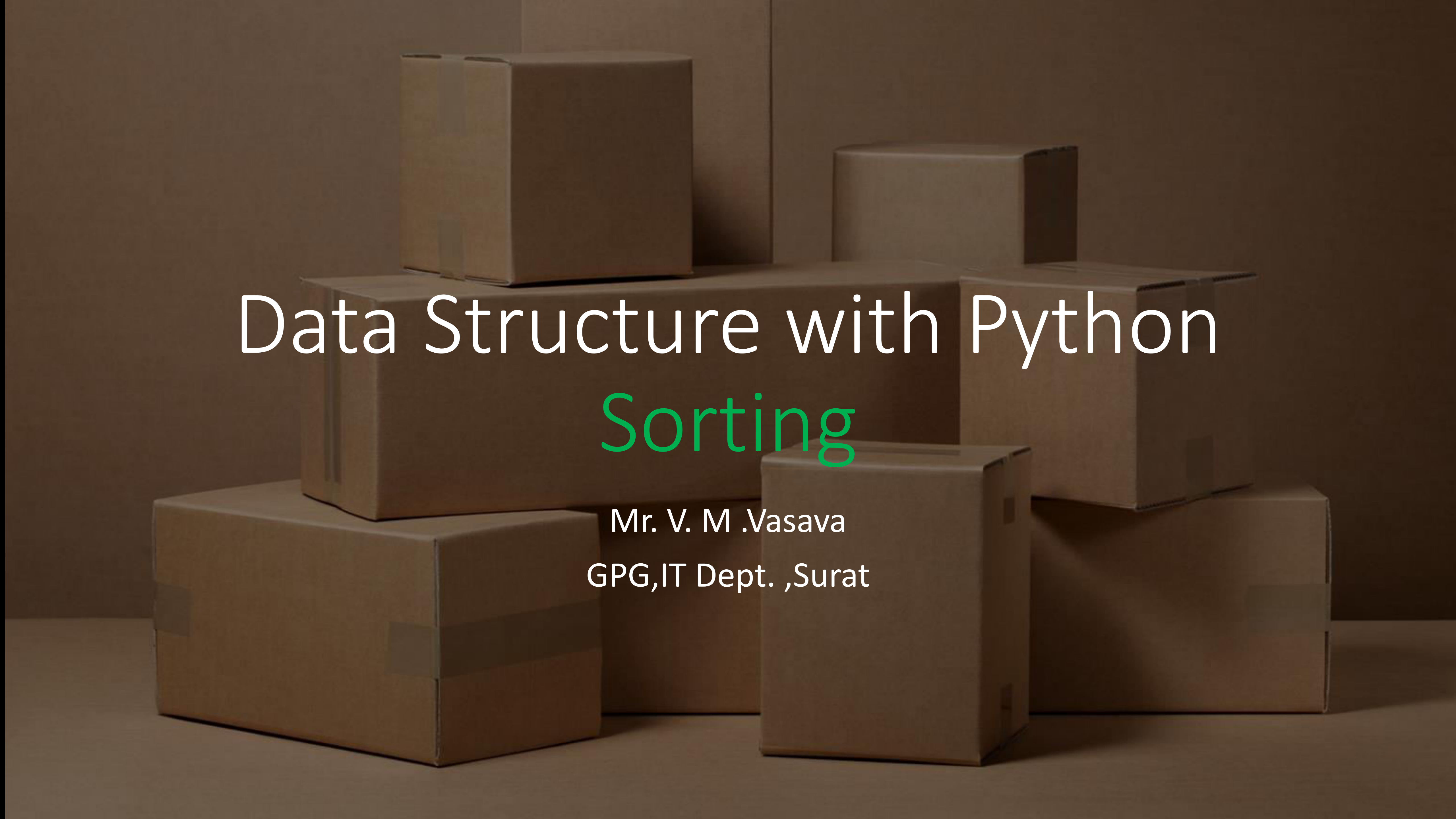


Data Structure with Python
Sorting

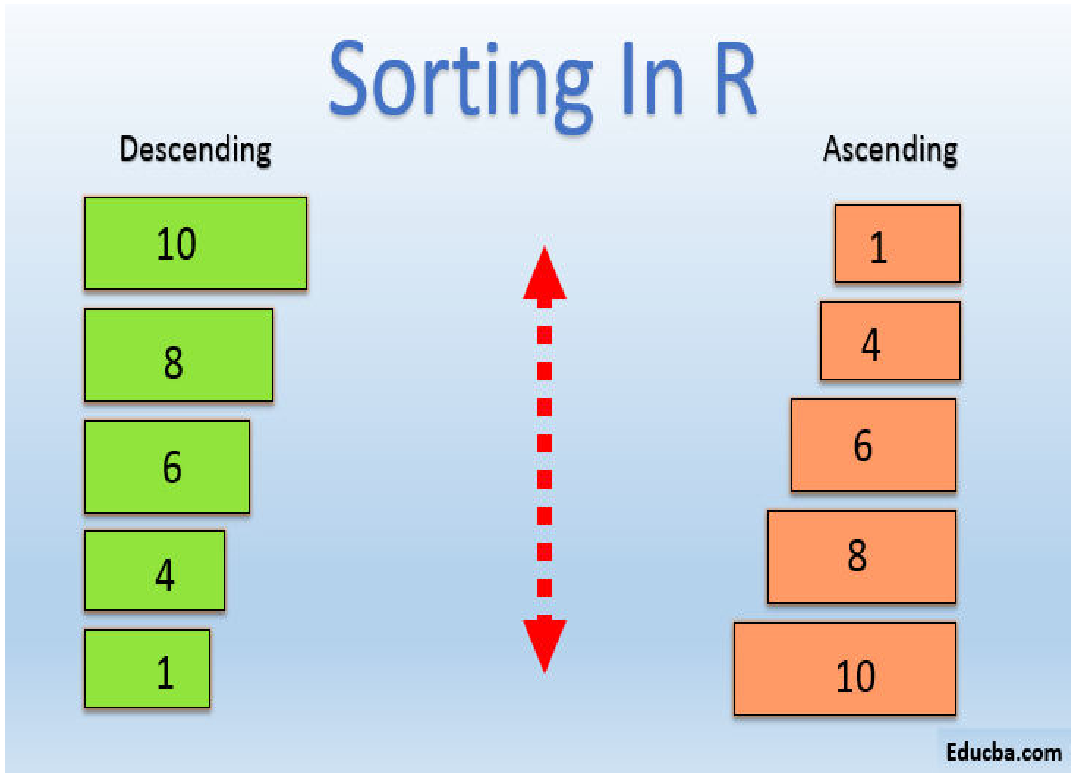
Mr. V. M .Vasava

GPG,IT Dept. ,Surat

Agenda

- Objectives: Sorting
- Why Sorting and Example
- Types of sorting
- Complexity of algorithm

Sorting



- ***Sorting*** is a process that organizes a collection of data into either ascending or descending order.

- Sorting is an arrangement of data in a preferred order either ascending or descending.

Why Sorting

Efficient **sorting** is important for optimizing the efficiency of other algorithms (such as search and merge algorithms) that require input **data** to be in sorted lists.

Sorting is also often useful for canonicalizing **data** and for producing human-readable output.

Poor choice of algorithms affects the runtime of code.

Example : Compare two large sets of items and find out where they differ.

Types of Sorting

1.Internal sort

2.External sort

When all data that needs to be sorted can not be placed in-memory at a time, the sorting is called external sorting.

External Sorting is used for massive amount of data. Merge Sort and its variations are typically used for external sorting. Some external storage like hard-disk, CD, etc is used for external storage.

When all data is placed in-memory, then sorting is called internal sorting.

Types of sorting

1

1.Bubble Sort

2

2.Selection Sort

3

3.Insertion Sort

4

4.Merge Sort

5

5.Quick Sort

6

6.Shell Sort

7

7.Heap Sort

8

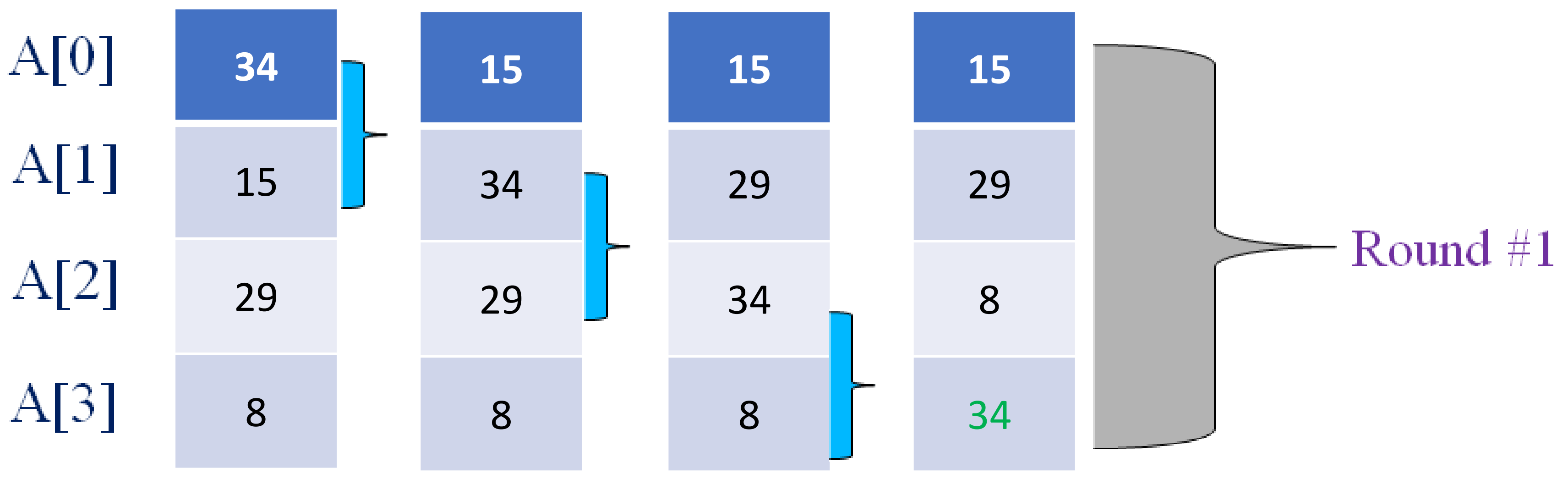
8.Radix Sort or Bucket Sort

Bubble Sort

- Principle: Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.
- This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.
- This algorithm is not suitable for large data sets

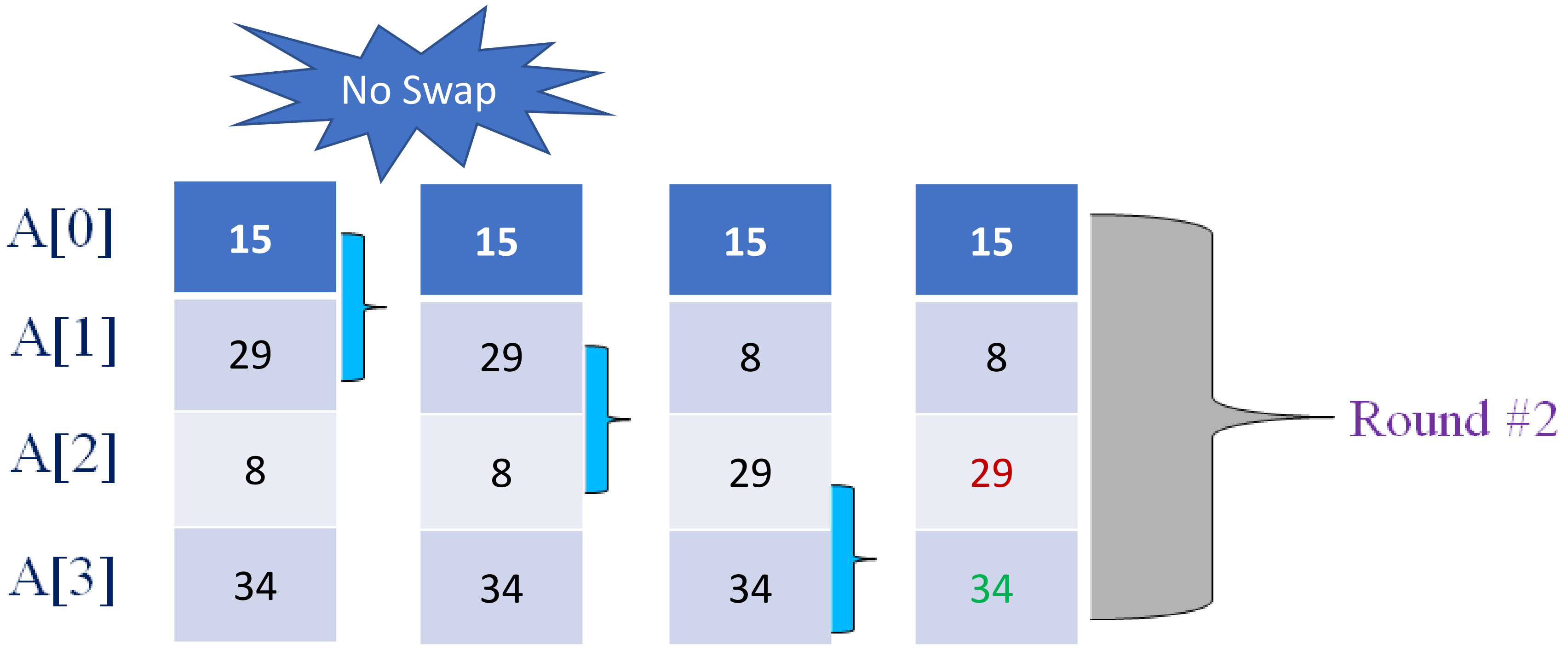
Bubble Sort

Consider Data : 34,15,29,8



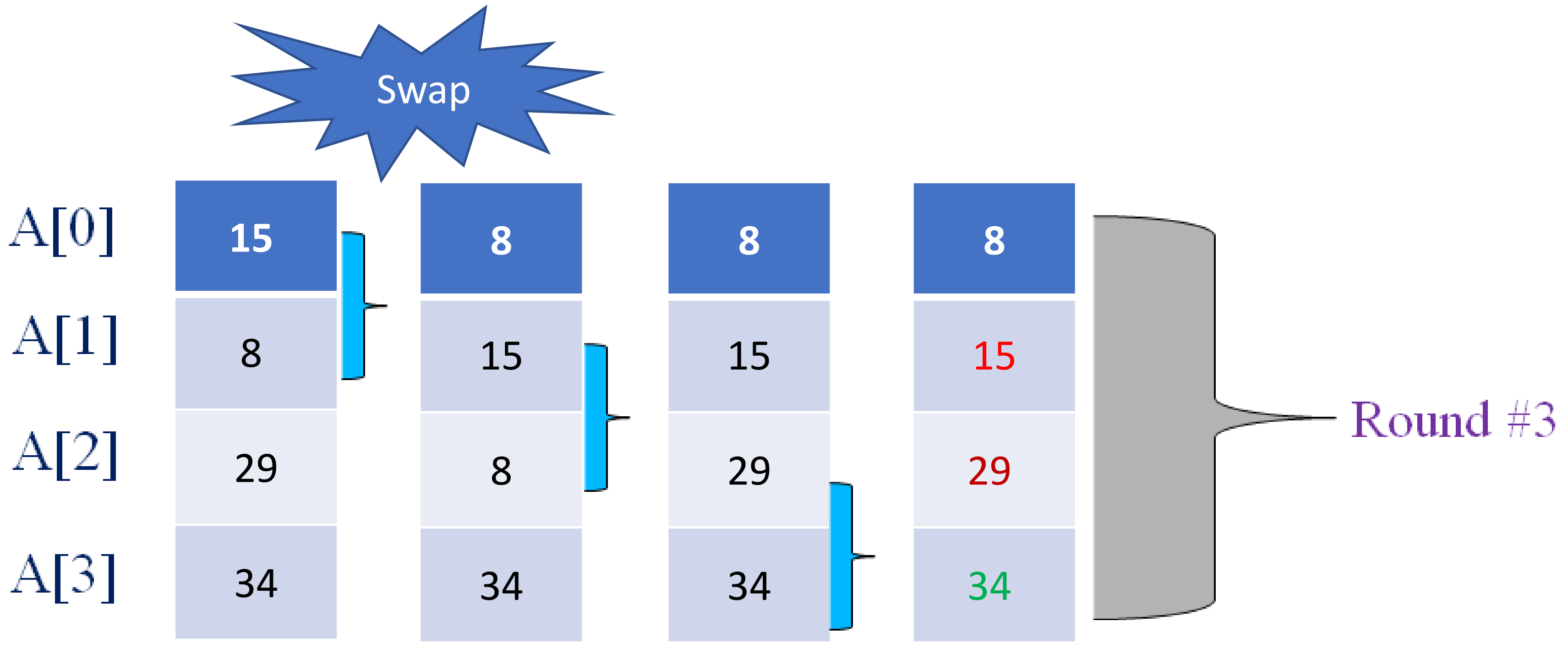
Similarly ,after other consecutive round, this list will be sorted.

Bubble Sort



Similarly ,after other consecutive round, this list will be sorted.

Bubble Sort

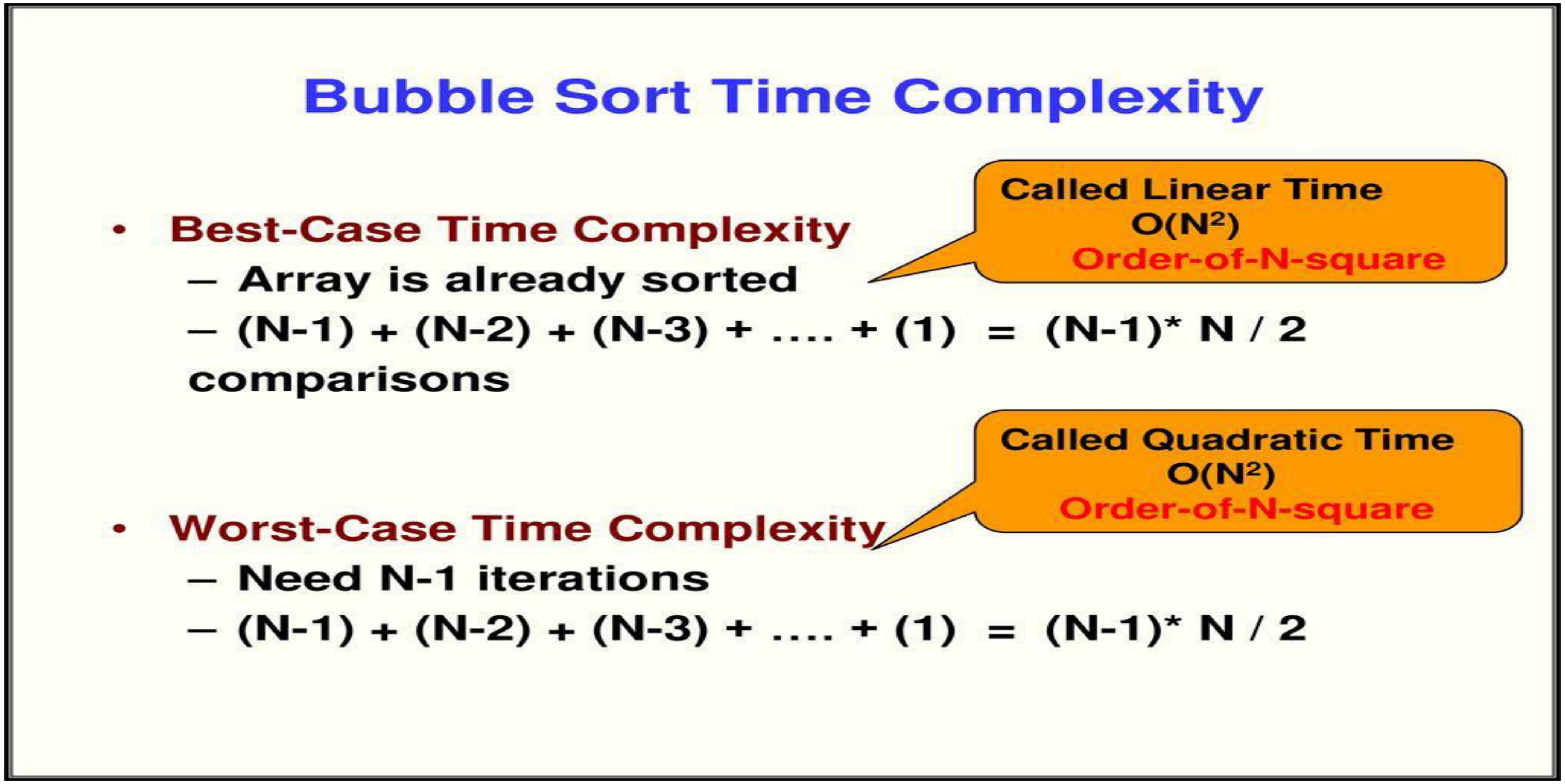


So, after n-1 phase (round), this list will be sorted.

Algorithm:Bubble Sort

```
BUBBLESORT( numList, n)
Step 1: SET i = 0
Step 2: WHILE i< n REPEAT STEPS 3 to 8
Step 3: SET j = 0
Step 4: WHILE j< n-i-1,REPEAT STEPS 5 to 7
            #n -i-1 because last i elements are already sorted in previous passes
Step 5: IF numList[j] > numList[j+1] THEN
Step 6: swap(numList[j],numList[j+1])
Step 7: SET j=j+1
Step 8: SET i=i+1
```

Time Complexity



Bubble Sort Time Complexity

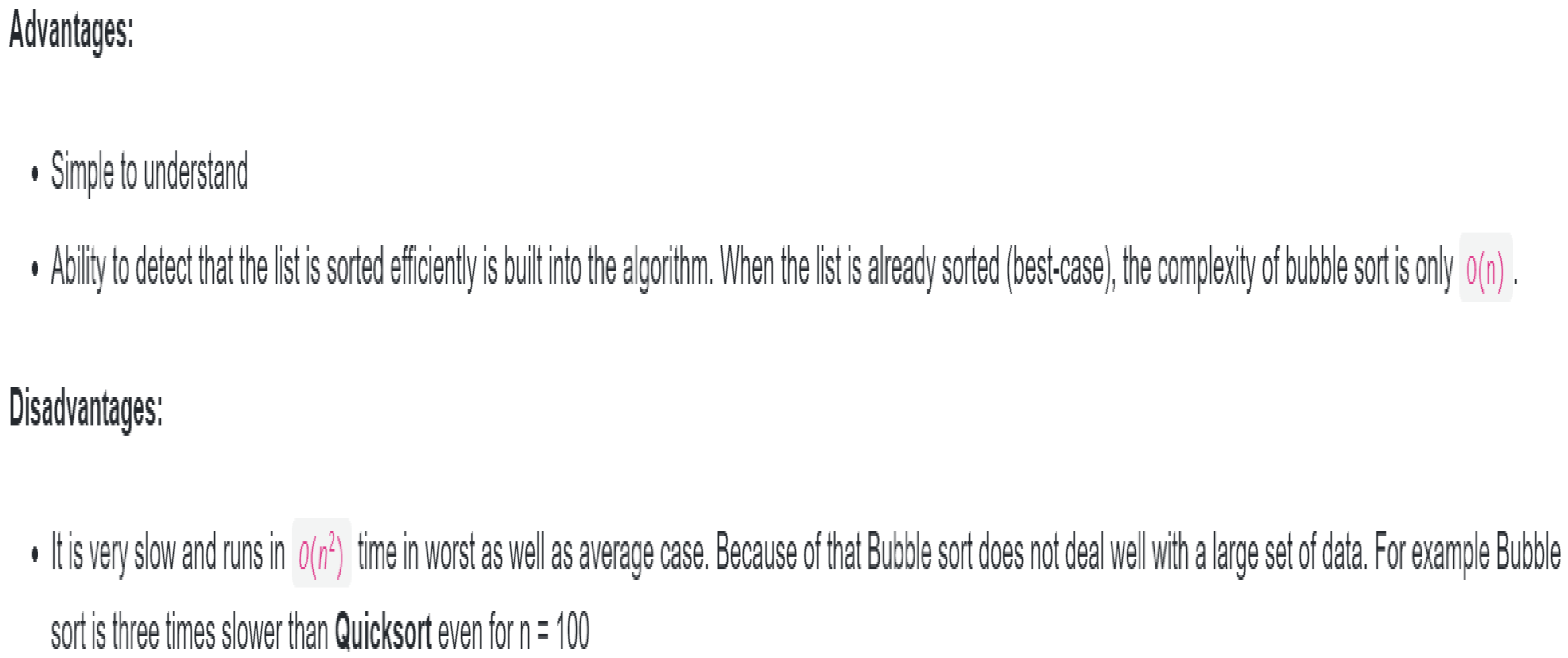
- Best-Case Time Complexity
  - Array is already sorted
  - (N-1) + (N-2) + (N-3) + ... + (1) = (N-1)* N / 2 comparisons

Called Linear Time
$O(N^2)$
Order-of-N-square

- Worst-Case Time Complexity
  - Need N-1 iterations
  - (N-1) + (N-2) + (N-3) + ... + (1) = (N-1)* N / 2

Called Quadratic Time
$O(N^2)$
Order-of-N-square

Advantages & Disadvantages



Advantages:

- Simple to understand
- Ability to detect that the list is sorted efficiently is built into the algorithm. When the list is already sorted (best-case), the complexity of bubble sort is only $O(n)$.

Disadvantages:

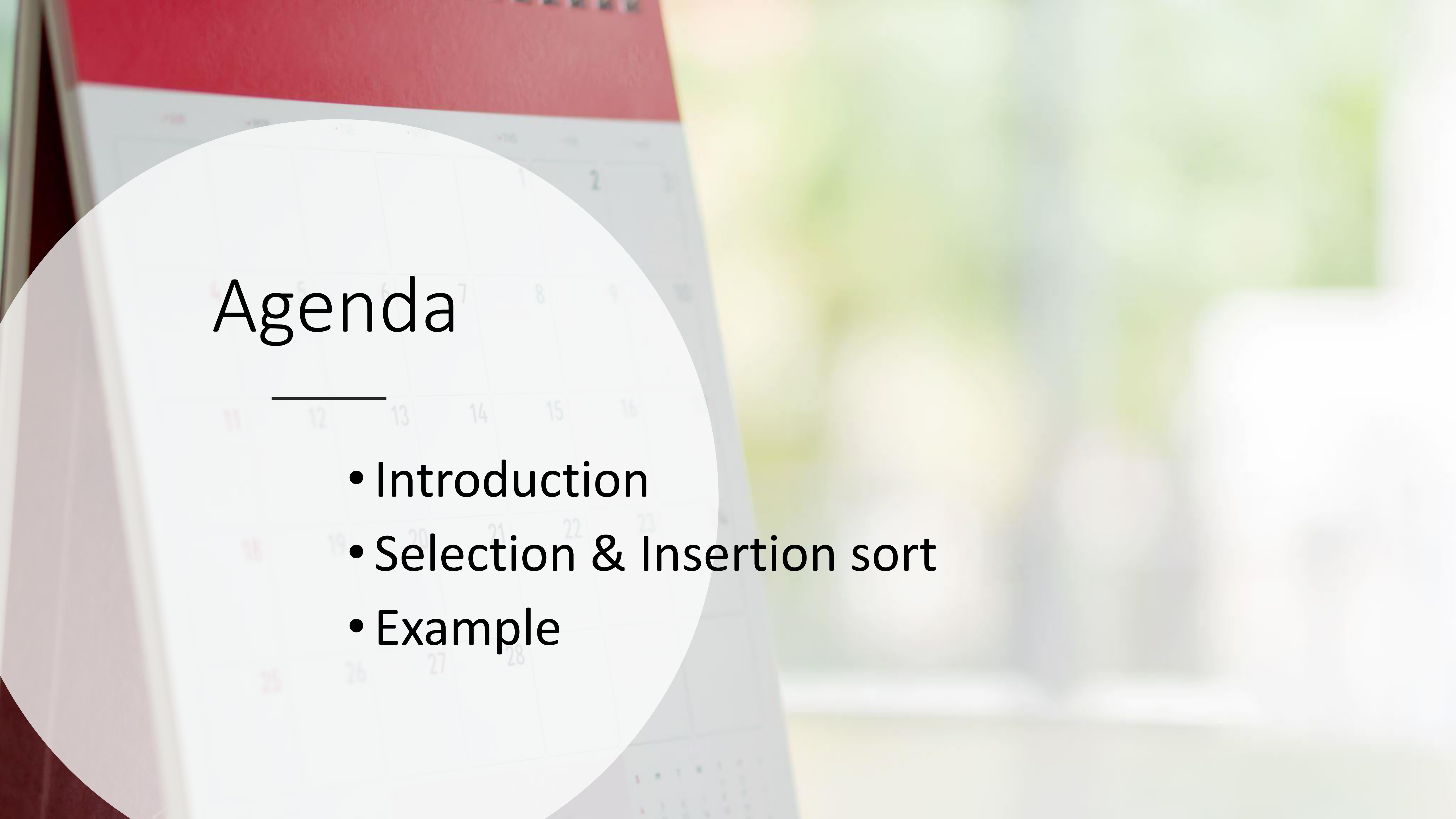
- It is very slow and runs in $O(n^2)$ time in worst as well as average case. Because of that Bubble sort does not deal well with a large set of data. For example Bubble sort is three times slower than **Quicksort** even for n = 100

Any Questions ???

Data Structure with Python
Selection & Insertion Sort

MR. V. M. Vasava

GPG, Surat. IT Dept.



Agenda

- Introduction
- Selection & Insertion sort
- Example

Selection Sorting

Objective: This sorting algorithm is an in-place comparison based algorithm.

The selection sort makes (n-1) number of passes through the list.

The list is considered to be divided into two lists

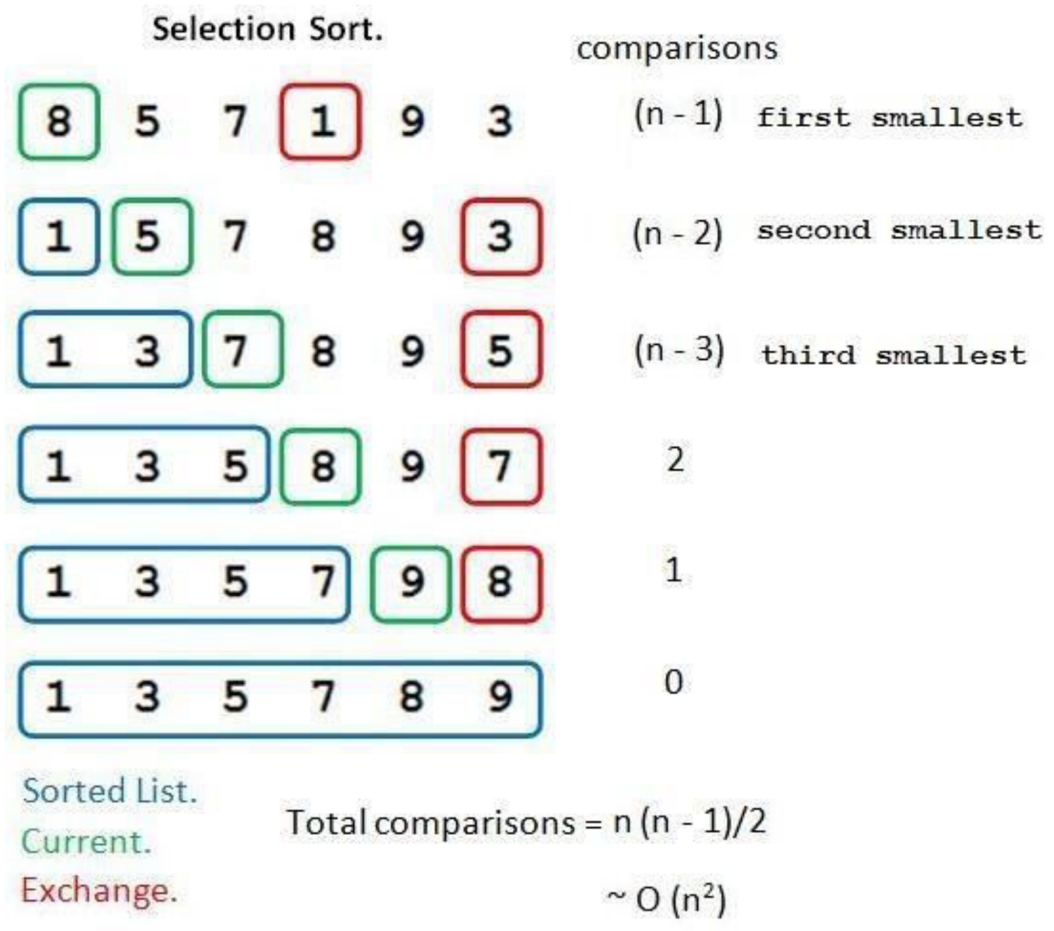
1.The left list containing the sorted elements.

2.The right list containing the unsorted elements.

Initially, the left list is empty, and the right list contains **all the elements.**

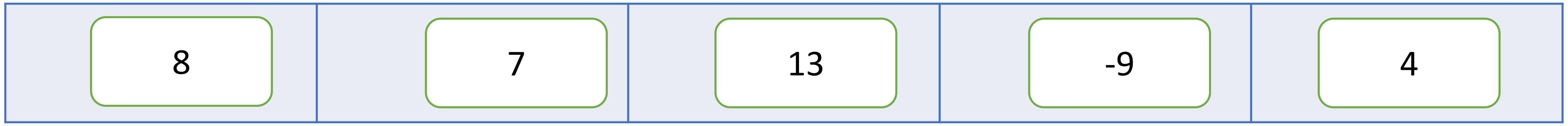
Selection Sorting

- This algorithm will first find the **smallest** element in the list and **swap it** with the element in the **first** position.
- After it will find the **second smallest** element and swap it with the element in the **second** position, and it will keep on doing this until the entire list is sorted.



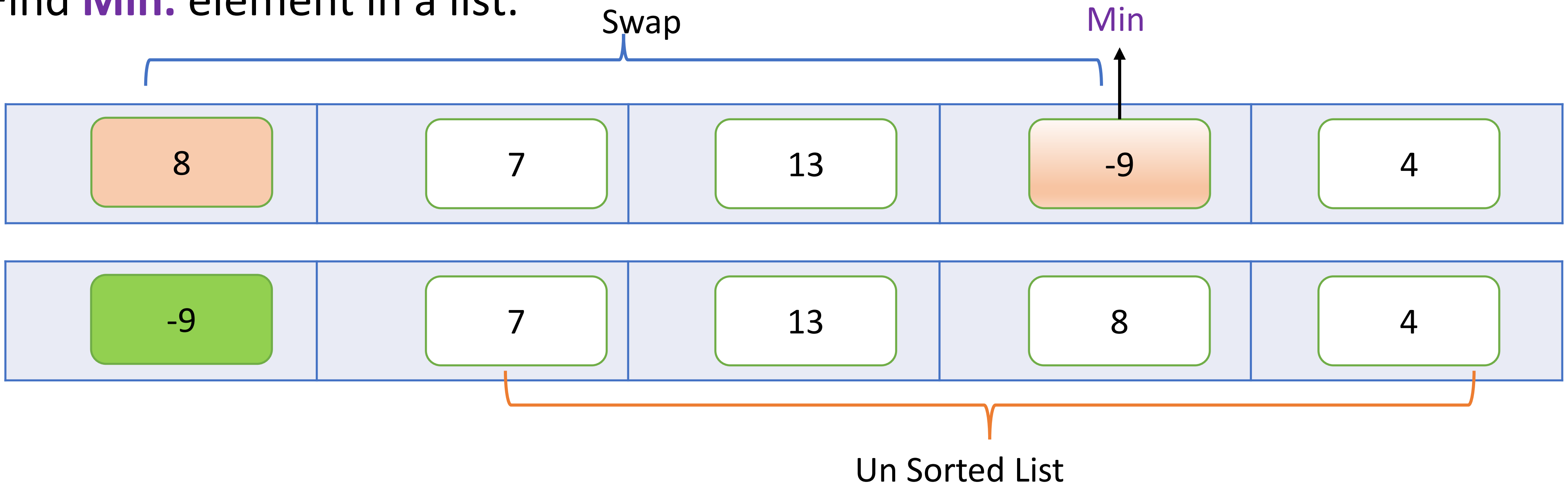
Example

List=



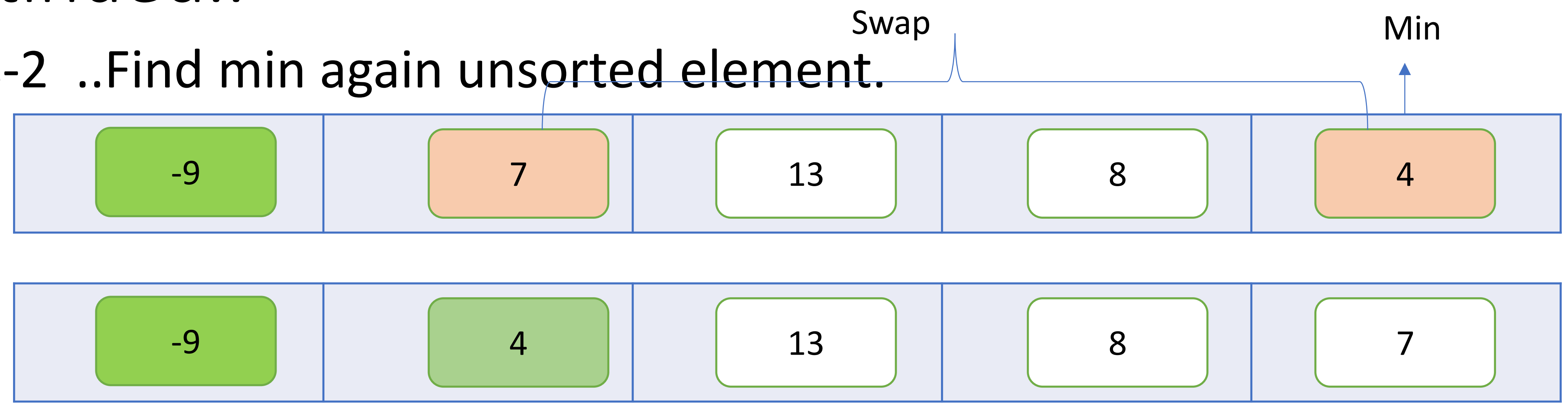
Step-1. Find **Min.** element in a list.

Pass -I

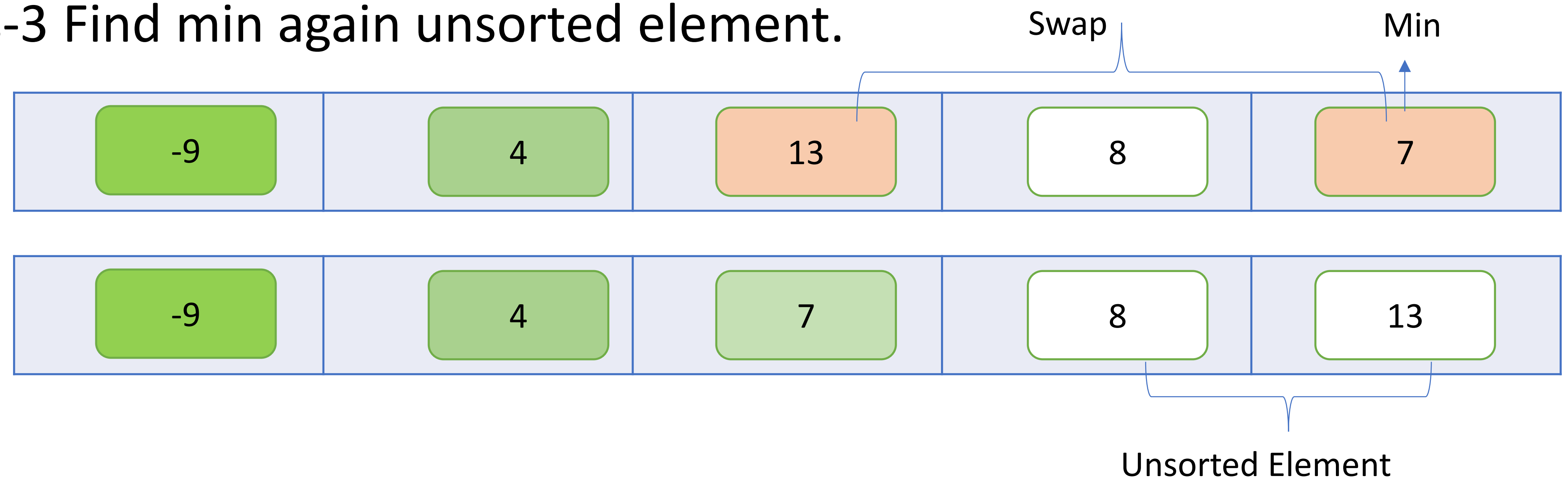


Continued..

- Pass-2 ..Find min again unsorted element.



- Pass-3 Find min again unsorted element.



Continued..

- Pass-4. Find again but no smallest element so There is already sorted.

-9	4	7	8	13
-9	4	7	8	13

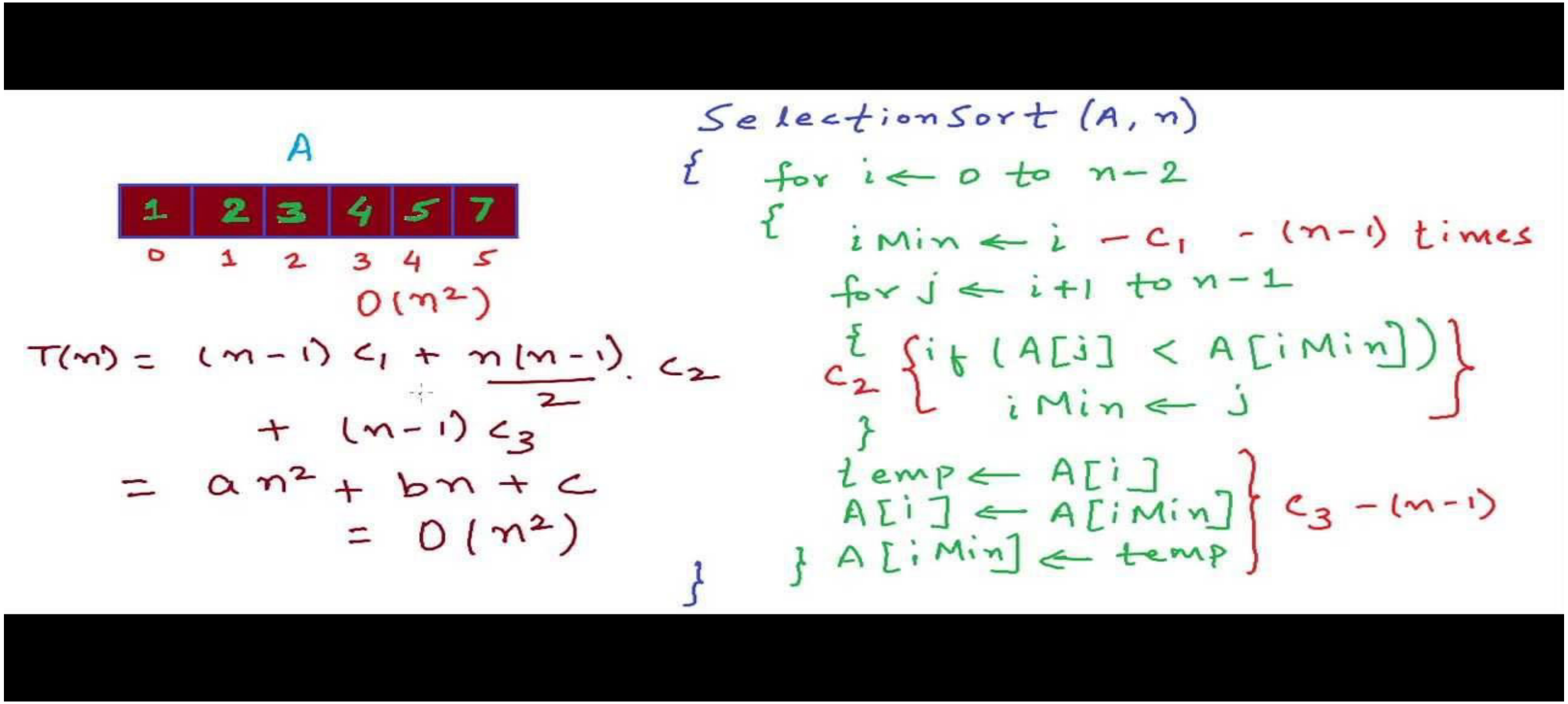
- Example : 24,41,33,42,17
- For i in range(n-1):
-   min =i   assume first element as min.
-     for j in range(i+1,n)
-           if arr[j]<a[min]:
-                 min =j
-                 swap(arr[i],arr[min])

Selection Sort([],n)

- ***SELECTIONSORT(numList, n)***

```
Step 1: SET i=0
Step 2: WHILE i< n REPEAT STEPS 3 to 11   [no. of passes]
Step 3: SET min = i, flag = 0
Step 4: SET j= i+1
Step 5: WHILE j<n, REPEAT STEPS 6 to 10
Step 6: IF numList[j] < numList[min] THEN  [Compare]
Step 7: min = j
Step 8: flag = 1
Step 9: IF flag = 1 THEN
Step 10: swap(numList[i],numList[min])
Step 11: SET i=i+1
```

Analysis of Algorithm



A

1	2	3	4	5	7
0	1	2	3	4	5

$O(n^2)$

$$T(n) = (n-1)c_1 + \frac{n(n-1)}{2} \cdot c_2 + (n-1)c_3$$
$$= an^2 + bn + c$$
$$= O(n^2)$$

```
SelectionSort(A, n)
{   for i ← 0 to n-2
    {
        iMin ← i   - c_1   - (n-1) times
        for j ← i+1 to n-1
        {
      c_2 { if (A[j] < A[iMin])
                iMin ← j          }
        }
        temp ← A[i]               }
        A[i] ← A[iMin]            } c_3 - (n-1)
    }   A[iMin] ← temp            }
}
```

Time Complexity

- To sort an unsorted list with **'n'** number of elements, we need to make **((n-1)+(n-2)+(n-3)+......+1) = (n (n-1))/2** number of comparisions in the worst case.
- If the list is already sorted then it requires **'n'** number of comparisions.
- **Worst Case : O(n^2)**
  Best Case : Ω(n^2)
  Average Case : Θ(n^2)

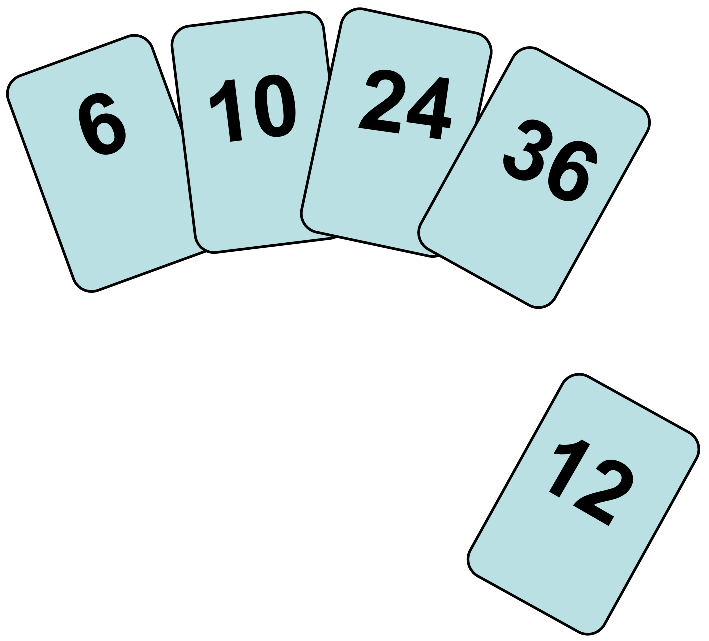
Insertion Sort

- Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands.
- This is an in-place comparison-based sorting algorithm.
- The insertion sort algorithm is performed using the following steps...
- **Step 1** - Assume that first element in the list is in sorted portion and all the remaining elements are in unsorted portion.
- **Step 2:** Take first element from the unsorted portion and insert that element into the sorted portion in the order specified.
- **Step 3:** Repeat the above process until all the elements from the unsorted portion are moved into the sorted portion.

Insertion Sort

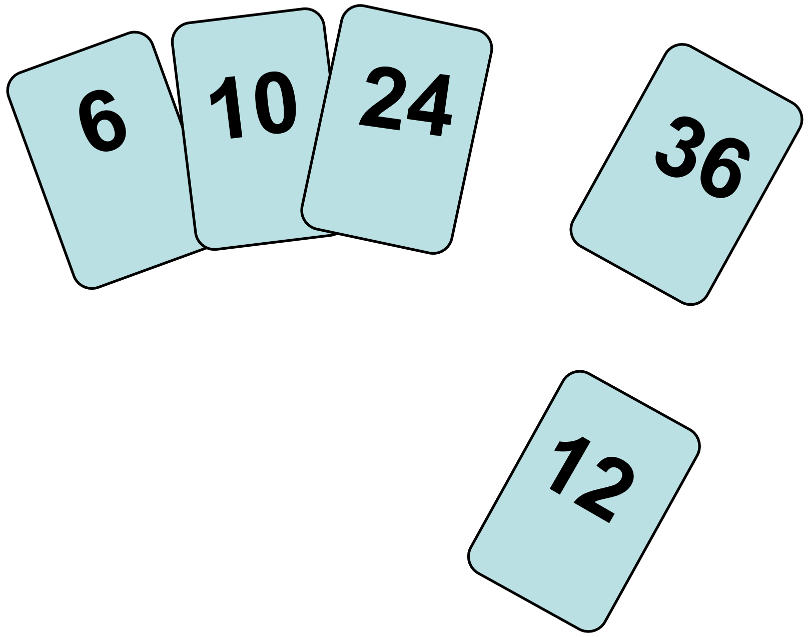
- Idea: like sorting a hand of playing cards
  - Start with an empty left hand and the cards facing down on the table.
  - Remove one card at a time from the table, and insert it into the correct position in the left hand
    - compare it with each of the cards already in the hand, from right to left
  - The cards held in the left hand are sorted
    - these cards were originally the top cards of the pile on the table

Insertion Sort

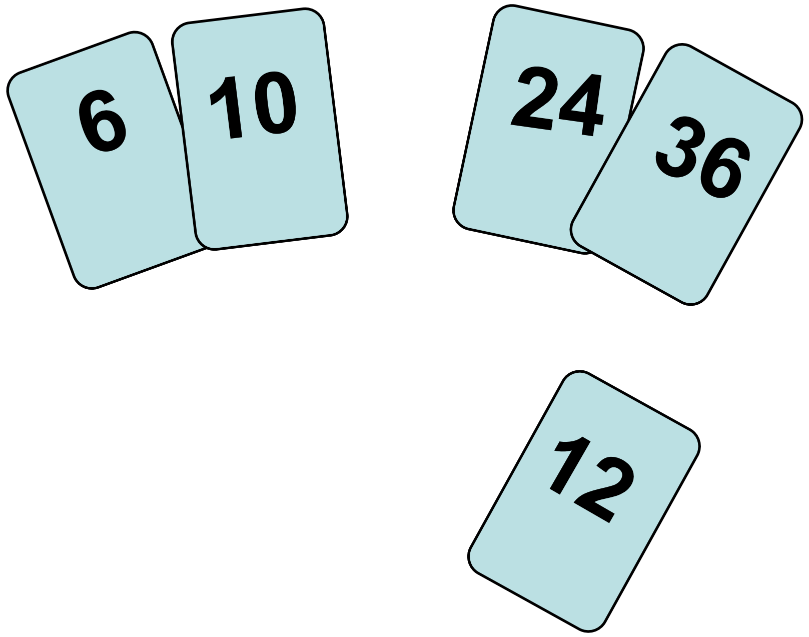


To insert 12, we need to make room for it by moving first 36 and then 24.

Insertion Sort



Insertion Sort



Insertion Sort

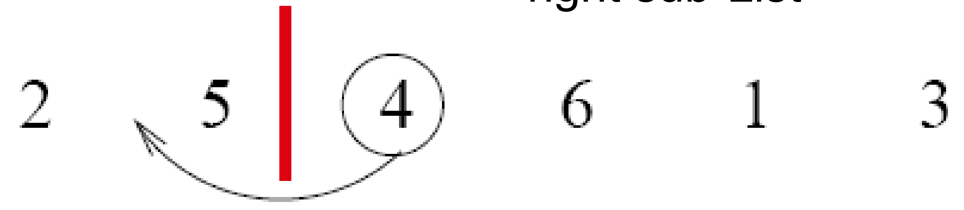
input List

5 2 4 6 1 3

at each iteration, the List is divided in two sub-List:

left sub-List

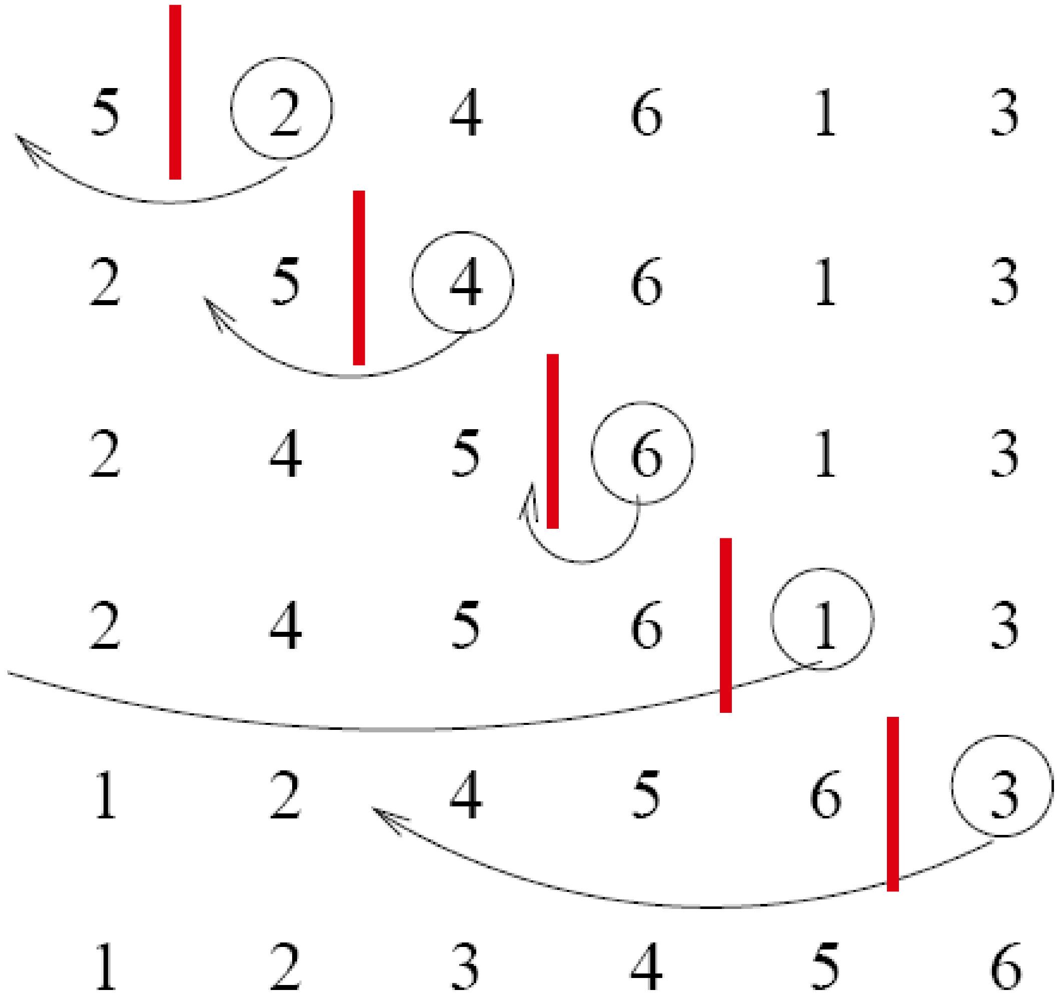
right sub-List

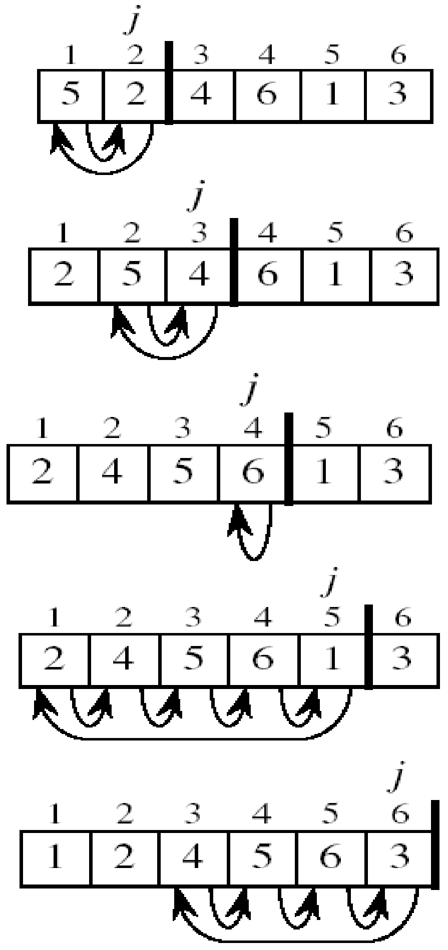


sorted

unsorted

Insertion Sort





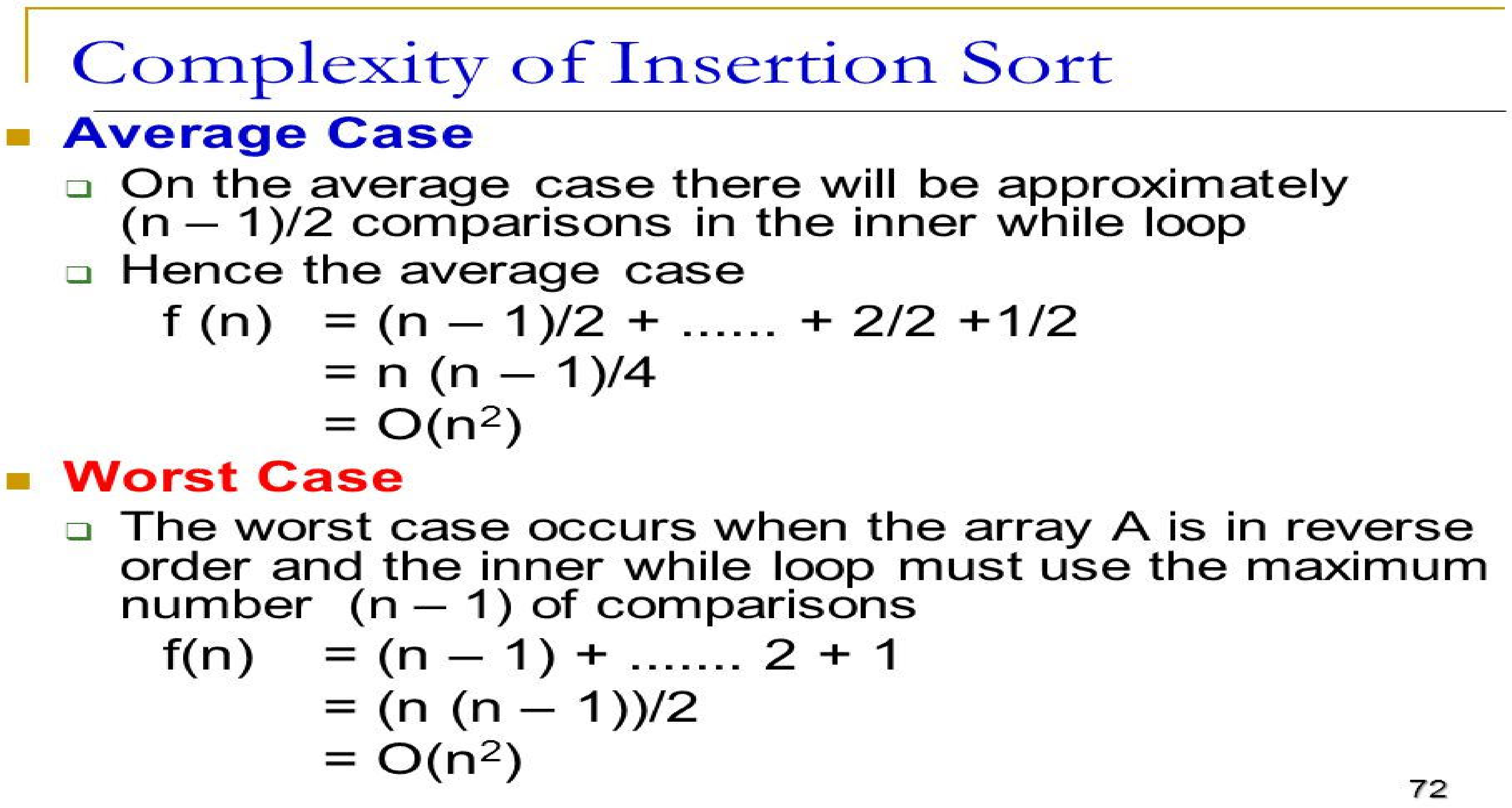
Algorithm :: insertion sort

```
INSERTIONSORT( numList, n)
Step 1: SET i=1
Step 2: WHILE i< n REPEAT STEPS 3 to 9
Step 3: temp = numList[i]
Step 4: SET j = i-1
Step 5: WHILE j> = 0 and temp<numList[j],REPEAT STEPS 6 to 7
Step 6: numList[j+1] = numList[j]
Step 7: SET j=j-1
Step 8: numList[j+1] = temp #insert temp at position j
Step 9: set i=i+1
```

Algorithm: Insertion Sort

```
def InsertionSort(a):   # traversing the array from 1 to length of the list(a)
        for i in range(1, len(a)):
                temp = a[i]    # Shift elements of list[0 to i-1], that are
      # greater than temp, to one position ahead and of their current position
                j = i-1
                while j >=0 and temp < a[j] :
                        a[j+1] = a[j]
                        j -= 1
                        a[j+1] = temp
        return a
```

Time Complexity



Complexity of Insertion Sort

- **Average Case**
  - On the average case there will be approximately $(n - 1)/2$ comparisons in the inner while loop
  - Hence the average case

    $$f(n) = (n - 1)/2 + \ldots\ldots + 2/2 + 1/2$$
    $$= n(n - 1)/4$$
    $$= O(n^2)$$
- **Worst Case**
  - The worst case occurs when the array A is in reverse order and the inner while loop must use the maximum number $(n - 1)$ of comparisons

    $$f(n) = (n - 1) + \ldots\ldots\ 2 + 1$$
    $$= (n(n - 1))/2$$
    $$= O(n^2)$$

72

Example: solve data using Insertion sort & Selection sort.

34, 15,29,8,17

Any Question ?????

Data Structure with Python

Mr. V. M. Vasava

GPG,IT Dept.

Agenda

- Introduction
- Types of search
- Linear Search
- Binary Search

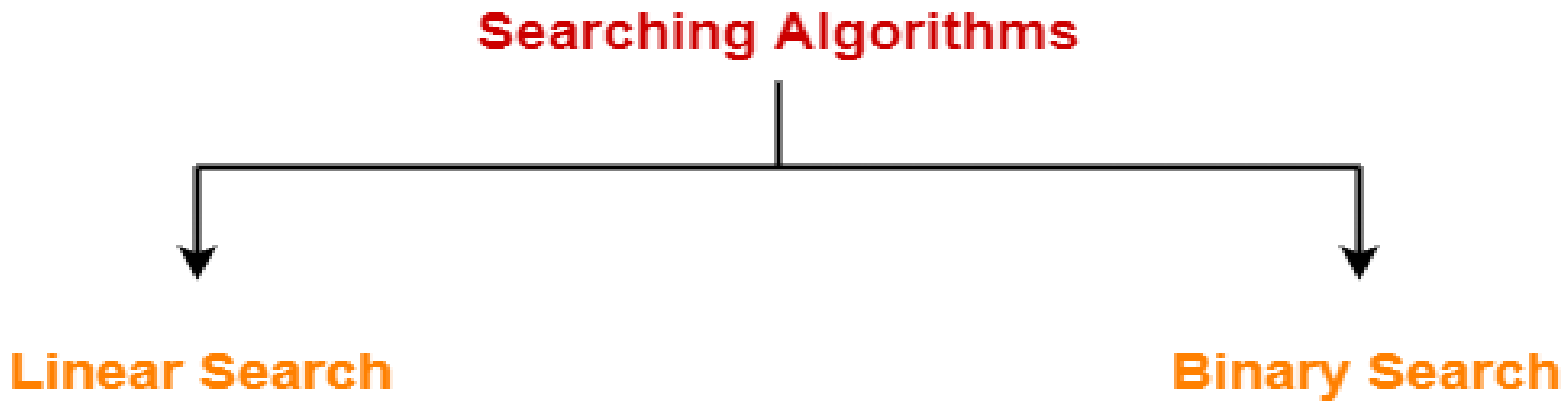
Searching

Searching is a process of finding a particular element among several given elements.

The search is successful if the required element is found.

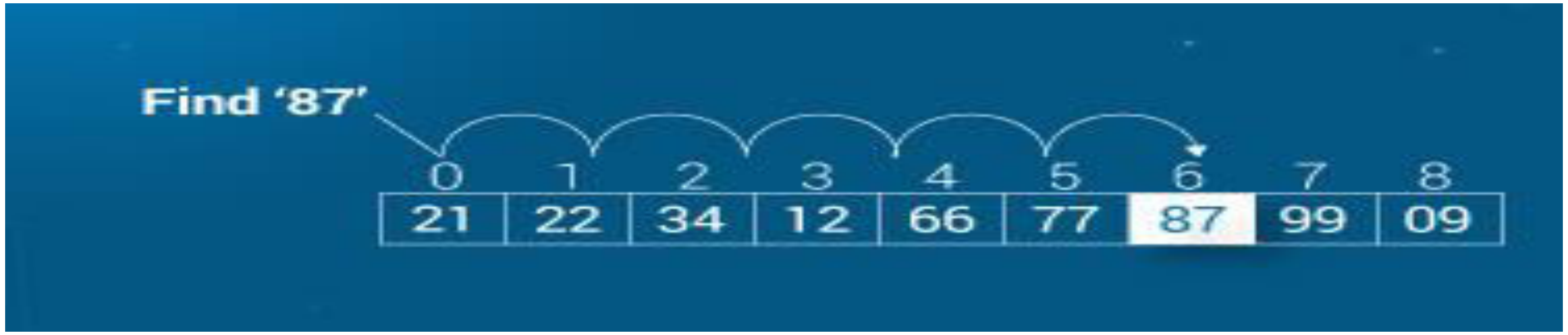
Otherwise, the search is unsuccessful.

Types of search



Linear Search

- Linear Search is the simplest searching algorithm.
- It traverses the **List sequentially** to locate the required element.
- It searches for an element by comparing it with each element of the List **one by one**. So, it is also called as **Sequential Search**.
- Linear Search or sequential search is a method for finding a target value within list.
- It sequentially checks each element of the list for target value until a match is found or until all the elements have been searched.



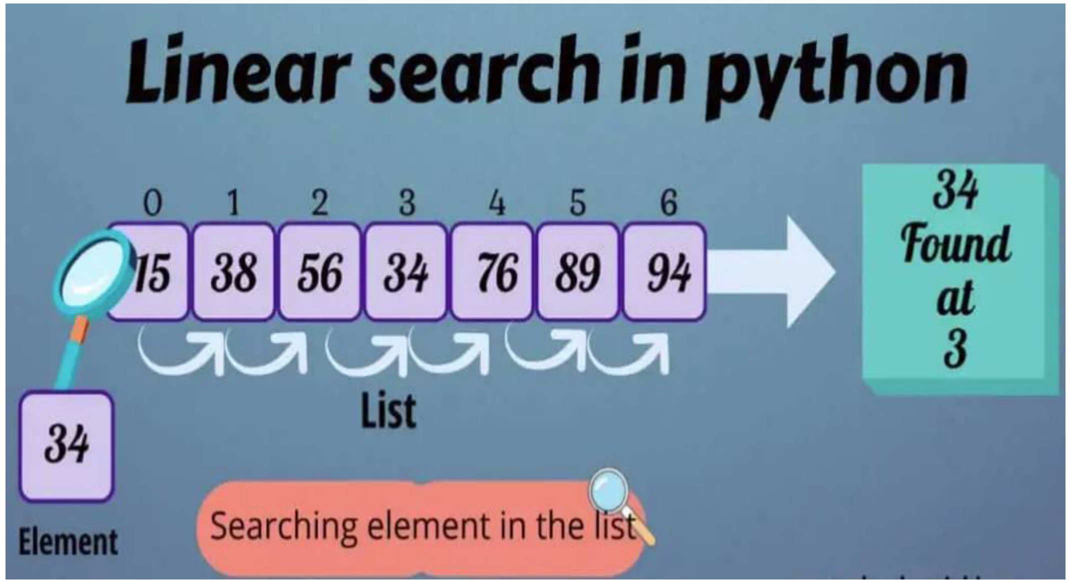
Linear Search

- Given an list of n elements, write a function to search a given element n in list.

```
def search(list1,key):
    for i in range(len(list1)): [Traverse all ele. Until notfound]
        if key==list1[i]: [Compare one by one ele.]
            print("Element found at index: ", i)
            break
    else:
        print("Element not found")
```

Algorithm : Linear Search

1. Input a list of numbers.
2. Input the item to be searched.
3. Compare that element with each and every element of the list one by one.
4. If the match is found, then return index.
5. If the match is not found in the whole list, then return False.



Time Complexity

Best case-

In the best possible case, The element being searched may be found at the first position.

In this case, the search terminates in success with just one comparison.

Thus in best case, linear search algorithm takes O(1) operations.

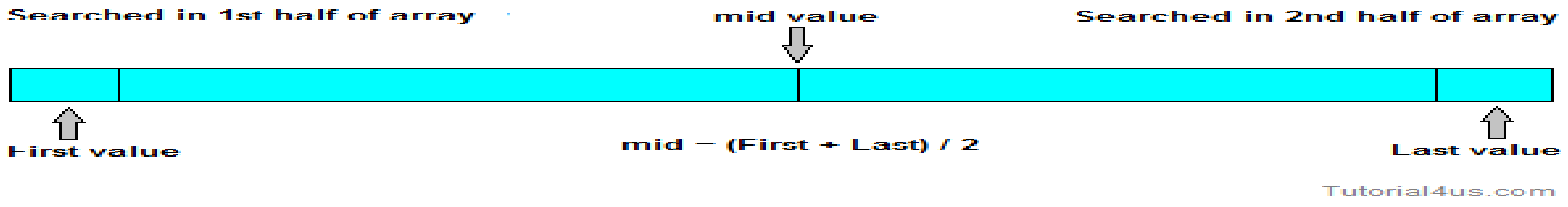
Worst Case-

The element being searched may be present at the last position or not present in the list at all.

Time Complexity O(n)

Binary Search

- Binary Search is applied on the sorted list .
- Binary search follows divide and conquer approach in which, the list is divided into two halves .
- 1. First find the middle element of List.
- 2. Compare the mid element with an item.
- 3. There are three cases:
  - A).If it is desired element then search is successful.
  - B). If it is < desired item then search only first half of list.
  - C). If it is > desired item then search only second half of list.

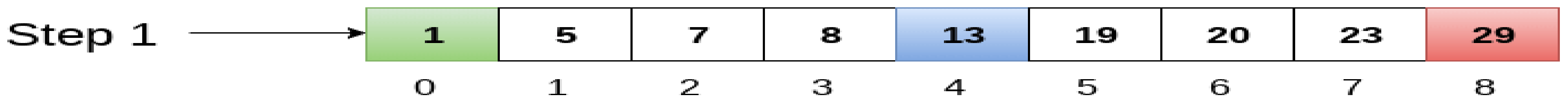


BinarySearch(List,Item)

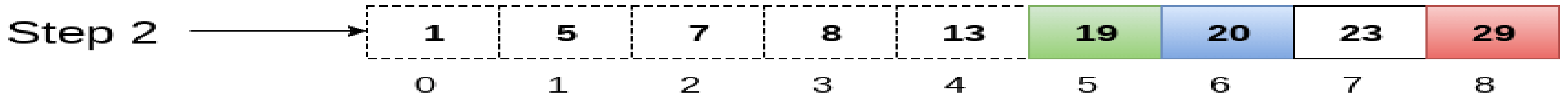
```
DEF BINARY_SEARCH(ITEM_LIST,KEY):
1. [initialize var.]
            FIRST = 0
            LAST = LEN(ITEM_LIST)-1
            FOUND = FALSE
2.          WHILE( FIRST<=LAST AND NOT FOUND): [ CHECK FOUND OR NOT TILL ALL ELEMENT]
                  MID = (FIRST + LAST)//2
                  IF KEY==ITEM_LIST[MID] : [COMPARE KEY WITH MID ELEMENT]
                        FOUND = TRUE
                  ELIF KEY < ITEM_LIST[MID]: [COMPARE KEY WITH FIRST SUBLIST]
                        LAST = MID - 1
                  ELSE:                     [COMPARE KEY WITH SECOND SUBLIST]
                        FIRST = MID + 1
3.          RETURN FOUND
```

Example

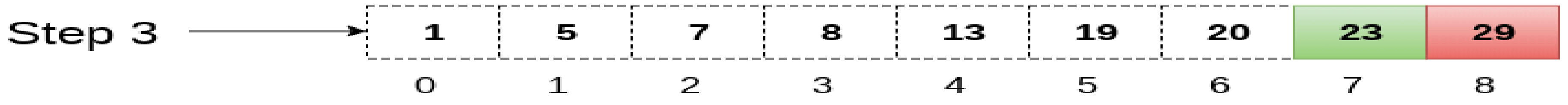
Item to be searched = 23



a [mid] = 13
13 < 23
beg = mid + 1 = 5
end = 8
mid = (beg + end)/2 = 13 / 2 = 6



a [mid] = 20
20 < 23
beg = mid + 1 = 7
end = 8
mid = (beg + end)/2 = 15 / 2 = 7



a [mid] = 23
23 = 23
loc = mid

Return location 7

Example:

- 1. Find data 25 using Binary search below data :

12	14	16	21	25	29

- 2. Solve Below data using Linear search , Find data 16.

10	11	5	21	16	29

Time Complexity

1	**Worst case**	**O(log n)**
2	Best case	O(1)
3	Average Case	O(log n)

Any Questions ??????

Data Structure
Quick sort

Mr. V. M .Vasava

GPG, IT Dept. Surat

Agenda

Introduction

Quick Sorting

Example

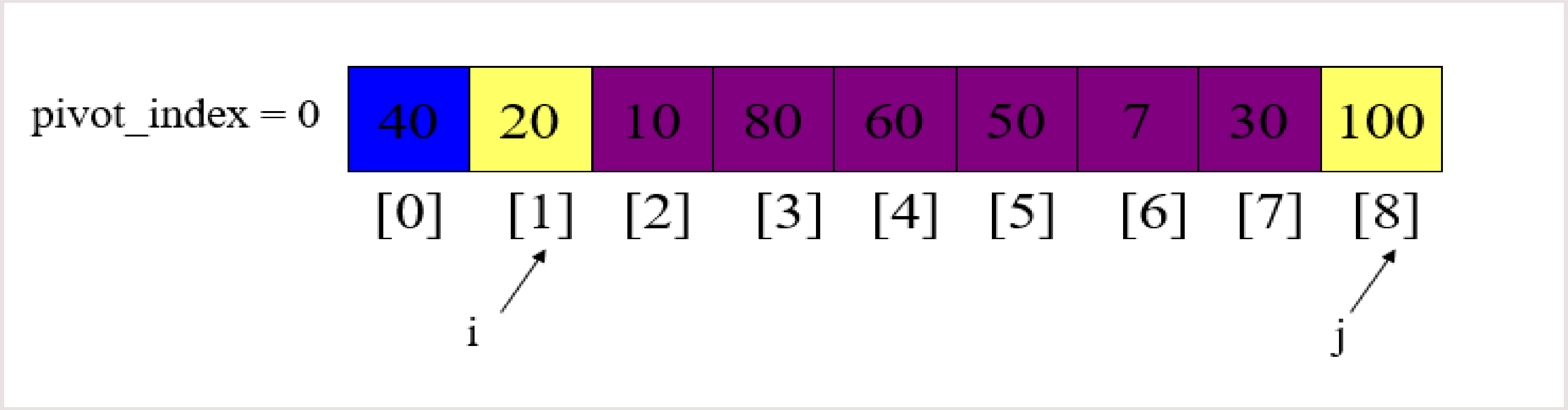
Time complexity

Introduction

- Quick sort is also known as **Partition-exchange sort** based on the rule of **Divide and Conquer.**
- Quick sort algorithm is invented by **C. A. R. Hoare**.
- It is a highly efficient sorting algorithm.
- Quick sort is the quickest comparison-based sorting algorithm.
- It is very fast and requires less additional space, only O(n log n) space is required.
- Quick sort picks an element as pivot and partitions the array around the picked pivot.

Quick Sort

+ This algorithm divides the list into three main parts:
+ 1. Elements less than the Pivot element
+ 2. Pivot element
+ 3. Elements greater than the pivot element



Working of quick sort

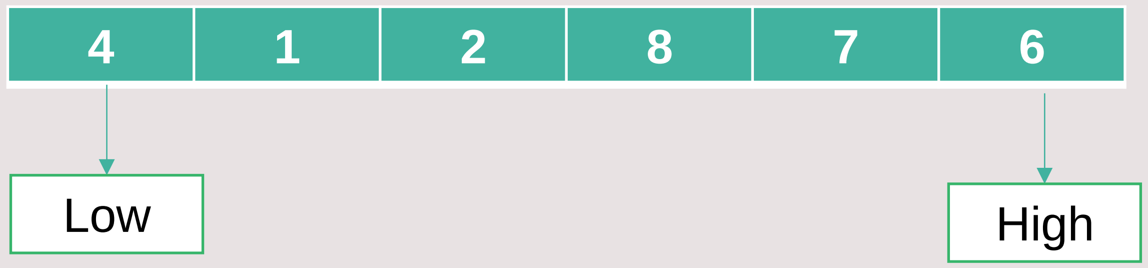
+ In Quick sort algorithm, partitioning of the list is performed using following steps...
+ Step 1 - Consider the first element of the list as **pivot** (i.e., Element at first position in the list).
+ Step 2 - Define two variables i and j. Set i and j to first and last elements of the list respectively.
+ Step 3 - Increment i until A[i] > pivot then stop.
+ Step 4 - Decrement j until A[j] < pivot then stop.
+ Step 5 - If i < j then exchange A[i] and A[j].
+ Step 6 - Repeat steps 3,4 & 5 until i > j.
+ Step 7 - Exchange the pivot element with A[j] element.

Algorithm :Quick_sort(a[],n)

Recursively sort the two partitions

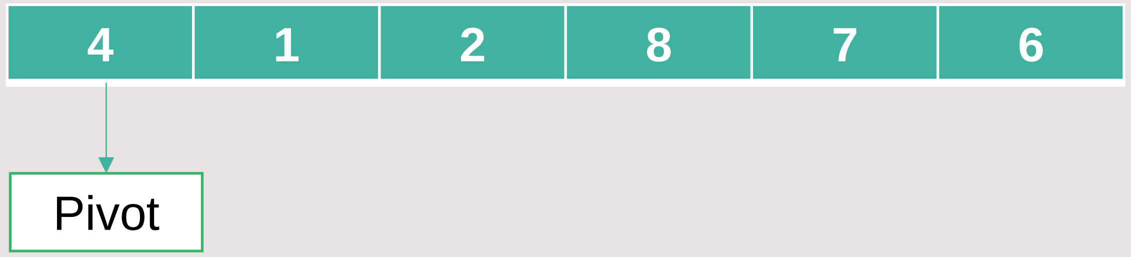
```
def quick_sort(array, low, high):
    if low>= high:
        return

    pivot = partition(array, low, high)
    quick_sort(array, low, pivot-1)
    quick_sort(array, pivot+1, high)
```



Partition(arr,low,high)

```
P=arr[low],i=low+1,j=high     #initialize variable
While True
        while i<=j and arr[i]<=p:
                i+=1
        while i<=j and arr[j]>=p:
                j-=1
        if i<=j:
            arr[i],arr[j]=arr[j],arr[i]  [#Swap between i & j index value]
        else:
                break
Arr[low],arr[j]=arr[j],arr[low] [#Swap between pivot & j index value]
Return j
```



+Any questions???

Data Structure with Python Merge Sort

Mr. V. M. Vasava

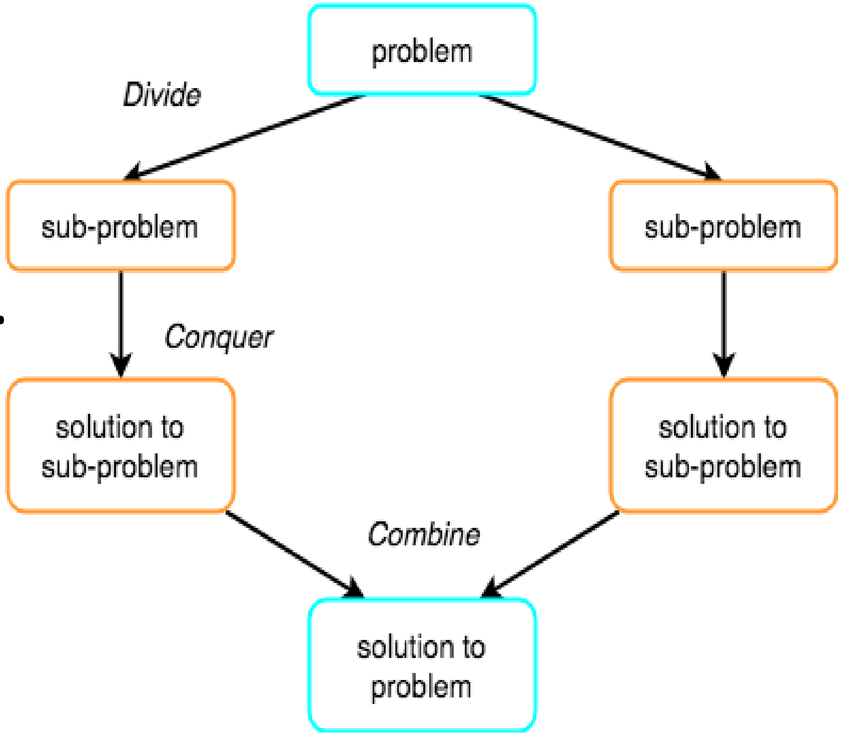
GPG,IT Dept. Surat

Agenda

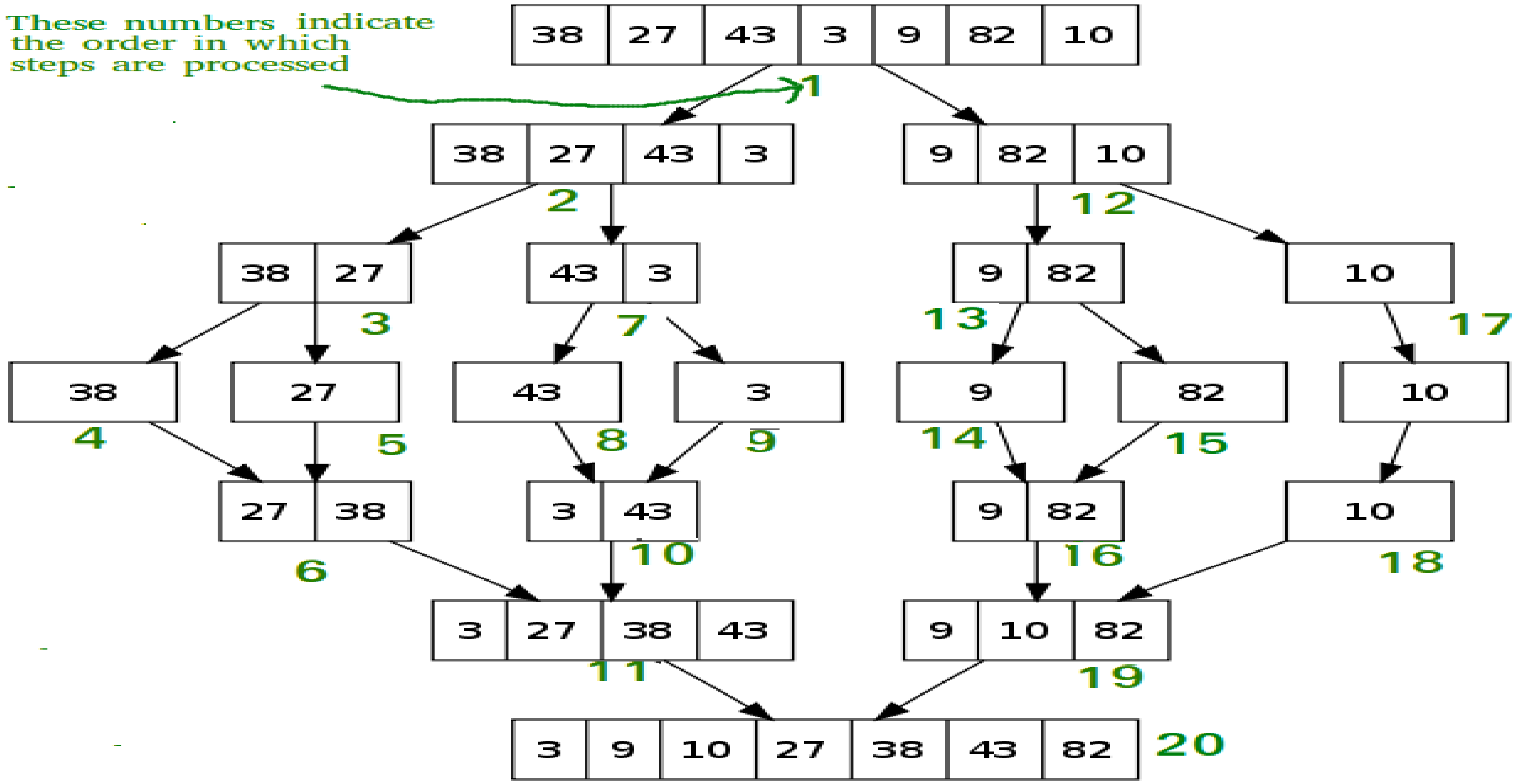
- Introduction
- Working of Merge Sort
- Merge Sort Algorithm
- Example
- Time complexity

Introduction

- Merge sort is a sorting technique based on divide and conquer technique.
- The concept of Divide and Conquer involves three steps:
- **Divide** the problem into multiple small problems.
- **Conquer** the subproblems by solving them. The idea is to break down the problem into atomic subproblems, where they are actually solved.
- **Combine** the solutions of the subproblems to find the solution of the actual problem.
- Merge sort runs in `O(n*log n)` time in all the cases.



Example


These numbers indicate the order in which steps are processed
38 27 43 3 9 82 10
1
38 27 43 3
9 82 10
2
12
38 27
43 3
9 82
10
3
7
13
17
38
27
43
3
9
82
10
4
5
8
9
14
15
27 38
3 43
9 82
10
6
10
16
18
3 27 38 43
9 10 82
11
19
3 9 10 27 38 43 82
20

Merge Sort

1. Divide For dividing the List, we calculate the midpoint of the array or list by using the formula mid = len(list)//2

2. Conquer In this step, we divide the list into sub-arrays with the help of the midpoint calculated.

This division into sub-list and calculation of midpoint is done recursively for the same.

- a single element in an List is always sorted. Therefore, we have to continuously divide the sub-list until all elements in the list are single element in a List.

Merge Sort

3. Combine Since, now we are done with the formation of sub-Lists, the last step is to combine them back in sorted order.

Sorting List

The merge sort algorithm divides the given List into roughly two equal halves and sorts them recursively. This process is continued until arrays have only one element left.

a). It is to create copies of List. The first List contains the elements from [start_index,middle]and the **second** from[middle+1,end_index].

b).we traverse both copies of the List with the help of pointers and select the smaller value of the two values and add them to the sorted List.

c).if we run out of elements in any List, we select the remaining elements and place them in the sorted List.

Merge Sort

```
def mergesort(Arr,L,R):
# Sort the slice A[left:right]
        if len(arr)<= 1: # Base case
                return(Arr)
        # Recursive call
        mid = (L+R)//2
          L = arr[:mid]           #call first left list
          R = arr[mid:]           #call second right list
        L = mergesort(L)
        R = mergesort(R)
        return(merge(L,R))
```

Merge Sort

```
def merge(L,R): # Merge A[0:m],B[0:n]
new = []            #create a new list for merge of two list
(i,j) = (0,0) # Current positions in L,R
m, n = len(L), len(R)
while i < m and j < n:# i,j is number of elements merged so far
        if L[i] < R[j]:
                new.append(L[i])
                i += 1
        else:
                new.append(R[j])
                j += 1
new.extend(L[i:]) # add at end remaining element from L
new.extend(R[j:]) # add at end remaining element from R
return new
```

Approach2:Merging two sorted lists

- **For example:**
- **~~32~~ ~~74~~ ~~89~~**
- **~~21~~ ~~55~~ ~~64~~**

21 32 55 64 74 89

Combining sorted lists

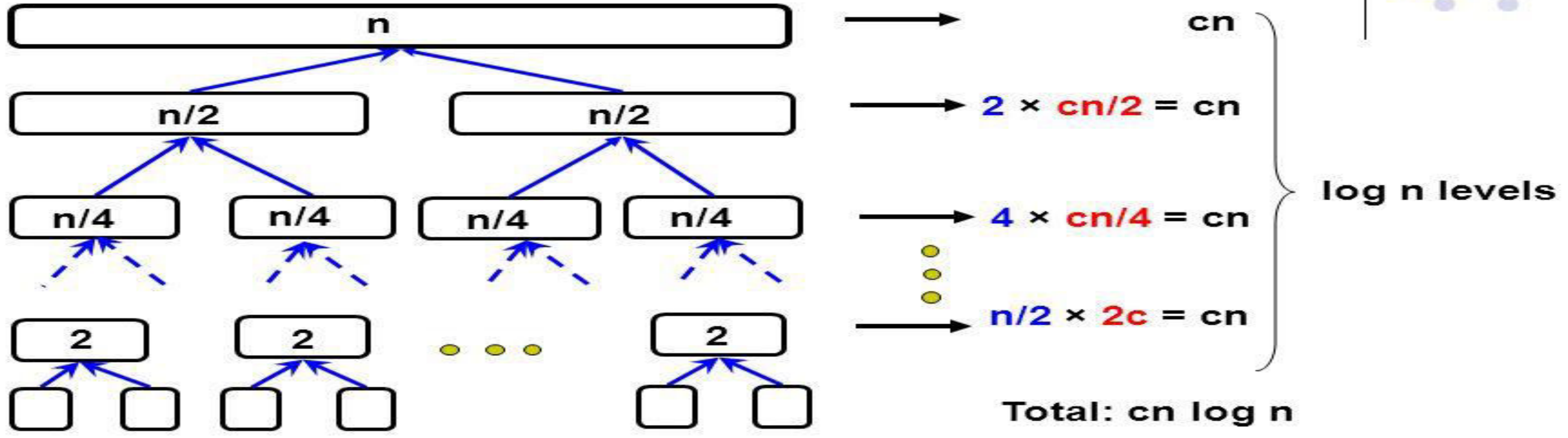
- Given two sorted lists A and B, combine into a
- sorted list C
- Compare first element of A and B
- Move it into C
- Repeat until all elements in A and B are over
- Merging A and B

Merging sorted lists

- Combine two sorted lists A and B into C
- If A is empty, copy B into C
- If B is empty, copy A into C
- Otherwise, compare first element of A and B and
- move the smaller of the two into C
- Repeat until all elements in A and B have been
- moved

Time complexity

Merge-Sort Analysis



- Total running time: $\Theta(n\log n)$
- Total Space: $\Theta(n)$

Example :10, 6, 12, 9, 8, 2, 35,9,11 using Merge Sort

Any Questions???