CL2001 **Data Structures Lab**	**Lab 04** **Linear, Binary & Interpolation Search, Elementary Sorting Techniques (Bubble, Selection, Insertion, Comb)**

NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES

Fall 2025

Lab Content

1. Bubble
2. Insertion
3. Selection
4. Shell sort
5. Comb sort
6. Binary search
7. Interpolation search
8. Linear Search

BUBBLE SORT:

Bubble Sort, the two successive strings arr[i] and arr[i+1] are exchanged whenever arr[i]> arr[i+1]. 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.

Example:

```
//you need to take input from user and display the unsorted array.
//sort the array using the following steps .

for (int i = 0; i < n; i++) {
   for (int j = 0; j < n - 1; j++) {
      if (a[j] > a[j + 1]) {
         // Swap elements if they are in the wrong order
         int temp = a[j];
         a[j] = a[j + 1];
         a[j + 1] = temp;
      }
   }
}
//display your sorted array.
```

SELECTION SORT:

Key Points:

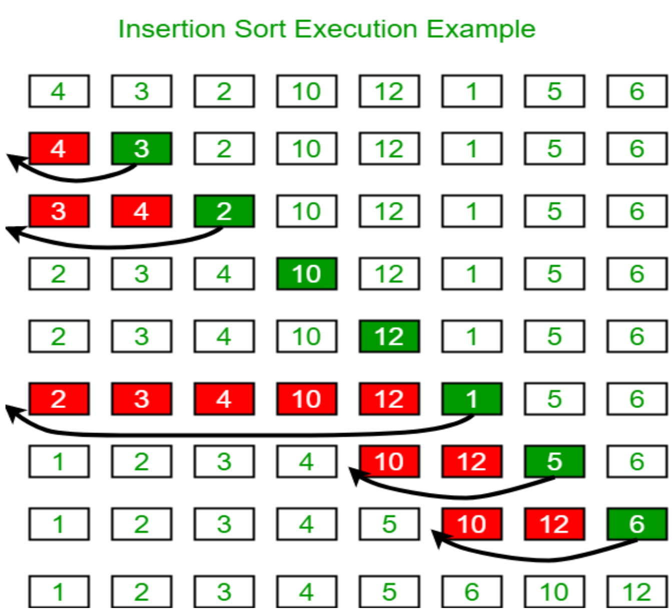
void selectionSort(int *array, int size) {
Find the smallest element in the array and exchange it with the element in the first position.
Find the second smallest element in the array and exchange it with the element in the second position.
Continue this process until done.
}

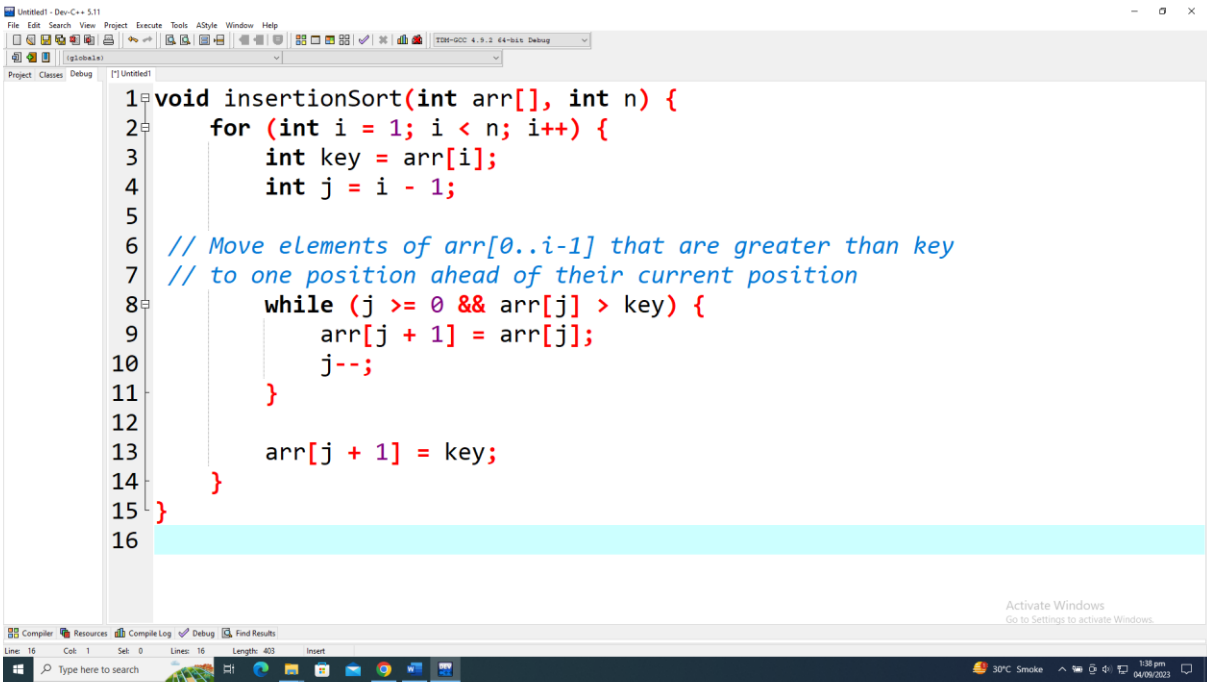
- Example: (5,10,3,5,4)

```cpp
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int min_index = i;

        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_index]) {
                min_index = j;
            }
        }

        int temp = arr[i];
        arr[i] = arr[min_index];
        arr[min_index] = temp;
    }
}
```

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.





5,10,3,2

<u>SHELL SORT :</u>

```
shellSort(array, size)
 for interval i <- size/2n down to 1
  for each interval "i" in array
      sort all the elements at interval "i"
end shellSort
```

Problem Description

1. Shell sort is an improvement over insertion sort.
2. It compares the element separated by a gap of several positions.
3. A data element is sorted with multiple passes and with each pass gap value reduces.

Problem Solution

1. Assign gap value as half the length of the array.
2. Compare element present at a difference of gap value.
3. Sort them and reduce the gap value to half and repeat.
4. Display the result.
5. Exit.

```cpp
#include <iostream>

void shellSort(int myarr[], int n1) {
    // Start with a big gap, then reduce the gap
    for (int gap = n1 / 2; gap > 0; gap /= 2) {
        // Perform a gapped insertion sort for this gap size
        for (int j = gap; j < n1; 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;
        }
    }
}

int main() {
    int myarr[] = { 12, 34, 54, 2, 3 };
    int n1 = sizeof(myarr) / sizeof(myarr[0]);

    shellSort(myarr, n1);

    std::cout << "Sorted array: ";
    for (int i = 0; i < n1; i++) {
        std::cout << myarr[i] << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Key Points:

1. Take input of data.
2. Create and call function name as ShellSort() contains two arguments.('arr' the array of data and 'n' the number of values).
3. Implement Sorting algorithm using nested for loop.
4. The first loop will run on 'gap' Which decides the gap value to compare two elements.
5. The second loop will run on 'j' from j to n.
6. The third loop will run on 'res' & sort the element having "gap" as a gap between their index.
7. Switch the values if arr[res] < arr[res-gap].
8. Return to main and display the result.

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.

```cpp
#include <iostream>

// Function to perform Comb Sort
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;
            }
        }
    }
}

```

SEARCHING ALGORITHMS:

LINEAR SEARCH ALGORITHM: Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.

```
int i;
for (i = 0; i < N; i++)
if (arr[i] == x)
return i;
}
```

BINARY SEARCH ALGORITHM:

Binary Search is a searching algorithm for finding an element's position in a sorted array. In this approach, the element is always searched in the middle of a portion of an array. Binary search can be implemented only on a sorted list of items. If the elements are not sorted already, we need to sort them first.

```
while (left <= right) {
   int mid = left + (right - left) / 2;
   if (arr[mid] == key) {
      return mid;
   }
   else if (arr[mid] < key) {
      left = mid + 1;
   }
   else {
      right = mid - 1;
   }
}
return -1;
```

INTERPOLATION SEARCH:

The Interpolation Search is an improvement over Binary Search for instances, where the values in a sorted array are uniformly distributed. Interpolation constructs new data points within the range of a discrete set of known data points.

```
#include <iostream>

int interpolationSearch(int arr[], int size, int x) {
    int low = 0;
    int high = size - 1;

    while (low <= high && x >= arr[low] && x <= arr[high]) {
        if (low == high) {
            if (arr[low] == x) return low;
            return -1;
        }

        // Estimate the position
        int pos = low + ((x - arr[low]) * (high - low)) / (arr[high] - arr[low]);

        // Check if the estimated position has the target value
        if (arr[pos] == x) return pos;

        // If the target value is greater, ignore the left half
        if (arr[pos] < x) low = pos + 1;
        // If the target value is smaller, ignore the right half
        else high = pos - 1;
    }
    return -1;
}

int main() {
    int arr[] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};
    int size = sizeof(arr) / sizeof(arr[0]);

    int x;
    std::cout << "Enter the value to search: ";
    std::cin >> x;

    int index = interpolationSearch(arr, size, x);

```

COMPARATIVE TABLE OF SORTING AND SEARCHING ALGORITHMS

Algorithm	Type	Best Case	Worst Case	Average Case	Space Complexity	Key Characteristics
Bubble Sort	Sorting	O(n)	O(n^2)	O(n^2)	O(1)	Easy to implement, inefficient for large datasets. Values "bubble" to the correct position in each pass.
Selection Sort	Sorting	O(n^2)	O(n^2)	O(n^2)	O(1)	Simple but inefficient for large datasets. Swaps are reduced compared to Bubble Sort.
Insertion Sort	Sorting	O(n)	O(n^2)	O(n^2)	O(1)	Efficient for small or nearly sorted arrays. Performs better than Bubble or Selection Sort on small datasets.
Shell Sort	Sorting	O(nlogn)	O(n^2)	O(n log n)	O(1)	Gap reduces in multiple passes, improves efficiency over Insertion Sort.
Comb Sort	Sorting	O(nlogn)	O(n^2)	O(n^2)	O(1)	Reduces the number of comparisons compared to Bubble Sort. More efficient for larger arrays.
Linear Search	Searching	O(1)	O(n)	O(n)	O(1)	Simple, but inefficient for large datasets as it doesn't take advantage of any sorting.
Binary Search	Searching	O(1)	O(log n)	O(log n)	O(1)	Efficient on sorted arrays. It divides the search space by half at each step, significantly reducing the search time.
Interpolation Search	Searching	O(1)	O(n)	O(log log n)	O(1)	Efficient when elements are uniformly distributed, but can degrade to linear search when the distribution is not uniform.

Lab Tasks:

1. Implement a linear search algorithm to find a specific integer in an array. The program should take an array of integers and a target value as input from the user. Print the index of the target value if it's found or a message indicating that the element is not in the array.

2. Write a program to implement the Bubble Sort algorithm. The program should take an array of integers as input from the user, sort the array using bubble sort, and then display both the unsorted and sorted arrays.

3. Implement the Selection Sort algorithm. The program should take an array of integers from the user, find the smallest element, and exchange it with the element at the first position. This process should continue until the array is completely sorted. Display the sorted array to the user.

4. Use Insertion Sort to sort an array of integers provided by the user. Then, implement the Binary Search algorithm to find a target value within the newly sorted array. The program should display the sorted array and the index of the found element or a message if it's not present.

5. Implement the Interpolation Search algorithm. The program should take a sorted array with uniformly distributed values and a target value from the user. The algorithm should estimate the position of the target and check if the estimated position holds the correct value. If the element is found, print its index; otherwise, indicate that it's not present.

6. Implement the Comb Sort algorithm to sort an array of integers. This algorithm improves upon bubble sort by reducing the number of comparisons and swaps using a "gap" that decreases over time. After sorting, perform a complexity analysis: count and display the total number of swaps and comparisons performed during the sorting process.

7. Implement the Shell Sort algorithm to sort an array of integers. The algorithm should start with a large gap, then perform a gapped insertion sort, and gradually reduce the gap until it's 1. After sorting, compare its performance against the Bubble Sort and Insertion Sort algorithms for the same input array size (e.g., 1000 elements). Record and display the time taken and the number of comparisons/swaps for all three algorithms to demonstrate why Shell Sort is an improvement over Insertion Sort.