Data Structures Lab 04 (A)

Course: Data Structures (CL2001) **Semester**: Fall 2024
Instructor: Muhammad Nouman Hanif

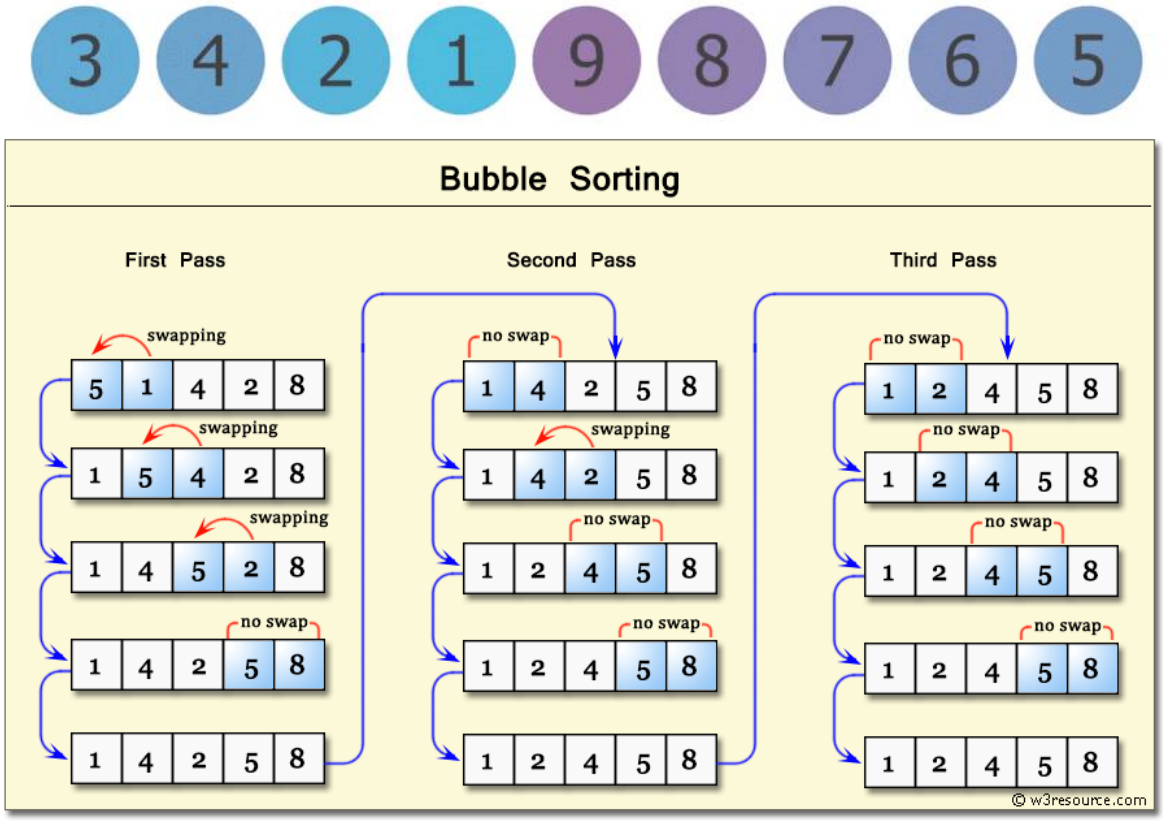
Note:

- Lab manual cover following below elementary sorting algorithms
  {Bubble Sort, Selection Sort, Insertion Sort, Shell Sort,Comb Sort}
- Maintain discipline during the lab.
- Listen and follow the instructions as they are given.
- Just raise hand if you have any problem.
- Completing all tasks of each lab is compulsory.
- Get your lab checked at the end of the session.

Sorting Algorithms

Bubble Sort

Bubble Sort is the simplest sorting algorithm that works by **repeatedly swapping** the adjacent elements if they are in the wrong order. The larger values sink to the bottom and hence called **sinking sort**. At the end of each pass, smaller values gradually "bubble" their way upward to the top and hence called **bubble sort**.



Pseudocode:

```
BubbleSort(arr[], n)
{
    for (i = 0 to n-1)
    {
        swapped = false
        for (j = 0 to n-i-1)
        {
                //when the current item is bigger than next
            if (arr[j] > arr[j+1])
            {
                temp = arr[j]
                arr[j] = arr[j+1]
                arr[j+1] = temp
                swapped = true
            }
        }
        if (swapped == false)
            break
    }
}
```

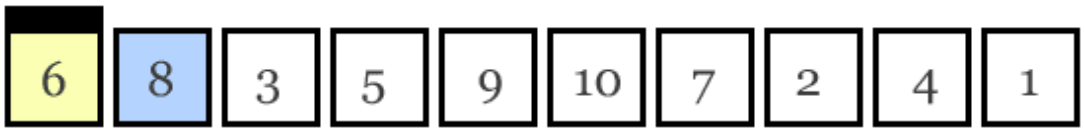
➔ Go through the array, **one value at a time**.
➔ For each value, **compare the value** with the next value.
➔ If the value is higher than the next one, swap the values so that the **highest value comes last.**
➔ Go through the array **as many times as there are values** in the array.

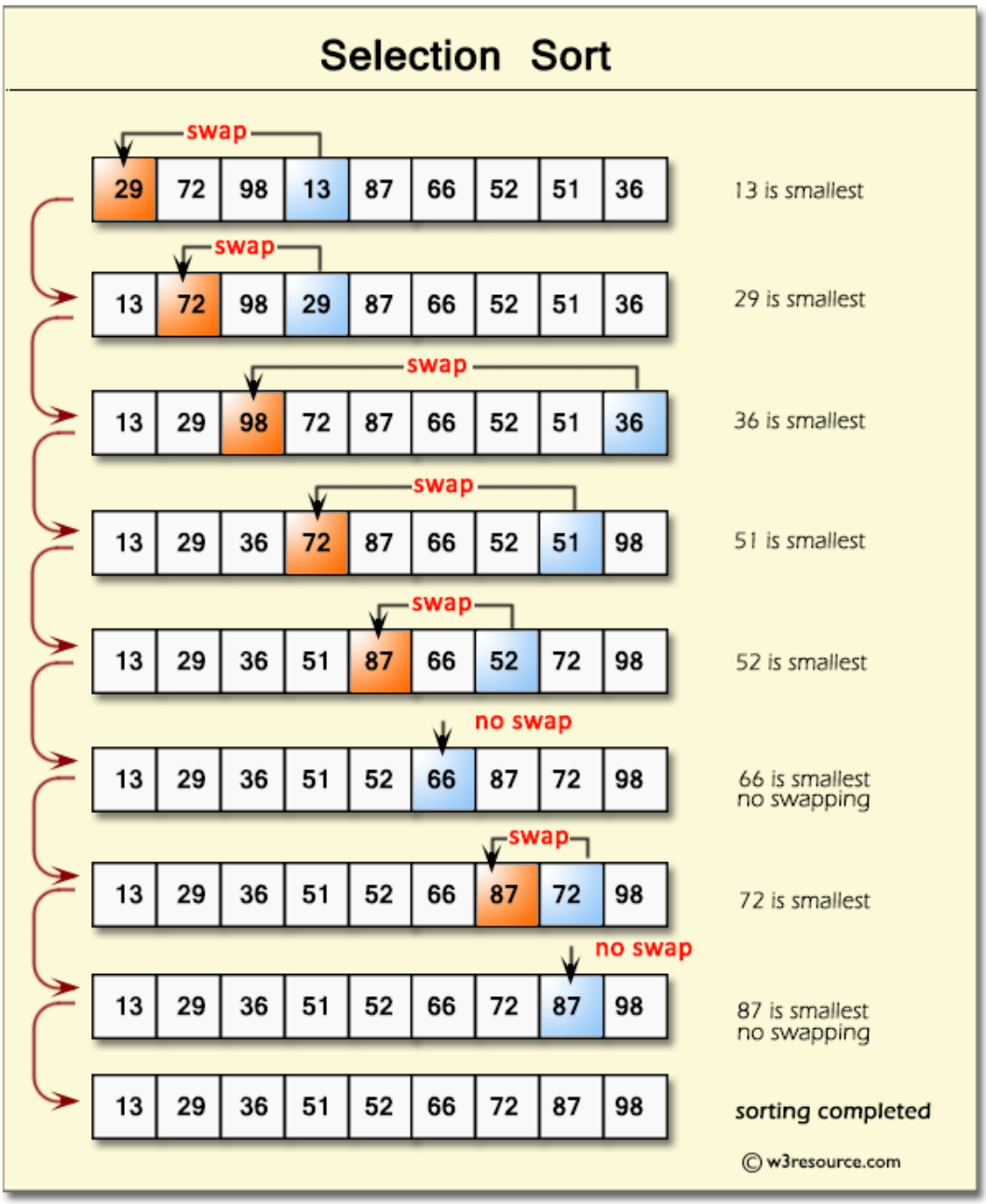
Complexity Analysis of Bubble Sort:

➔ **Time Complexity:**
- Worst Case - $O(N^2)$
- Average Case - $O(N^2)$
- Best Case - $O(N)$

Selection Sort

Selection sort is a simple and efficient sorting algorithm that works by **repeatedly selecting the smallest (or largest) element** from the **unsorted portion** of the list and moving it to the sorted portion of the list. The algorithm repeatedly selects the smallest (or largest) element from the unsorted portion of the list and swaps it with the first element of the unsorted part. This process is repeated for the remaining unsorted portion until the entire list is sorted.





Pseudocode:

```
SelectionSort(arr[], n)
{
    for (i = 0 to n-1)
    {
        minIndex = i
           // Find the smallest element in the unsorted part of the array
        for (j = i+1 to n)
        {
            if (arr[j] < arr[minIndex])
            {
                // Update minIndex if a smaller element is found
                minIndex = j
            }
        }
          // Swap the found minimum element with the first element   of the
          unsorted part
        if (minIndex != i)
        {
            temp = arr[i]
            arr[i] = arr[minIndex]
            arr[minIndex] = temp
        }
    }
}
```

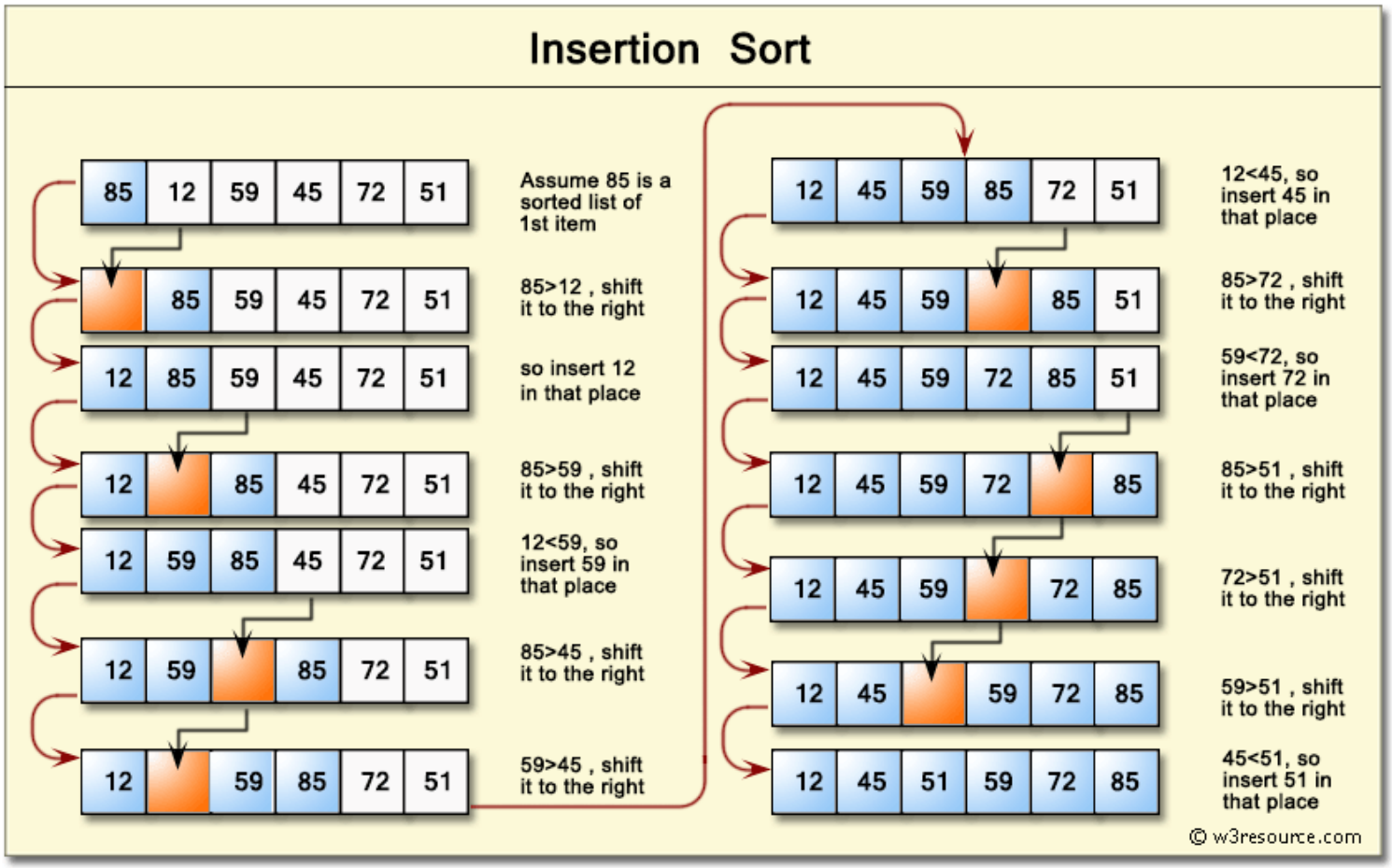
➔ Go through the array to **find the lowest value**.
➔ **Move the lowest value** to the front of the unsorted part of the array.
➔ Go through the array again **as many times as there are values** in the array.

Complexity Analysis of Selection Sort:

➔ **Time Complexity:**
- Worst Case - $O(N^2)$
- Average Case - $O(N^2)$
- Best Case - $O(N^2)$

Insertion Sort

Insertion Sort is a sorting algorithm that gradually builds a **sorted sequence by repeatedly inserting unsorted elements into their appropriate positions.** In each iteration, an unsorted element is taken and placed within the sorted portion of the array. This process continues until the entire array is sorted. It is a stable sorting algorithm, meaning that elements with equal values maintain their relative order in the sorted output. Insertion sort is like sorting playing cards in your hands. You split the cards into two groups: the sorted cards and the unsorted cards. Then, you pick a card from the unsorted group and put it in the right place in the sorted group.



Pseudocode:

```
InsertionSort(arr[], n)
{
    for (i = 1 to n)
    {
        key = arr[i]
        j = i – 1
           // Move elements of the sorted part to make space for the current
           element
        while (j >= 0 and arr[j] > key)
        {
            arr[j+1] = arr[j]
            j = j - 1
        }
           // Insert the current element into its correct position
        arr[j+1] = key
    }
}
```

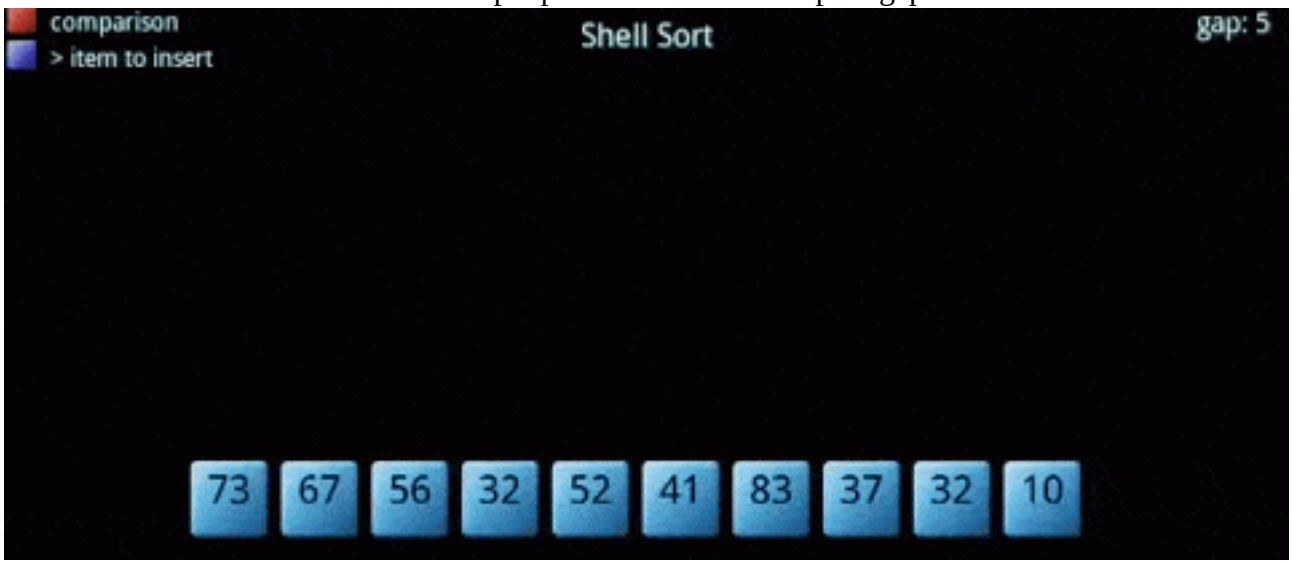
➔ Take the first value from the **unsorted part of the array**.
➔ **Move the value into the correct place** in the sorted part of the array.
➔ Go through the unsorted part of the array **again as many times as there are values**.

Complexity Analysis of Insertion Sort:

➔ **Time Complexity:**
- Worst Case - $O(N^2)$
- Average Case - $O(N^2)$
- Best Case - $O(N)$

Shell Sort

Shell sort is a variation of Insertion Sort. In **Insertion Sort, elements are moved one position at a time**, which can be slow for distant elements. Shell sort speeds this up by **first sorting elements that are far apart**, then gradually **reducing the gap between elements**. The process **continues until the gap is 1**, at which point the array is fully sorted. Shell sort is an improvement over insertion sort. It compares the element separated by a gap of several positions. A data element is sorted with multiple passes and with each pass gap value reduces.



First Pass (gap = 3)

10	1	23	50	4	9	-4
10	1	23	50	4	9	-4
10	1	23	50	4	9	-4
10	1	23	50	4	23	-4
10	1	9	50	4	23	-4
10	1	9	50	4	23	-4
10	1	9	50	4	23	50
10	1	9	10	4	23	50
-4	1	9	10	4	23	50

9 temp

-4 temp

Second Pass (gap = 1)

-4	1	9	10	4	23	50
-4	1	9	10	4	23	50
-4	1	9	10	4	23	50
-4	1	9	10	4	23	50
-4	1	9	10	10	23	50
-4	1	9	9	10	23	50
-4	1	4	9	10	23	50
-4	1	4	9	10	23	50
-4	1	4	9	10	23	50

4 temp

Pseudocode:

```
ShellSort(arr[], n)
{
    // Start with a large gap, then reduce the gap
    for (gap = n / 2; gap > 0; gap = gap / 2)
    {
        // Perform a gapped insertion sort for this gap size
        for (i = gap to n)
        {
            key = arr[i]
            // Shift elements that are gap positions apart
            j = i
            while (j >= gap and arr[j - gap] > key)
            {
                arr[j] = arr[j - gap]
                j = j - gap
            }
            // Put temp in its correct position
            arr[j] = key
        }
    }
}
```

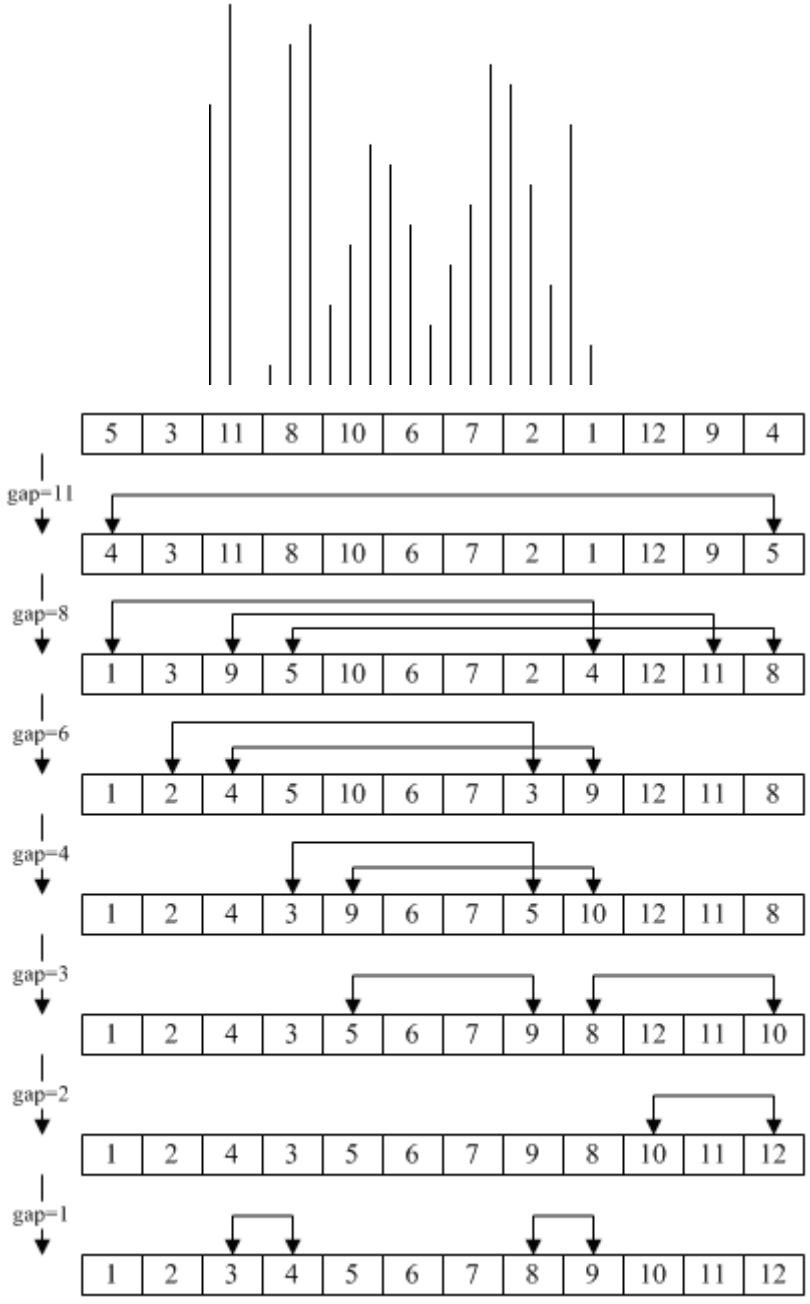
➔ Initialize the **value of gap**.
➔ Divide the list into **smaller sub-list of equal intervals for gap**.
➔ Sort these **sub-lists using insertion sort**.
➔ Repeat until complete list is sorted.

Complexity Analysis of Shell Sort:

➔ **Time Complexity:**
- Worst Case - $O(N^2)$
- Average Case - $O(N \log^2 N)$
- Best Case - $O(N \log N)$

Comb Sort

Comb Sort is an efficient sorting algorithm designed to **improve upon Bubble Sort by reducing the number of comparisons and swaps required**. It works by **initially sorting elements that are far apart and gradually reducing the gap between elements being compared**. The core idea of Comb Sort is to use a "gap" that decreases over time, which allows the algorithm to move elements into their correct positions more quickly compared to traditional sorting methods. As the gap decreases, the algorithm performs a final pass with a gap of 1, similar to Bubble Sort, to ensure the entire array is sorted. This method helps in reducing the time complexity compared to simple Bubble Sort, especially for larger arrays.



Pseudocode:

```
CombSort(arr[], n)
{
    // Initialize the gap size
    gap = n
    // Initialize the shrink factor (usually 1.3)
    shrink = 1.3
    swapped = true
    // Continue sorting while gap > 1 or a swap occurred in the last pass
    while (gap > 1 or swapped == true)
    {
        gap = int(gap / shrink)
        // Ensure gap is at least 1
        if (gap < 1)
            gap = 1
        swapped = false
        // Perform the comparison and swapping within the current gap
        for (i = 0 to n - gap - 1)
        {
            if (arr[i] > arr[i + gap])
            {
                temp = arr[i]
                arr[i] = arr[i + gap]
                arr[i + gap] = temp
                swapped = true
            } } } }
```

➔ Calculate the **gap value if gap value == 1 then print the sorted array** else iterate.
➔ **Iterate over data set and compare each item** with gap item.
➔ **Swap the element if require** otherwise continue the iteration.
➔ Repeat until complete list is sorted.

Complexity Analysis of Comb Sort:

➔ **Time Complexity:**

- Worst Case - $O(N^2)$
- Average Case - $O(N^2 / 2^{1.3})$
- Best Case - O(N Log N)

Lab Exercises

1. Implement the bubble sort algorithm to sort the in descending order (starting from the initial pass). Take array [10] = {5,1,3,6,2,9,7,4,8,10}. You can also take your array as user input.

2. You are tasked with implementing the Shell Sort algorithm to sort the weights of employees in a company. However, instead of using the traditional gap sequence (where the gap is divided by 2), you must create and implement a custom gap sequence of your choice that you think can align with the problem.

3. You are asked to sort a list of product prices for a retail store using Comb Sort. However, instead of using the standard shrink factor of 1.3 (as typically used in Comb Sort), you must create and implement a custom shrink factor of your choice that you think can align with the problem.

4. In a bustling corporate office, the facilities management team is tasked with organizing the seating arrangements for employees based on their designations. The office layout consists of rows of computer desks, and each desk has a designated employee. The priority is to sort out the computer desks for employees using the Insertion Sort algorithm, with the designation determining the sorting order. The higher the designation, the closer the employee should be seated to the corner office. The designations and their corresponding priorities are as follows:

   i. CEO (Chief Executive Officer) - Highest Priority
   ii. CTO (Chief Technology Officer)
   iii. CFO (Chief Financial Officer)
   iv. VP (Vice President)
   v. MGR (Manager)
   vi. EMP (Employee) - Lowest Priority

Here's the initial arrangement of employees' desks from left to right:

   i. Employee (EMP)
   ii. CFO (Chief Financial Officer)
   iii. Manager (MGR)
   iv. Employee (EMP)
   v. VP (Vice President)
   vi. CTO (Chief Technology Officer)
   vii. Manager (MGR)
   viii. CEO (Chief Executive Officer)

5. Develop C++ solution such that day month and year are taken as input for 5 records and perform Sorting Dates based on year using Selection Sort. Note: Input must be taken from user.
[Hint: Struct or Class can be used]
It`s not strictly necessary to take inputs in the format as shown in example, but, the output should be in the given format.

Example Input:	**Example Output:**
01/02/2022	4/07/2015
5/01/2018	5/01/2018
4/07/2015	12/10/2021
12/10/2021	01/02/2022
11/12/2023	11/12/2023

6. A clerk at a shipping company is charged with the task of rearranging a number of large crates in order of the time they are to be shipped out. Thus, the cost of compares is very low relative to the cost of exchanges (move the crates). The warehouse is nearly full: there is extra space sufficient to hold any one of the crates, but not two. Which sorting method should the clerk use? Implement this question via a user generated array?