Data Structures Lab 04(a)

Course: Data Structures (CL2001) **Semester**: Fall 2024
Instructor: Sameer Faisal **T.A**: N/A

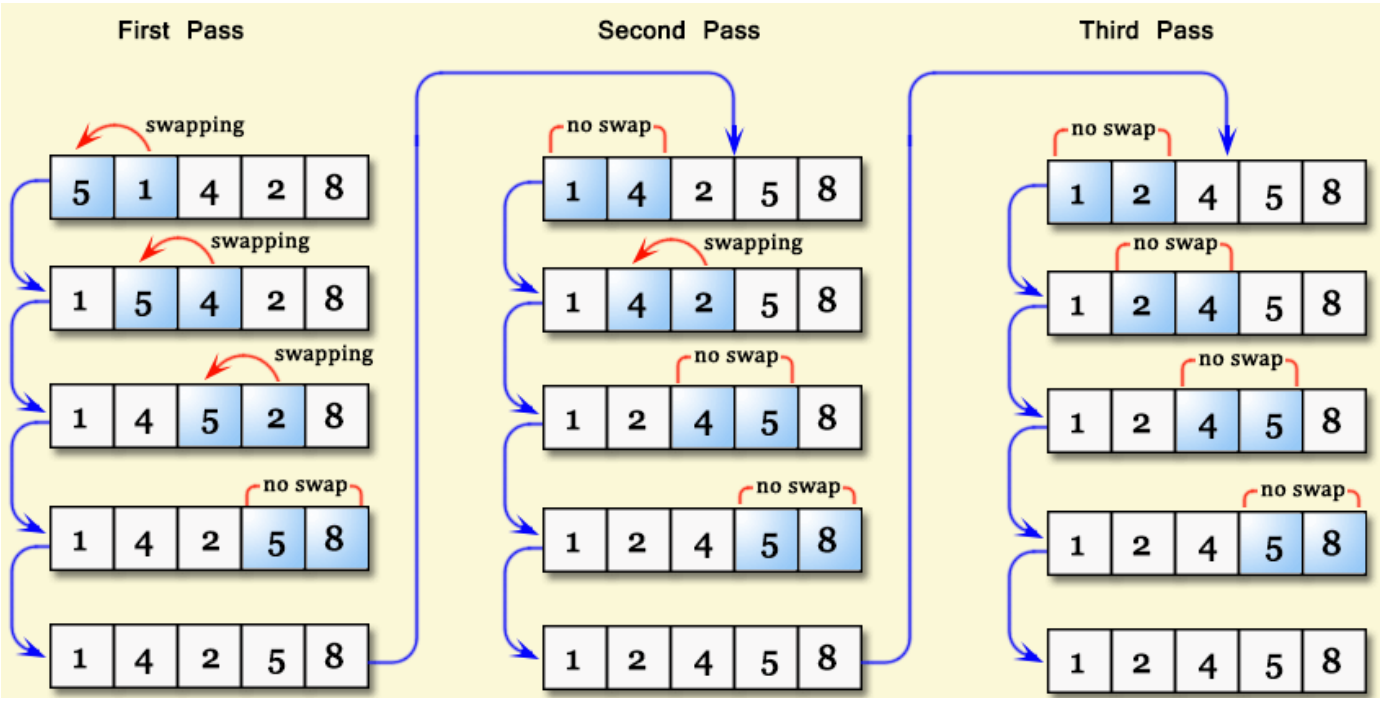
Note:

- 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

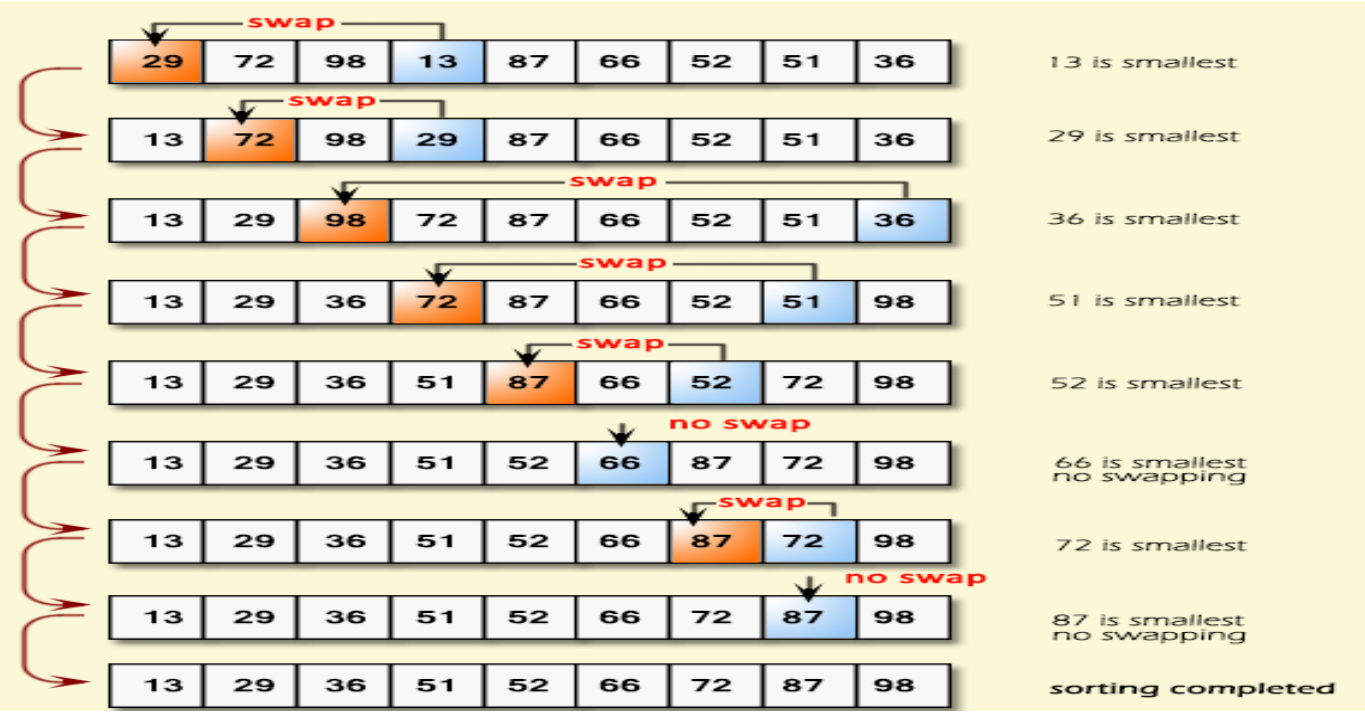
Algorithm : Bubble sort

```
Data: Input array A[]
Result: Sorted A[]
int i, j, k;
N = length(A);
for j = 1 to N do
    for i = 0 to N-1 do
        if A[i] > A[i+1] then
            temp = A[i];
            A[i] = A[i+1];
            A[i+1] = temp;
        end
    end
end
```

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

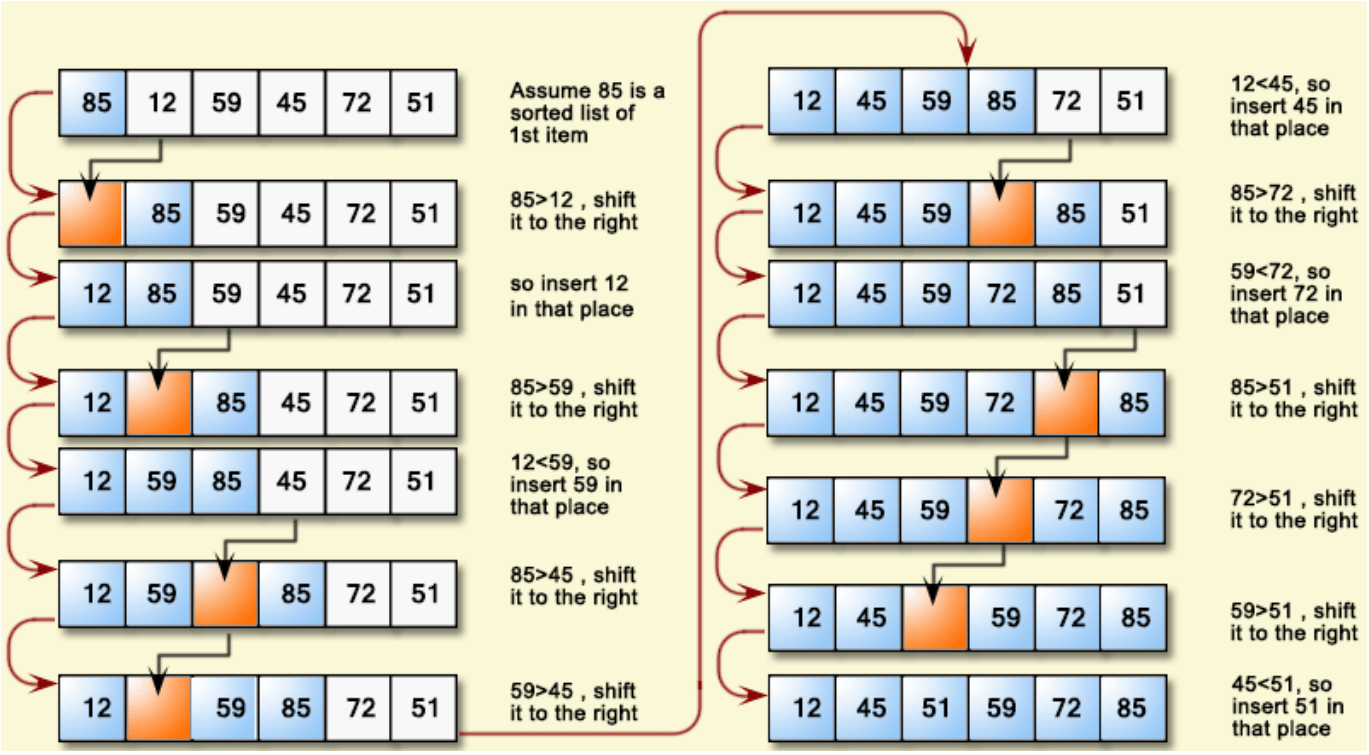
Algorithm 4 Selection Sort

```
1: for i = 1 to n − 1 do
2:     min = i
3:     for j = i + 1 to n do
4:         // Find the index of the i^th smallest element
5:         if A[j] < A[min] then
6:             min = j
7:         end if
8:     end for
9:     Swap A[min] and A[i]
10: end for
```

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.



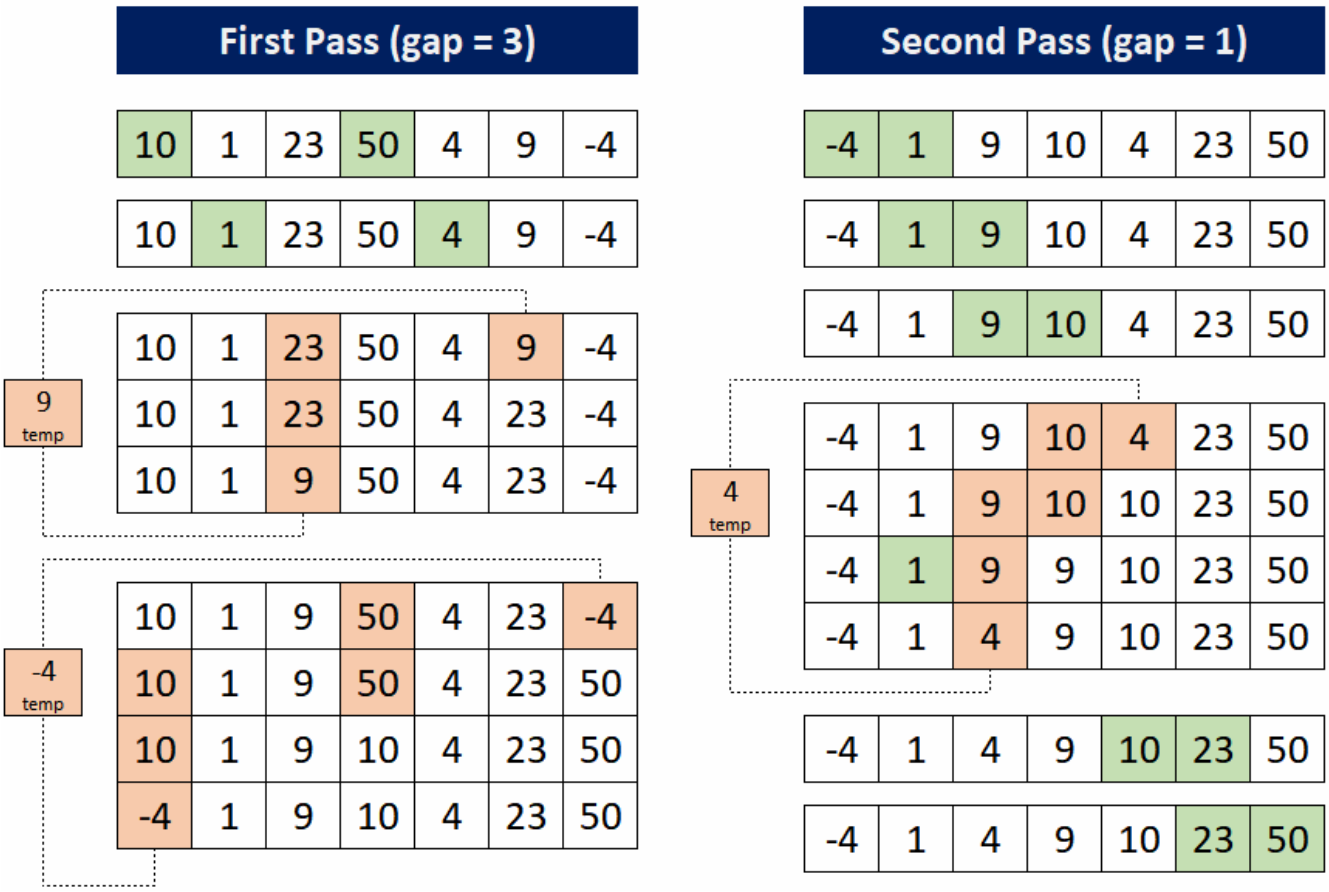
Pseudocode

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

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.



Code

```
void shellSort(int myarr[], int size) {
    // Start with a big gap, then reduce the gap
    for (int gap = size / 2; gap > 0; gap /= 2) {
        // Perform a gapped insertion sort for this gap size
        for (int j = gap; j < size; j++) {
            int temp = myarr[j];
            int res = j;
            // Shift earlier gap-sorted elements up until the correct location for myarr[j] is found
            while (res >= gap && myarr[res - gap] > temp) {
                myarr[res] = myarr[res - gap];
                res -= gap;
            }
            // Put temp (the original myarr[j]) into its correct location
            myarr[res] = temp;
        }
    }
}
```

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.

Code

```
void combSort(int arr[], int n) {
    float shrink = 1.3; // Shrink factor
    int gap = n; // Initialize gap to the size of the array
    bool swapped = true;

    while (gap > 1 || swapped) {
        // Update the gap using the shrink factor
        gap = (int)(gap / shrink);
        if (gap < 1) {
            gap = 1; // Ensure the gap is at least 1
        }

        swapped = false;

        // Perform a gapped bubble sort
        for (int i = 0; i + gap < n; ++i) {
            if (arr[i] > arr[i + gap]) {
                // Swap arr[i] and arr[i + gap]
                int temp = arr[i];
                arr[i] = arr[i + gap];
                arr[i + gap] = temp;
                swapped = true;
            }
        }
    }
}
```

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. 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

3. 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)

4. 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.

5. 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.