Batch: D3	**Roll No.:16010123294**
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)

The sorting process is the arrangement of data in a specific order, typically ascending or descending. This makes data easier to analyze, search, and manage, enhancing efficiency in data processing.

Applications of Sorting:

Databases: Enables efficient data retrieval by organizing records systematically.

Searching Algorithms: Sorted data speeds up searches, e.g., binary search.

Data Analytics: Sorts data to analyze trends, like arranging sales by amount.

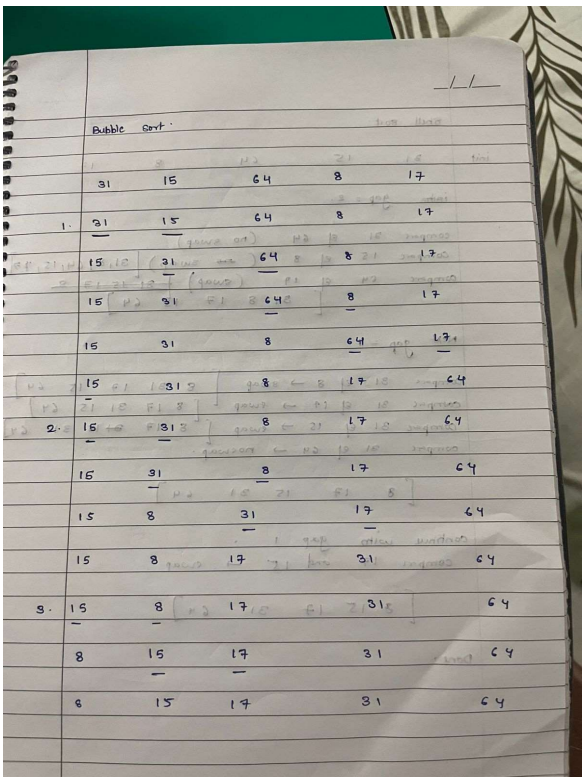
Networking: Sorts packets by priority for optimized routing.

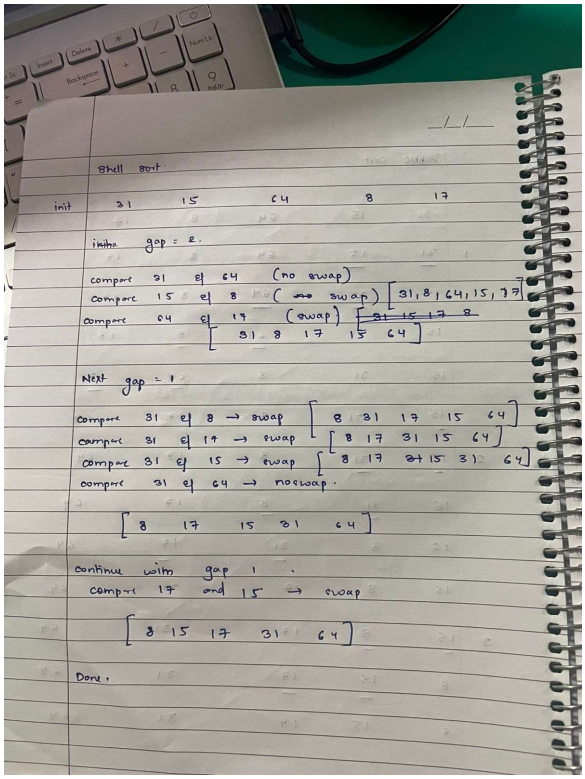
E-commerce: Sorts products by relevance, price, or popularity for user-friendly browsing.

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.

The above is a pen-paper activity, take a picture of the solution and put it here.





Algorithm for Implementation:

Bubble Sort Algorithm

Bubble Sort repeatedly swaps adjacent elements if they are in the wrong order.

1. Start from the beginning of the array.
2. For each element, compare it with the next element.
    - If the current element is greater than the next element, swap them.
3. Repeat this process for each element in the array, reducing the range by one after each pass, as the largest elements "bubble up" to their correct positions.
4. Continue until no swaps are needed, indicating that the array is sorted.

Shell Sort Algorithm

Shell Sort sorts elements far apart from each other first, gradually reducing the gap until it becomes 1, at which point it becomes similar to Insertion Sort.

1. Start with a large gap (typically n / 2, where n is the length of the array).
2. For each gap, perform a gapped insertion sort:
    - Pick an element, then shift it down the array until it is in the correct position relative to the other elements with the same gap.
3. Reduce the gap size by half and repeat step 2.
4. Continue until the gap is reduced to 1, at which point a final pass sorts the array.

Program:

Bubble sort

```
#include <stdio.h>

void bubbleSort(int arr[], int n) {
    int i, j, temp;
    for (i = 0; i < n - 1; i++) {
        printf("Pass %d: ", i + 1);
        for (j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap if elements are in wrong order
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
        // Print array after each pass
        for (int k = 0; k < n; k++) {
            printf("%d ", arr[k]);
        }
        printf("\n");
    }
}

int main() {
    int arr[] = {23, 45, 12, 67, 34, 89, 10, 5, 76};
    int n = sizeof(arr) / sizeof(arr[0]);

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

    bubbleSort(arr, n);

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

    return 0;
}
```

Shell Sort

```
#include <stdio.h>

void shellSort(int arr[], int n) {
    for (int gap = n / 2; gap > 0; gap /= 2) {
        printf("Gap %d: \n", gap);
        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;

            // Print array after each gap pass
            for (int k = 0; k < n; k++) {
                printf("%d ", arr[k]);
            }
            printf("\n");
        }
        printf("\n");
    }
}
```

```
int main() {
    int arr[] = {23, 45, 12, 67, 34, 89, 10, 5, 76};
    int n = sizeof(arr) / sizeof(arr[0]);

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

    shellSort(arr, n);

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

    return 0;
}
```

Output screenshots:

```
Original array: 23 45 12 67 34 89 10 5 76

Pass 1: 23 12 45 34 67 10 5 76 89
Pass 2: 12 23 34 45 10 5 67 76 89
Pass 3: 12 23 34 10 5 45 67 76 89
Pass 4: 12 23 10 5 34 45 67 76 89
Pass 5: 12 10 5 23 34 45 67 76 89
Pass 6: 10 5 12 23 34 45 67 76 89
Pass 7: 5 10 12 23 34 45 67 76 89
Pass 8: 5 10 12 23 34 45 67 76 89

Sorted array: 5 10 12 23 34 45 67 76 89


...Program finished with exit code 0
Press ENTER to exit console.
```

```
Original array: 23 45 12 67 34 89 10 5 76

Gap 4:
23 45 12 67 34 89 10 5 76
23 45 12 67 34 89 10 5 76
23 45 10 67 34 89 12 5 76
23 45 10 5 34 89 12 67 76
23 45 10 5 34 89 12 67 76

Gap 2:
10 45 23 5 34 89 12 67 76
10 5 23 45 34 89 12 67 76
10 5 23 45 34 89 12 67 76
10 5 23 45 34 89 12 67 76
10 5 12 45 23 89 34 67 76
10 5 12 45 23 67 34 89 76
10 5 12 45 23 67 34 89 76

Gap 1:
5 10 12 45 23 67 34 89 76
5 10 12 45 23 67 34 89 76
5 10 12 45 23 67 34 89 76
5 10 12 23 45 67 34 89 76
5 10 12 23 45 67 34 89 76
5 10 12 23 34 45 67 89 76
5 10 12 23 34 45 67 89 76
5 10 12 23 34 45 67 76 89

Sorted array: 5 10 12 23 34 45 67 76 89
```

Conclusion:-Implemented Bubble Sort and Shell sort algorithms on the arrays .

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 reducing the number of comparisons and swaps through the use of gaps, allowing elements to move farther in each step. This "gap-based" sorting allows larger, out-of-place elements to move toward their final positions faster, which minimizes the total number of comparisons needed.

- Shell Sort: Uses a sequence of gaps to compare and move elements far apart, gradually reducing the gap until it becomes 1 (like bubble sort at the end).
- Bubble Sort: Only compares adjacent elements, leading to many small swaps and requiring multiple full passes through the array for larger elements to reach their correct position.

Shell sort is generally much faster because it addresses the inefficiency of adjacent-only swaps in bubble sort.

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

The gap in Shell sort determines how far apart elements are compared and moved in each pass, allowing elements to shift toward their final positions more efficiently.

Gap Sequence Significance:

- Larger gaps at the beginning help to quickly reduce large displacements of elements, speeding up the sorting of distant out-of-place values.
- Smaller gaps as sorting progresses allow finer adjustments, similar to insertion sort, but on a nearly sorted array, which is faster.

Impact on Performance:

Choosing an effective gap sequence significantly affects Shell sort's efficiency. Sequences like Knuth's or Hibbard's offer faster performance by balancing larger and smaller gaps, reducing the number of comparisons and swaps needed compared to simpler sequences like halving, which can be slower on large arrays.

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

Shell sort is preferable over bubble sort when dealing with moderately large datasets where efficient sorting is needed without using extra space, as Shell sort is faster and operates in-place. It's especially effective for:

- Nearly sorted datasets: Shell sort can complete sorting quickly with fewer comparisons.
- Datasets with random or partially ordered elements: The gap sequence allows Shell sort to handle larger, disordered sections more efficiently than bubble sort.

Shell sort is ideal when a simple yet faster alternative to bubble sort or insertion sort is desired, especially for data that isn't extremely large but still benefits from performance improvements.

4) Provide examples of real-world applications or scenarios where bubble sort or shell sort might be utilized, considering their characteristics.

Bubble Sort:

Educational tools: Used for teaching basic sorting concepts due to its simplicity.

Small datasets: Suitable for tiny, already nearly sorted lists where its simplicity and adjacent comparisons are sufficient.

Data integrity checks: Useful when minimal data movement is necessary, as in small embedded systems.

Shell Sort:

Moderate-sized datasets: Employed in applications like scheduling or task prioritization where performance matters but space is limited.

Database optimization: Used for sorting non-continuous data or records that require reduced comparisons.

Cache optimization: Helps in scenarios where data resides in memory cache by leveraging locality with gap-based sorting, improving data access speed in systems with limited memory.

Shell sort's efficiency with partially sorted data makes it a practical choice in scenarios needing faster in-place sorting without the overhead of more complex algorithms.