



**Batch: A1      Roll No.: 16010123012**

**Experiment / assignment / tutorial No. 11**

**Grade: AA / AB / BB / BC / CC / CD / DD**

**Signature of the Staff In-charge with date**

**Title:** Implementation of sorting Algorithms.

**Objective:** To Understand and Implement Bubble & Shell Sort

**Expected Outcome of Experiment:**

CO	Outcome
4	Demonstrate sorting and searching methods.

**Books/ Journals/ Websites referred:**

1. *Fundamentals Of Data Structures In C* – Ellis Horowitz, Satraj Sahni, Susan Anderson-Fred
2. *An Introduction to data structures with applications* – Jean Paul Tremblay, Paul G. Sorenson
3. *Data Structures A Pseudo Approach with C* – Richard F. Gilberg & Behrouz A. Forouzan

**Abstract:** (Define sorting process, state applications of sorting)

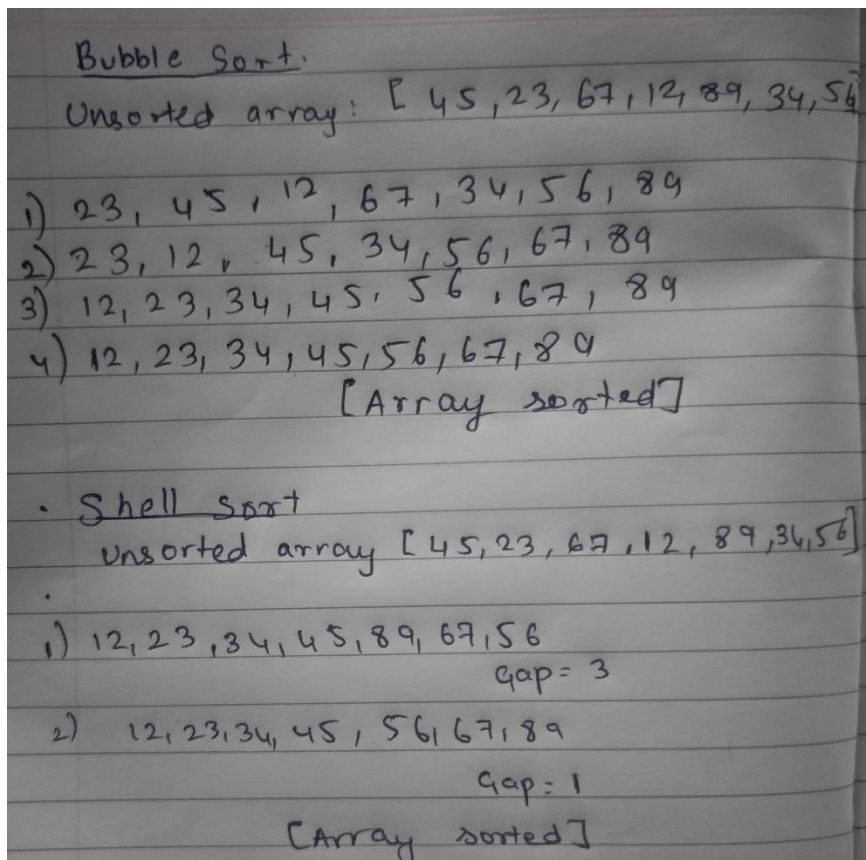
Sorting refers to ordering data in an increasing or decreasing fashion according to some linear relationship among the data items.

Sorting can be done on names, numbers and records. Sorting reduces the For example, it is relatively easy to look up the phone number of a friend from a telephone dictionary because the names in the phone book have been sorted into alphabetical order.

This example clearly illustrates one of the main reasons that sorting large quantities of information is desirable. That is, sorting greatly improves the efficiency of searching. If we were to open a phone book, and find that the names were not presented in any logical order, it would take an incredibly long time to look up someone's phone number.

**Example:**

*Take any random unsorted sequence of numbers and solve by using the Bubble and Shell Sort. Clearly showcase the sorted array after every pass.*



Bubble Sort.  
Unsorted array: [ 45, 23, 67, 12, 89, 34, 56 ]

- 1) 23, 45, 12, 67, 34, 56, 89
- 2) 23, 12, 45, 34, 56, 67, 89
- 3) 12, 23, 34, 45, 56, 67, 89
- 4) 12, 23, 34, 45, 56, 67, 89  
[Array sorted]

• Shell Sort  
Unsorted array [ 45, 23, 67, 12, 89, 34, 56 ]

- 1) 12, 23, 34, 45, 89, 67, 56  
gap = 3
- 2) 12, 23, 34, 45, 56, 67, 89  
gap = 1  
[Array sorted]

## Algorithm for Implementation:

### Bubble Sort:

1. The algorithm takes an array of  $n$  integers as input.
2. Two nested loops are used to iterate over the array:  
The outer loop (controlled by  $i$ ) runs from 0 to  $n-1$ , representing each pass through the array.  
The inner loop (controlled by  $j$ ) runs from 0 to  $n-i-2$ , comparing adjacent elements.
3. For each element at index  $j$ , the algorithm compares it with the element at index  $j + 1$ .
4. If the element at index  $j$  is greater than the element at index  $j + 1$ , the two elements are swapped. This operation ensures that, after the first complete iteration of the outer loop, the largest element has "bubbled" to the end of the array.
5. The swapped flag is used to optimize the algorithm. If no swaps occur during a pass (indicating that the array is already sorted), the algorithm breaks out of the loop early, reducing unnecessary iterations.
6. The process repeats with the next iterations of the outer loop, where the inner loop decreases its range by  $i$  each time (since the last  $i$  elements are already sorted).

### Shell Sort:

1. The algorithm takes an array of  $n$  integers as input.
2. The array is initially divided into subarrays based on a gap  $g$ . The gap starts at half the size of the array ( $g = n / 2$ ) and decreases with each iteration.
3. For each subarray defined by the gap, the algorithm performs a gapped insertion sort. It compares and swaps elements that are  $g$  positions apart in the array.
4. The algorithm compares elements at indices  $j$  and  $j - \text{gap}$ . If the element at index  $j - \text{gap}$  is greater than the element at index  $j$ , they are swapped to maintain order.
5. After each pass through the array, the gap is reduced by halving it ( $g = g / 2$ ). This process continues until the gap becomes 1, at which point the algorithm behaves like a standard insertion sort.
6. When the gap becomes 0, the sorting process is complete, and the array is fully sorted.

### Program:

```
#include <stdbool.h>
#include <stdio.h>

void swap(int* s, int* b) {
    int temp = *s;
    *s = *b;
```

```
*b = temp;
}

void bubbleSort(int arr[], int n) {
    int i, j;
    bool swapped;

    for (i = 0; i < n - 1; i++) {
        swapped = false;
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(&arr[j], &arr[j + 1]);
                swapped = true;
            }
        }
        if (!swapped)
            break;
    }
}

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int n;
    printf("Enter number of elements: ");
    scanf("%d", &n);
    int arr[n];
    for (int i = 0; i < n; i++) {
        printf("Enter %d integer: ", i + 1);
        scanf("%d", &arr[i]);
    }
    bubbleSort(arr, n);

    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}
```

**Output screenshots:**

```
Enter number of elements: 6
Enter 1 integer: 241
Enter 2 integer: 3246
Enter 3 integer: 325
Enter 4 integer: 45
Enter 5 integer: 867
Enter 6 integer: 12
Sorted array: 12 45 241 325 867 3246

=== Code Execution Successful ===
```

**SHELL SORT**

```
#include <stdio.h>

void shellSort(int arr[], int n) {
    for (int gap = n / 2; gap > 0; gap /= 2) {
        for (int i = gap; i < n; i++) {
            int temp = arr[i];
            int j;
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {
                arr[j] = arr[j - gap];
            }
            arr[j] = temp;
        }
    }
}

void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
```

```
int arr[n];
for (int i = 0; i < n; i++) {
    printf("Enter integer %d: ", i + 1);
    scanf("%d", &arr[i]);
}
shellSort(arr, n);

printf("Sorted array: ");
printArray(arr, n);
return 0;
}
```

```
Enter the number of elements: 6
Enter 1 integer: 34
Enter 2 integer: 466
Enter 3 integer: 78
Enter 4 integer: 23
Enter 5 integer: 53
Enter 6 integer: 14
Sorted array: 14 23 34 46 53 78

=== Code Execution Successful ===
```

**Conclusion:-**

In this experiment, we examined two essential sorting algorithms: Bubble Sort and Shell Sort. Both algorithms utilize distinct methods for sorting arrays, each with unique performance traits and complexities.

**Post Lab Questions:**

- 1) Describe how shell sort improves upon bubble sort. What are the main differences in their approaches?

Shell Sort improves upon Bubble Sort by using a more efficient approach of comparing and swapping elements that are far apart, reducing the gap with each pass. This allows Shell Sort to move elements closer to their final positions faster, making it more efficient overall. Main differences are:

1. Comparison Strategy:  
Bubble Sort: Compares adjacent elements.  
Shell Sort: Compares elements separated by a gap that decreases over time.
2. Efficiency:  
Bubble Sort:  $O(n^2)$  time complexity, inefficient for large arrays.  
Shell Sort:  $O(n \log n)$  in practice, significantly faster for larger arrays.
3. Passes:  
Bubble Sort: Moves one element per pass.  
Shell Sort: Moves elements across larger gaps, reducing the need for many passes.

- 2) Explain the significance of the gap in shell sort. How does changing the gap sequence affect the performance of the algorithm?

Significance of the Gap:

- Improves Efficiency: By starting with larger gaps, Shell sort allows elements to move long distances in the early stages, reducing the number of comparisons needed in the later stages.
- Reduces Disorder: Sorting distant elements first helps to diminish large-scale disorder, leading to more efficient sorting as the gap decreases.

Effect of Changing Gap Sequence:

- Performance Impact: Different gap sequences affect the algorithm's efficiency. Common sequences include:
- Optimal Gaps: Choosing an optimal gap sequence reduces the total time complexity, sometimes approaching  $O(n^{3/2})$  or even  $O(n \log n)$  with advanced sequences, compared to the less efficient  $O(n^2)$  for a poor sequence.

- 3) In what scenarios would you choose shell sort over bubble sort? Discuss the types of datasets where shell sort performs better.

Shell sort would be chosen over bubble sort in scenarios where the dataset size is moderate and a more efficient sorting algorithm is desired without the complexity of advanced algorithms like quicksort or mergesort. While bubble sort is simple, it is inefficient with a time complexity of  $O(n^2)$ , making it impractical for anything but small datasets.



### Types of Datasets Where Shell Sort Performs Better:

- **Uniformly Distributed Data:** Shell sort works well on datasets with values that are uniformly distributed because it reduces the number of comparisons needed as the gap decreases.
  - **Nearly Sorted Datasets:** For datasets that are already mostly sorted, Shell sort performs much better than bubble sort. This is because it quickly eliminates small misplacements early in the sorting process.
  - **Small-to-Medium Datasets:** While Shell sort may not outperform more advanced algorithms on very large datasets, it performs significantly better than bubble sort on small to medium-sized datasets due to its efficient handling of distant elements during sorting.
- 4) Provide examples of real-world applications or scenarios where bubble sort or shell sort might be utilized, considering their characteristics.

### BUBBLE SORT:

**Educational Purposes:** Bubble sort is often used in introductory computer science courses to teach basic sorting concepts. It's a straightforward example of how sorting works, making it ideal for learning and understanding algorithmic principles.

**Small Data Sets:** In applications where the dataset is very small (e.g., less than 10 elements), the simplicity of bubble sort can be sufficient, and performance considerations may not be critical.

**Example:** A simple program that needs to sort a handful of user-entered numbers.

### SHELL SORT:

**Data with Insertion-Sort-Like Properties:** Shell sort excels in datasets where insertion sort would perform well, especially when dealing with nearly sorted data. It can be used in situations where sorting needs to happen in-place without extra memory.

**Example:** Organizing records or files that are mostly ordered but have some out-of-place elements, such as log files with slightly disordered timestamps.

**Embedded Systems or Memory-Constrained Devices:** In systems where memory usage must be minimized, Shell sort's in-place sorting nature makes it a good fit. Its performance is decent compared to more memory-intensive algorithms like mergesort.

**Example:** Sorting tasks on embedded systems such as microcontrollers that power small gadgets, sensors, or simple IoT devices.