



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 | 5. comb sort |
| 2. Insertion | 6. binary search |
| 3. Selection | 7. interpolation search |
| 4. shell sort | 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)

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

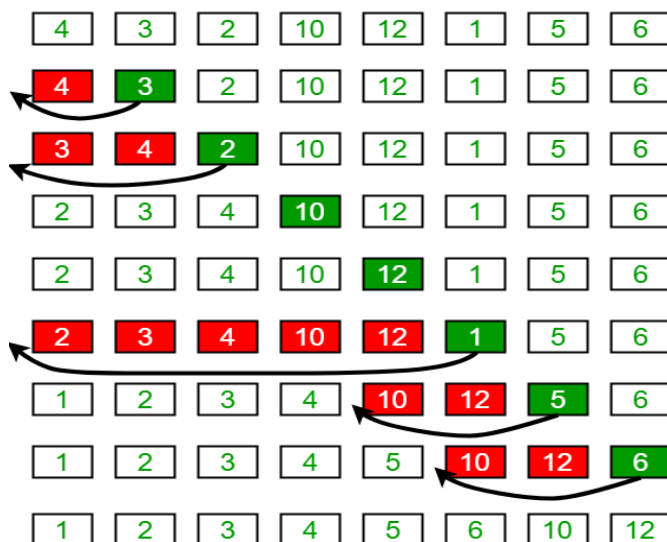
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_index]) {
                min_index = j;
            }
        }

        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.

Insertion Sort Execution Example





```
1 void insertionSort(int arr[], int n) {
2     for (int i = 1; i < n; i++) {
3         int key = arr[i];
4         int j = i - 1;
5
6         // Move elements of arr[0..i-1] that are greater than key
7         // to one position ahead of their current position
8         while (j >= 0 && arr[j] > key) {
9             arr[j + 1] = arr[j];
10            j--;
11        }
12        arr[j + 1] = key;
13    }
14 }
15 }
16
```

5,10,3,2

SHELL SORT :

```
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.

```
1 #include <iostream>
2
3 void shellSort(int myarr[], int n1) {
4     // Start with a big gap, then reduce the gap
5     for (int gap = n1 / 2; gap > 0; gap /= 2) {
6         // Perform a gapped insertion sort for this gap size
7         for (int j = gap; j < n1; j++) {
8             int temp = myarr[j];
9             int res = j;
10
11             // Shift earlier gap-sorted elements up until the correct location for myarr[j] is found
12             while (res >= gap && myarr[res - gap] > temp) {
13                 myarr[res] = myarr[res - gap];
14                 res -= gap;
15             }
16             // Put temp (the original myarr[j]) into its correct location
17             myarr[res] = temp;
18         }
19     }
20 }
21
22 int main() {
23     int myarr[] = { 12, 34, 54, 2, 3 };
24     int n1 = sizeof(myarr) / sizeof(myarr[0]);
25
26     shellSort(myarr, n1);
27
28     std::cout << "Sorted array: ";
29     for (int i = 0; i < n1; i++) {
30         std::cout << myarr[i] << " ";
31     }
32     std::cout << std::endl;
33
34     return 0;
35 }
```

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.

```
1  #include <iostream>
2
3  // Function to perform Comb Sort
4  void combSort(int arr[], int n) {
5      float shrink = 1.3; // Shrink factor
6      int gap = n; // Initialize gap to the size of the array
7      bool swapped = true;
8
9      while (gap > 1 || swapped) {
10         // Update the gap using the shrink factor
11         gap = (int)(gap / shrink);
12         if (gap < 1) {
13             gap = 1; // Ensure the gap is at least 1
14         }
15
16         swapped = false;
17
18         // Perform a gapped bubble sort
19         for (int i = 0; i + gap < n; ++i) {
20             if (arr[i] > arr[i + gap]) {
21                 // Swap arr[i] and arr[i + gap]
22                 int temp = arr[i];
23                 arr[i] = arr[i + gap];
24                 arr[i + gap] = temp;
25                 swapped = true;
26             }
27         }
28     }
29 }
30
```

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.

```
1  #include <iostream>  
2  
3  int interpolationSearch(int arr[], int size, int x) {  
4      int low = 0;  
5      int high = size - 1;  
6  
7      while (low <= high && x >= arr[low] && x <= arr[high]) {  
8          if (low == high) {  
9              if (arr[low] == x) return low;  
10             return -1;  
11         }  
12  
13         // Estimate the position  
14         int pos = low + ((x - arr[low]) * (high - low)) / (arr[high] - arr[low]);  
15  
16         // Check if the estimated position has the target value  
17         if (arr[pos] == x) return pos;  
18  
19         // If the target value is greater, ignore the left half  
20         if (arr[pos] < x) low = pos + 1;  
21         // If the target value is smaller, ignore the right half  
22         else high = pos - 1;  
23     }  
24     return -1;  
25 }  
26  
27 int main() {  
28     int arr[] = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100};  
29     int size = sizeof(arr) / sizeof(arr[0]);  
30  
31     int x;  
32     std::cout << "Enter the value to search: ";  
33     std::cin >> x;  
34  
35     int index = interpolationSearch(arr, size, x);  
36  
37 }
```



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	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	$O(1)$	Gap reduces in multiple passes, improves efficiency over Insertion Sort.
Comb Sort	Sorting	$O(n \log n)$	$O(n^2)$	$O(n^2)$	$O(1)$	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

Task # 01:

- Write a C++ program using **Bubble Sort** to sort an array of 10 integers entered by the user. Display the array before and after sorting.
- Modify Bubble Sort to stop early if the array becomes sorted before completing all passes. Show with an example input how it works.

Task # 02:

- Use **Selection Sort** to arrange names of 5 cities alphabetically. (Hint: compare strings instead of integers.)
- Count and display the **number of swaps** performed during Selection Sort on {29, 10, 14, 37, 13}.

Task # 03:

- Implement **Insertion Sort** to arrange a list of students' marks in **ascending order**. Print the sorted result.
- Modify Insertion Sort to arrange numbers in **descending order**. Test with: {15, 2, 78, 25, 63}

Task # 04:

- Implement **Shell Sort** to sort an array of **20 random numbers**. Display the gap sequence used.
- Compare **Shell Sort vs Insertion Sort** for the same input of {50, 40, 30, 20, 10}. Show which one sorts faster (by counting shifts/swaps).



Task # 05:

- Implement **Comb Sort** to sort an array of integers. Print the gap value at each pass.
- Compare the performance of **Comb Sort** with **Bubble Sort** for a reverse-sorted array {9, 8, 7, 6, 5, 4, 3, 2, 1}. Which one is faster and why?

Task # 06:

A university stores roll numbers of students in a sorted database. When an admin wants to check whether a student is registered, the system uses **Binary Search** instead of checking each roll number one by one.

Tasks:

1. Write a C++ program to search for a roll number in the sorted list:
{101, 105, 110, 120, 135, 150}
 - Input: a roll number from the user
 - Output: "Roll number found" or "Roll number not found"
2. Enhance the program to count and display how many comparisons were made while searching.

Example: Searching for 120 should show both the result and the number of steps taken.



Task # 07:

An e-commerce website stores product prices in sorted order. Since prices are **evenly distributed**, the system uses **Interpolation Search** to find prices faster than Binary Search.

Tasks:

1. Write a C++ program using **Interpolation Search** to find a product price from the sorted list:
{ 10, 20, 30, 40, 50, 60, 70, 80}
 - Input: a price entered by the user
 - Output: "Price found at position X" or "Price not available"
2. Compare **Binary Search** and **Interpolation Search** for the list:
{ 5, 10, 15, 20, 25, 30, 35, 40, 45, 50}
 - Search for 45
 - Show number of comparisons each method makes
 - Conclude which one is more efficient in this case.

Deliverables:

Submit the code file along with a Word file containing screenshots of the code and output for each task.