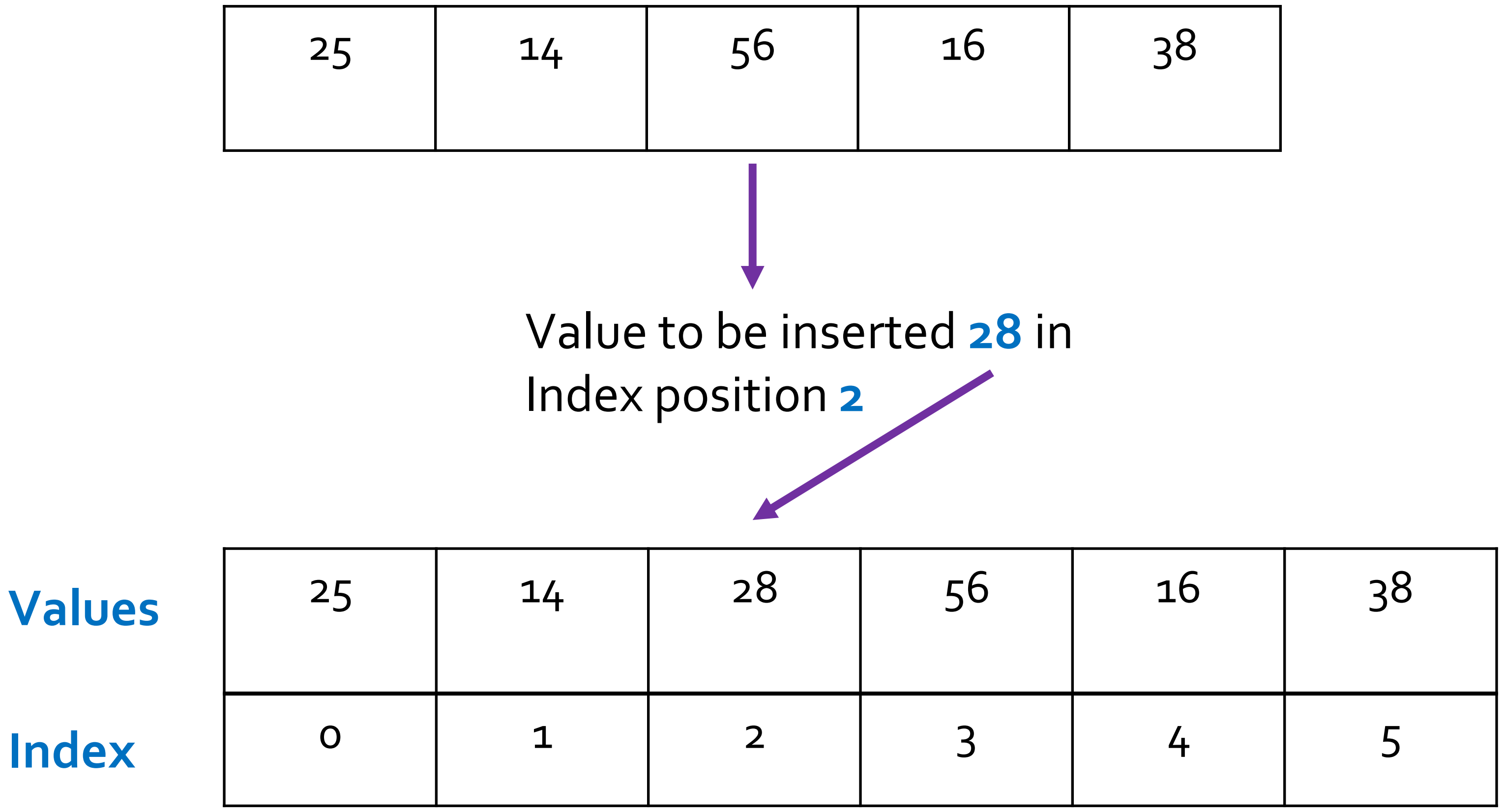
INSERT (LA, N, K, ITEM)

linear array with N elements and K is a positive integer such that K <= N.

Following algorithm inserts an element ITEM into the Kth position in LA.

```
1.[Initialize counter.] Set J := N.
2.Repeat Steps 3 and 4 while J >= K.
3.        [Move Jth element downward.] Set LA[J+1] := LA[J].
4.        [Decrease counter.] Set J := J – 1.
   [End of Step 2 loop.]
5.[Insert element.] Set LA[K] := ITEM.
6.[Reset N.] Set N := N + 1.
7.Exit.
```

## Insertion in arrays



|    |    |    |    |    |
|----|----|----|----|----|
| 25 | 14 | 56 | 16 | 38 |

Value to be inserted **28** in Index position **2**

|            |    |    |    |    |    |    |
|------------|----|----|----|----|----|----|
| **Values** | 25 | 14 | 28 | 56 | 16 | 38 |
| **Index**  | 0  | 1  | 2  | 3  | 4  | 5  |

## Deletion

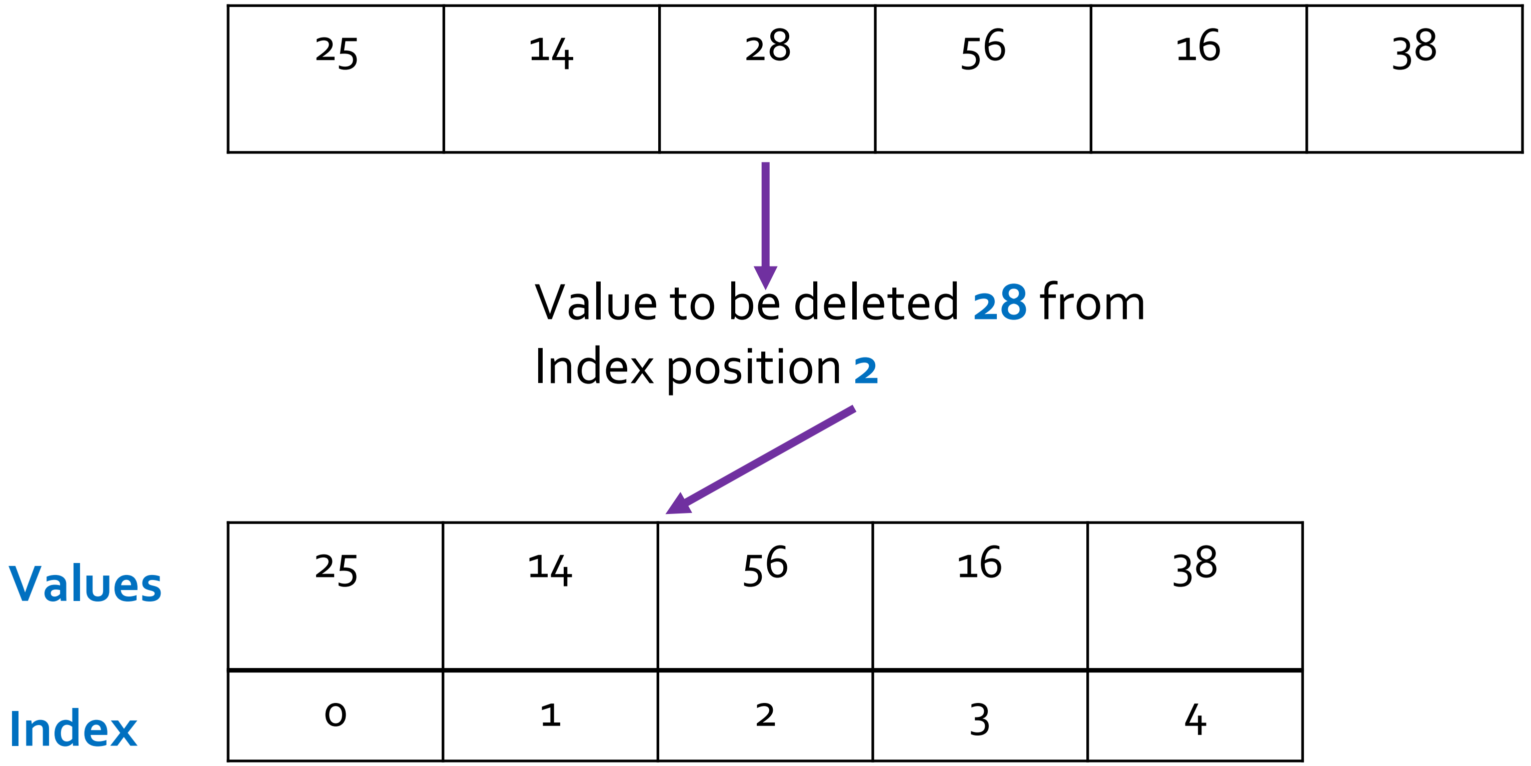
DELETE (LA, N, K, ITEM)

array with N elements and K is a positive integer such that K <= N.

Following algorithm deletes the Kth element from LA.

```
1. Set ITEM := LA[K].
2. Repeat for J = K to N - 1.
            [Move J + 1st element upward.] Set LA[J] := LA[J + 1].
   [End of loop.]
3. [Reset the number N of elements in LA.] Set N := N - 1.
4. Exit.
```

# Deletion



|    |    |    |    |    |    |
|----|----|----|----|----|----|
| 25 | 14 | 28 | 56 | 16 | 38 |

Value to be deleted **28** from
Index position **2**

| | | | | | |
|---|---|---|---|---|---|
| **Values** | 25 | 14 | 56 | 16 | 38 |
| **Index** | 0 | 1 | 2 | 3 | 4 |

## Dynamic Arrays

Problem:

- Fixed size
- Process of different data sets with same array is not possible

Solution:

- Dynamic Arrays

## Applications

- Accessing and storing sequential data.
- Creating string objects temporary.
- I/O routines as buffers
- Dynamic programming
- Lookup tables

## Complexity

| | Static | Dynamic |
|---|---|---|
| **Access** | O(1) | O(1) |
| **Search** | O(n) | O(n) |
| **Delete** | O(n) | O(n) |
| **Insertion** | O(1) | O(n) |

## Sorting

**Sorting can be defined as an operation of rearranging the elements such that:**

**A[1] < A[2] < …….. < A[N]**

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 77 | 42 | 35 | 12 | 101 | 5 |

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 5 | 12 | 35 | 42 | 77 | 101 |

## Bubble Sort

- Bubble Sort is a simple yet effective sorting algorithm.
- Data is placed adjacent to each other.
- The sort starts from one end (the beginning), compares 2 adjacent data, and swaps them if they are in the wrong order.
- It moves on down the list and continues doing so.
- When it reaches the end of the data, it starts over until all the data is in the right order.
- Its not efficient when dealing with a large set of data.
- Compared to other sorting algorithm, bubble sort is really slow.

## Algorithm

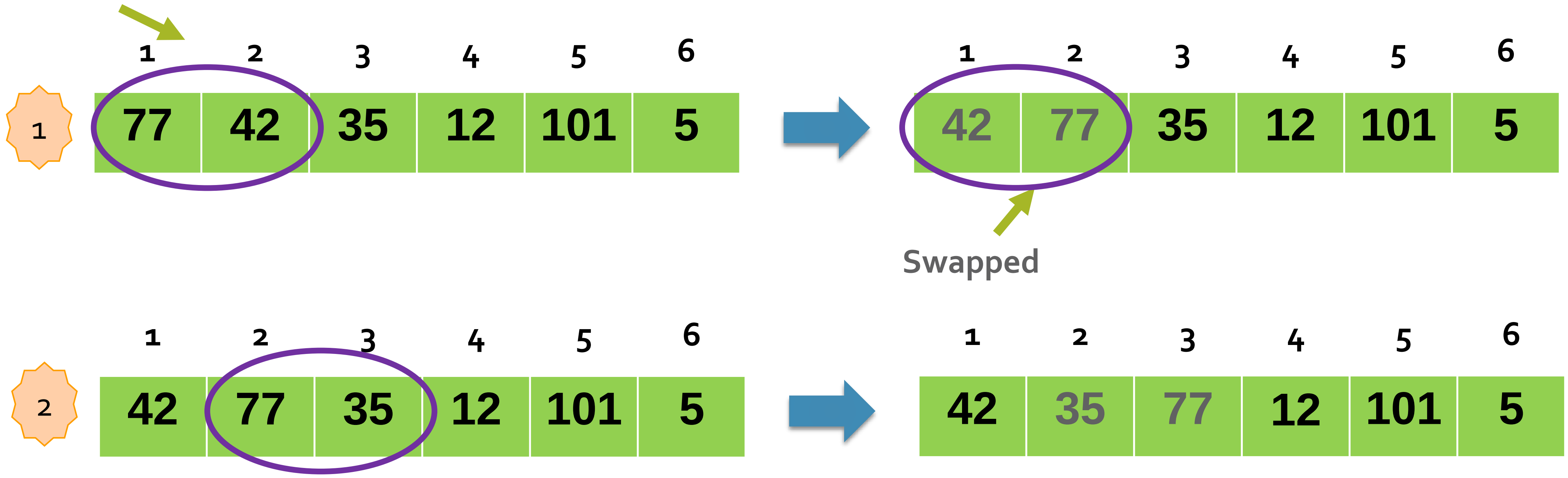
**Step 1** : Starting with the first element(index = 0), compare the current element with the next element of the array.

**Step 2**: If the current element is greater than the next element of the array, swap them.

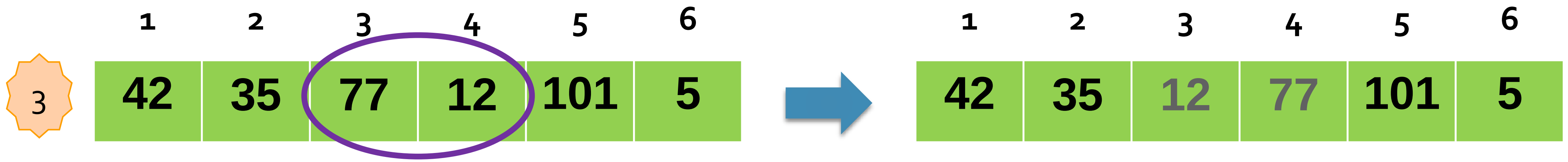
**Step 3**: If the current element is less than the next element, move to the next element. Repeat Step 1.

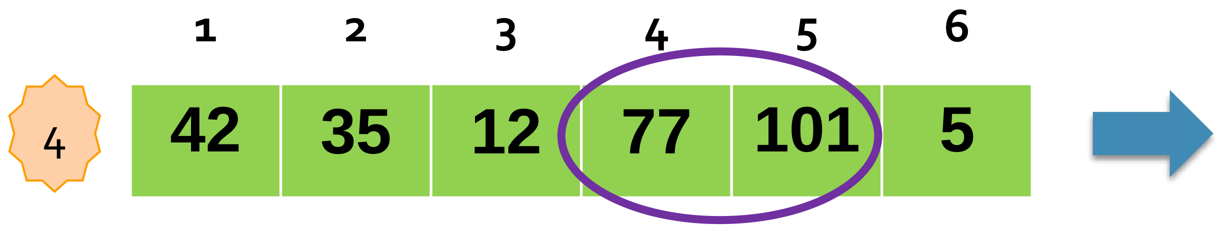
# Bubble Sort - Example

**First Iteration/pass**



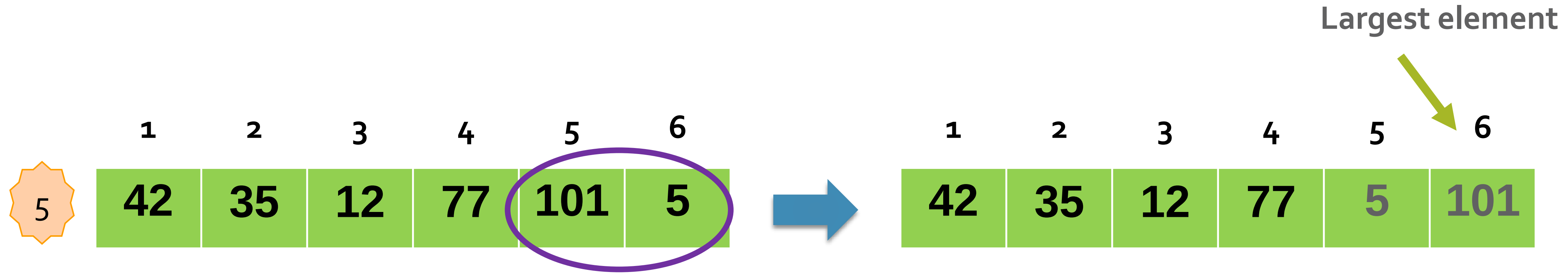
# Bubble Sort - Example



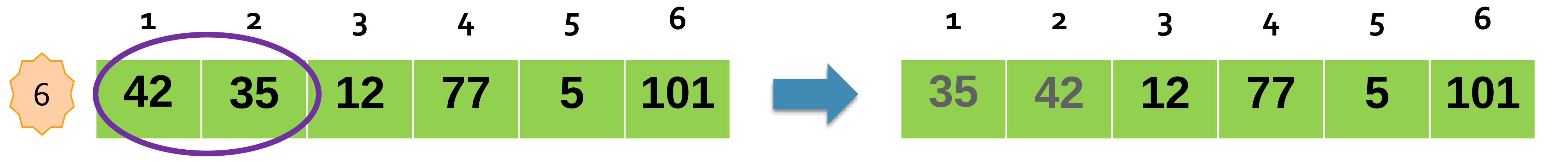


No need to swap

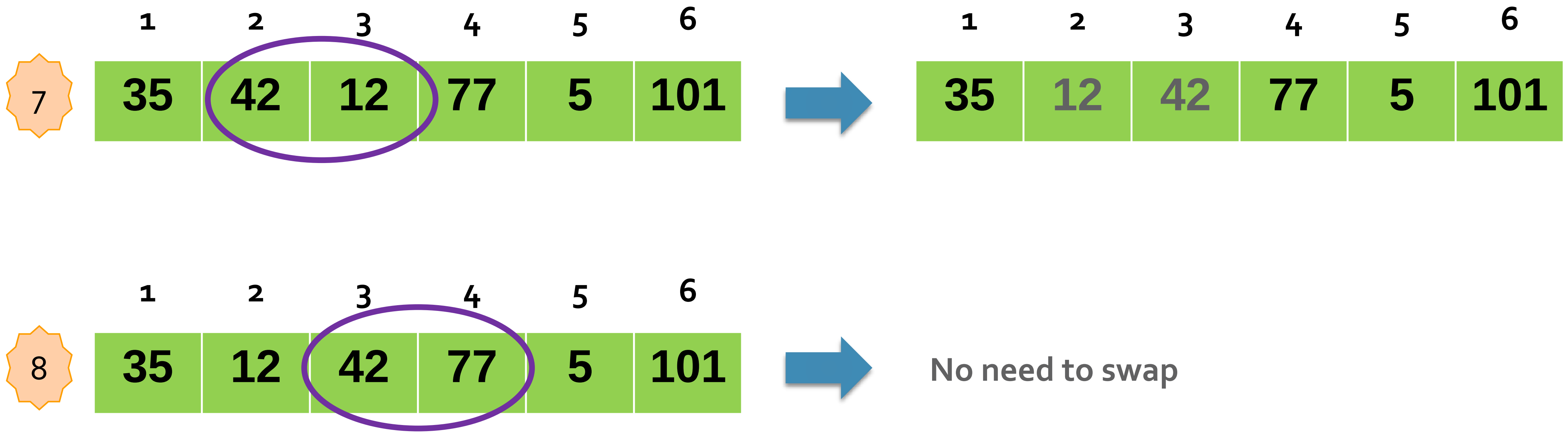
# Bubble Sort - Example



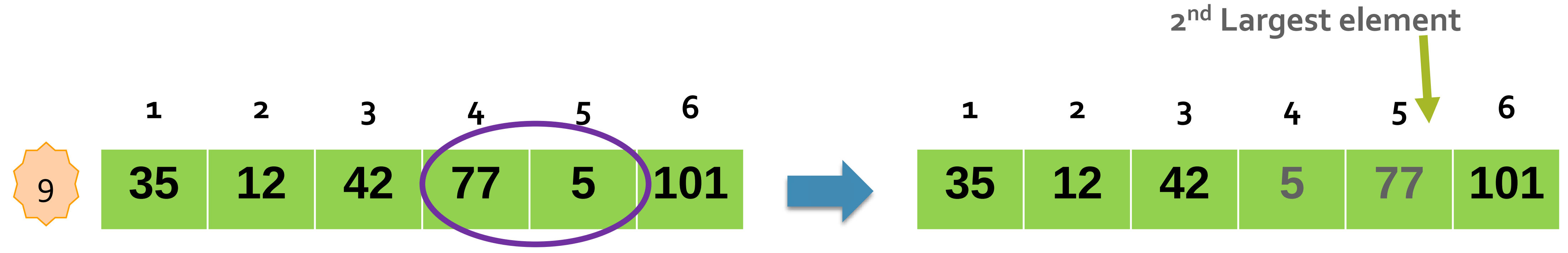
Second Iteration/pass



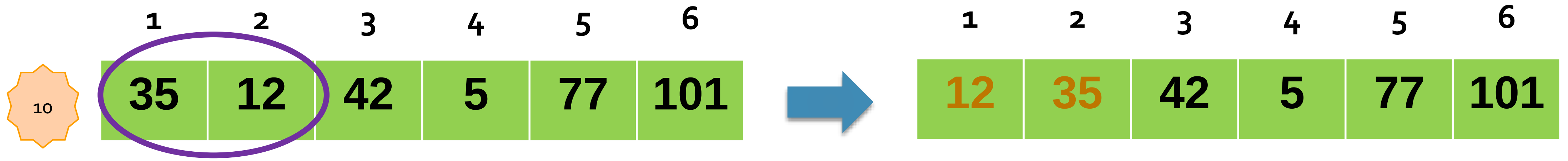
# Bubble Sort - Example



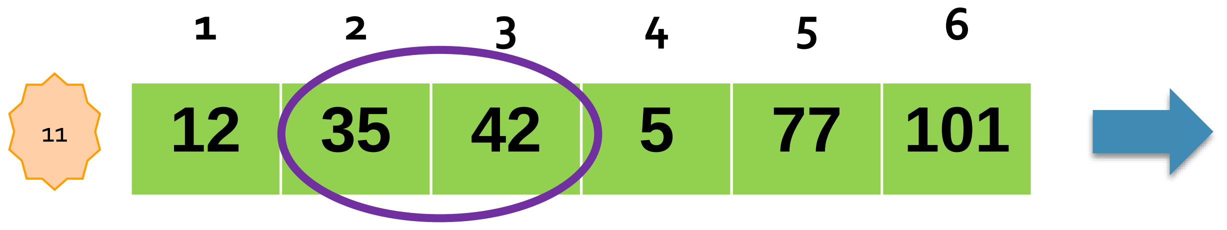
# Bubble Sort - Example



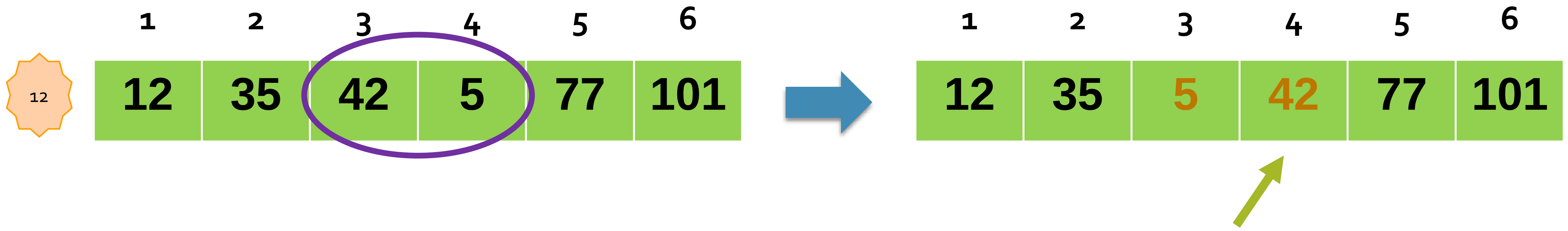
Third Iteration/pass



# Bubble Sort - Example



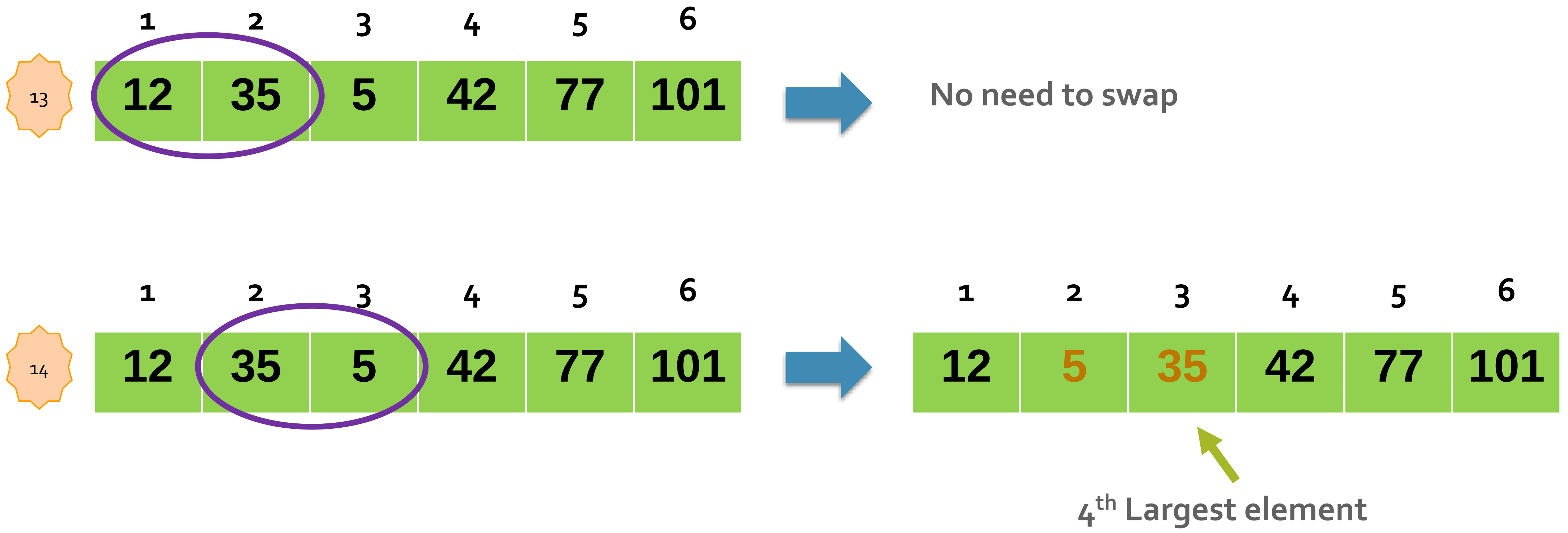
No need to swap



3rd Largest element

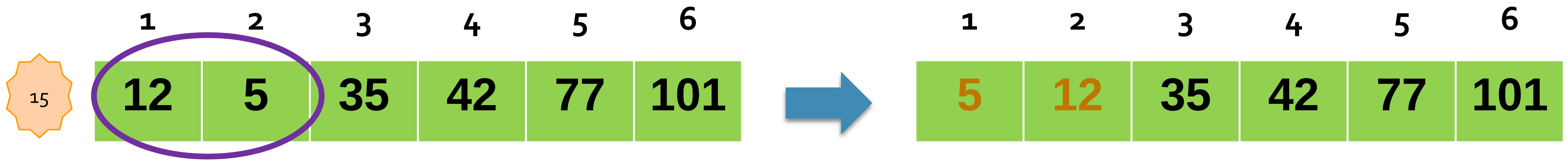
# Bubble Sort - Example

**Forth Iteration/pass**



# Bubble Sort - Example

**Fifth Iteration/pass**



Sorted list

## Selection Sort

- The Selection Sort searches (linear search) all of the elements in a list until it finds the smallest element. It “swaps” this with the first element in the list.
- Next it finds the smallest of the remaining elements, and “swaps” it with the second element and so on.
- Its not very efficient when dealing with a huge list of items.

## Algorithm

**Step 1** – Set MIN to location 0

**Step 2** – Search the minimum element in the list

**Step 3** – Swap with value at location MIN

**Step 4** – Increment MIN to point to next element

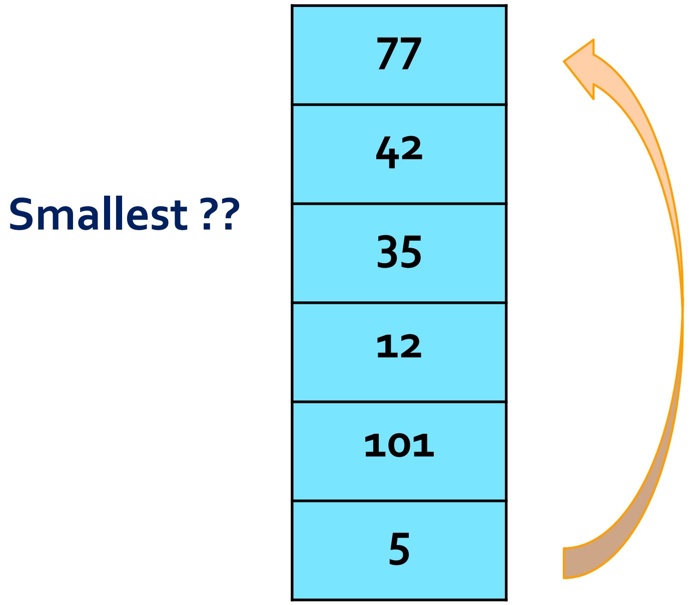
**Step 5** – Repeat until list is sorted

## Selection Sort - Example

**1st iteration/Pass:**

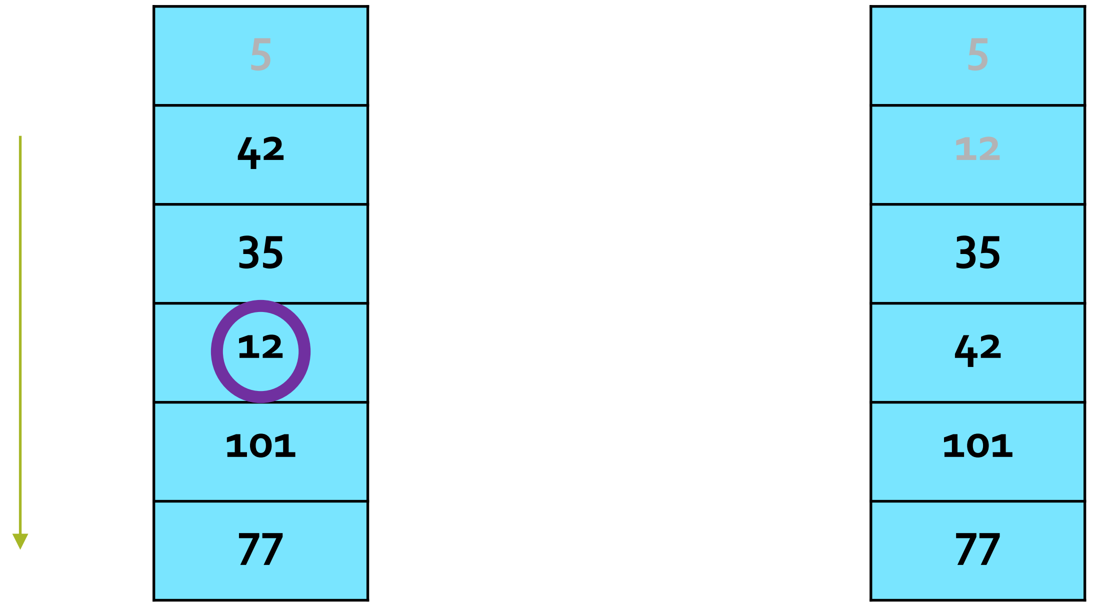
- ✓ Smallest = 77
- ✓ 42 < 77, smallest = 42
- ✓ 35 < 42, smallest = 35
- ✓ 12 < 35, smallest = 12
- ✓ 101 > 12, smallest = 12
- ✓ 5 < 12 smallest = 5

Swap 77 and 5



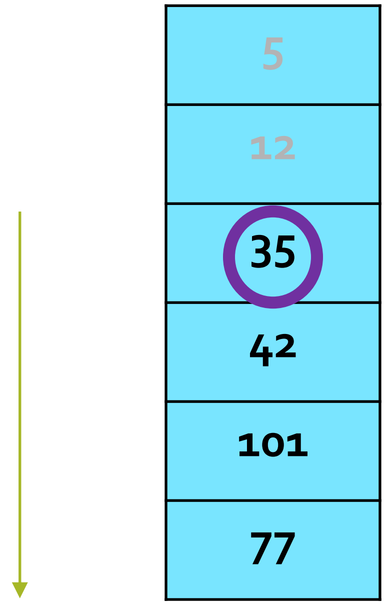
# Selection Sort - Example

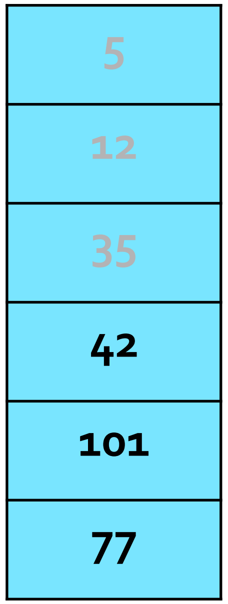
**Next Smallest ??**



# Selection Sort - Example

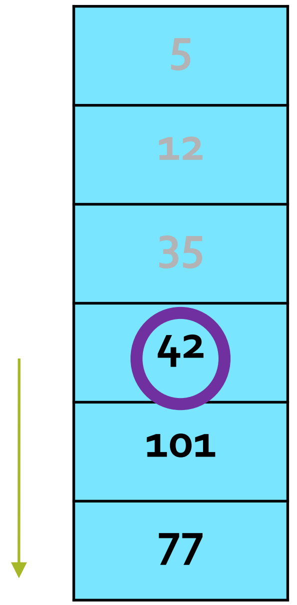
**Next Smallest ??**

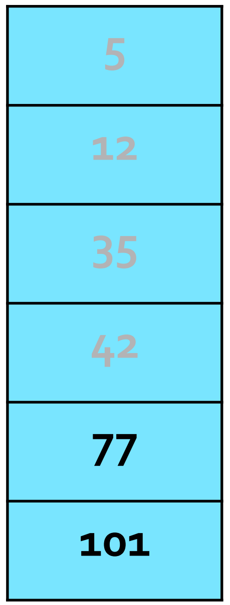




# Selection Sort - Example

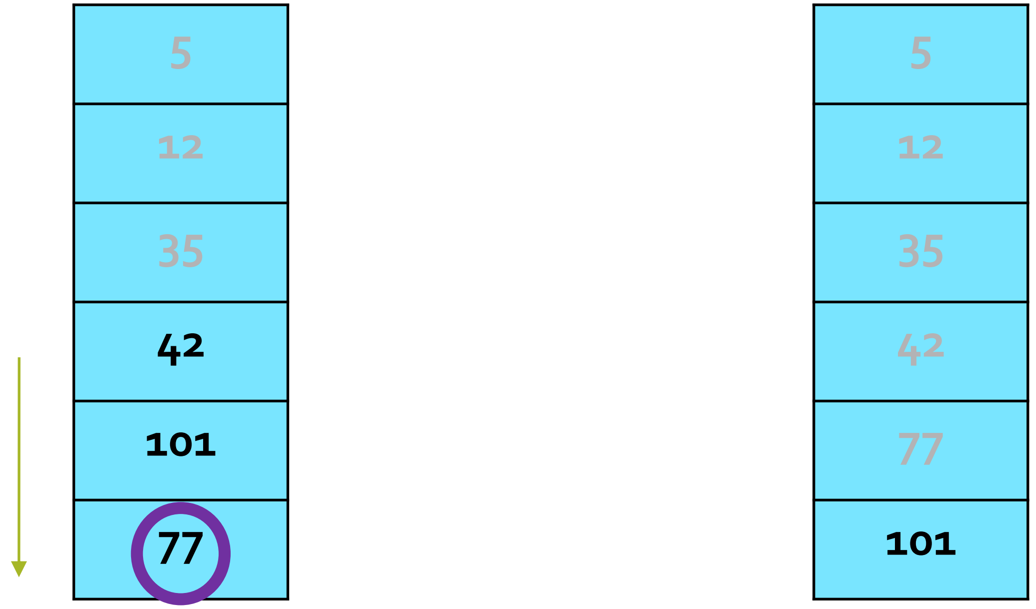
**Next Smallest ??**





# Selection Sort - Example

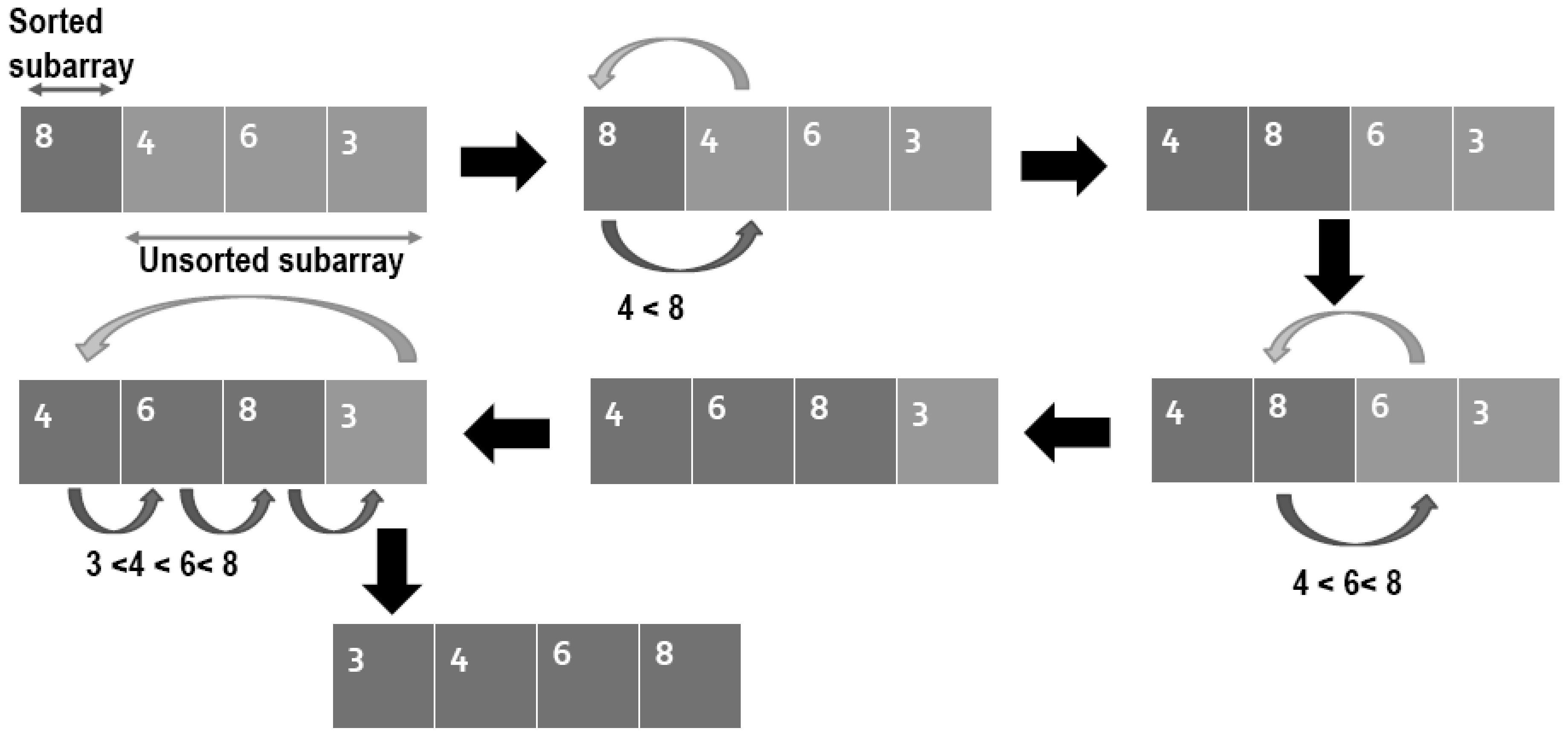
**Next Smallest ??**



## Insertion Sort

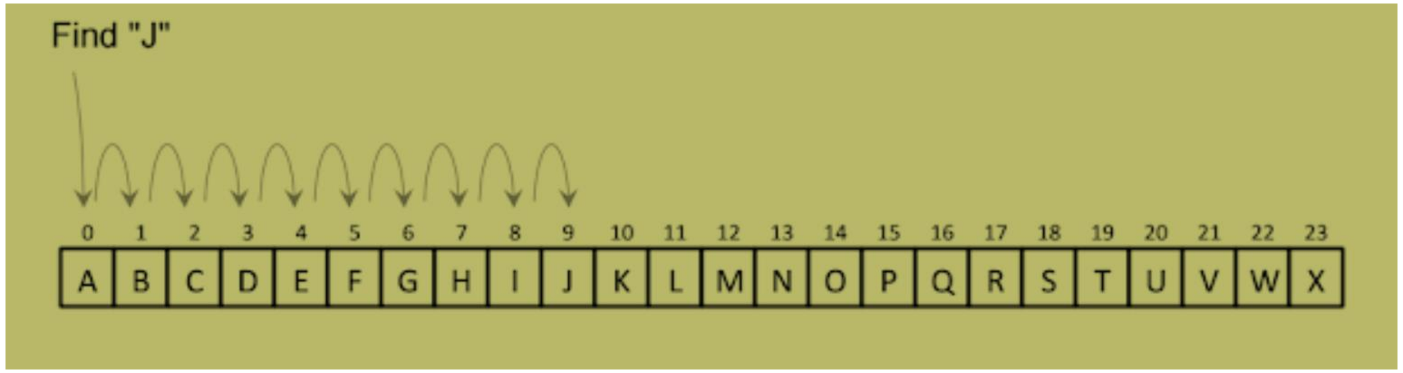
- Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands.
- The array is virtually split into a sorted and an unsorted part.
- Values from the unsorted part are picked and placed at the correct position in the

## Insertion Sort - Example



## Linear Search

- Linear search is used whether the given element is present in an array and if it present then at what location it is present.
- We keep on comparing each element with the element to search until the desired element is found or list ends.



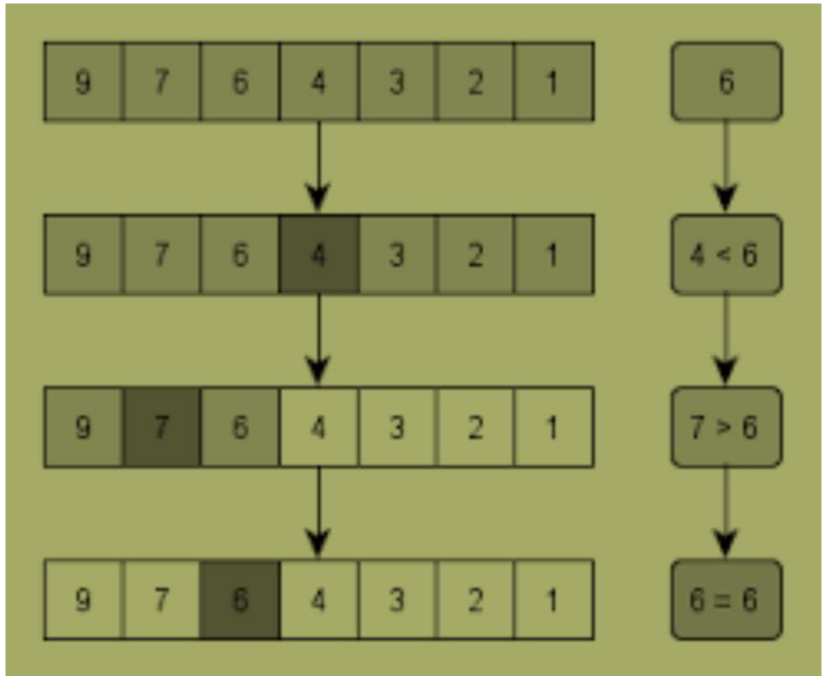
## Linear Search

```
1.[Initialize] Set K:=1 and LOC :=0
2.Repeat Steps 3 and 4 while LOC =0 and K<=N.
3.IF ITEM = DATA[K],then :Set LOC :=K.
4.Set K:= K+1.[Increments counter.]
  [End of Step 2 loop.]
5.[Successful?]
  If LOC =0,then:
  Write: ITEM is not in the array DATA.
  Else:
  Write: LOC is the location of ITEM.
```

A linear array DATA with N elements and a specific ITEM of information are given. This algorithm finds the location LOC of ITEM in the array DATA or sets LOC=0.

## Binary search

- Binary search is an efficient way for finding an item from an ordered list of items.
- It works by repeatedly dividing in half the portion of the list that could contain the item, until you've narrowed down the possible locations to just one.

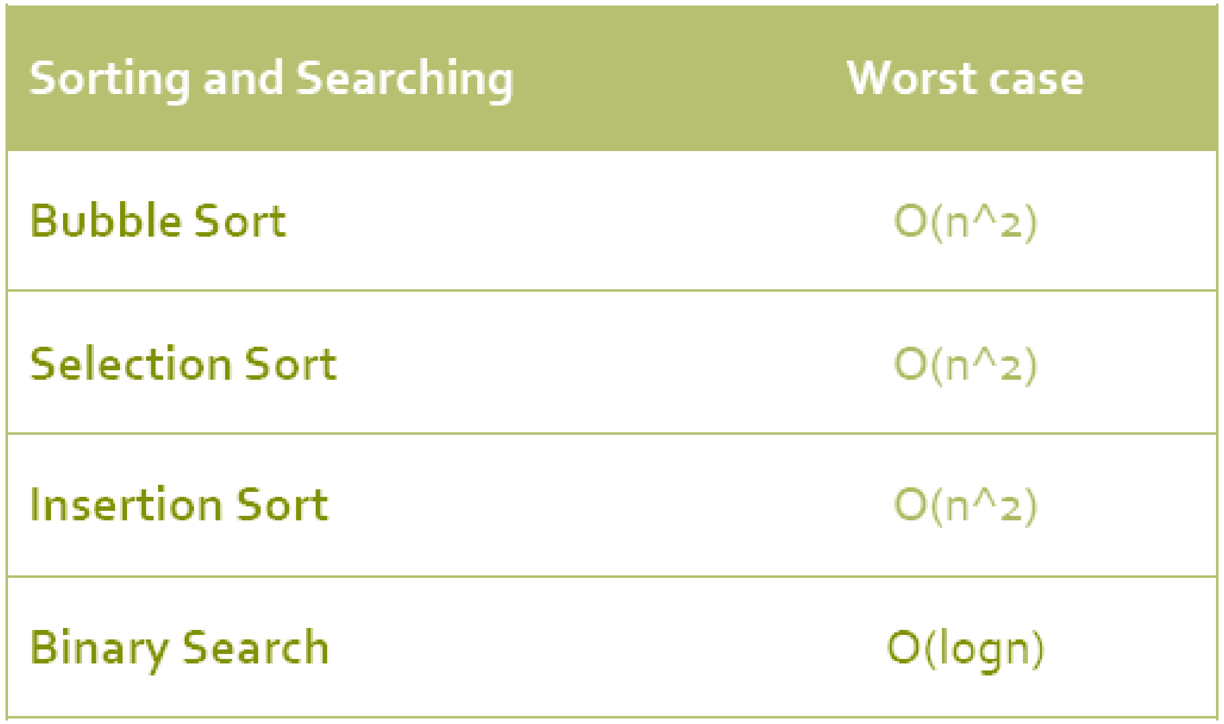


## Algorithm

```
BINARY(DATA,LB,UB,ITEM,LOC)

1.[Initialize segment variable.] Set BEG:=LB, END:=UB, MID:=INT((BEG+END)/2).

2.Repeat step 3 and 4 while BEG <= END and DATA [MID]!=ITEM.

3.If ITEM < DATA[MID]. then:
    Set END:=MID-1.
    Else
    Set BEG:=MID+1.
    [End of if structure.]

4.Set MID:= INT ((BEG+END)/2).
    [End of step 2 loop.]

5.If DATA[MID]=ITEM. then:
    Set LOC:=MID.
    Else
    Set LOC:=NULL.
    [End of if structure.]

6.Exit.
```

Here DATA is a sorted array with lower bound LB and upper bound UB and ITEM is the given item of information. The variables BEG, END and MID denote the beginning, the end and the middle of the segment of the element of DATA. This algorithm find the LOC of ITEM in DATA or sets LOC:=NULL if the search is unsuccessful.

# Complexity



| Sorting and Searching | Worst case |
|---|---|
| Bubble Sort | O(n^2) |
| Selection Sort | O(n^2) |
| Insertion Sort | O(n^2) |
| Binary Search | O(logn) |