Batch: E2 **Roll No.:** 16010123325

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)

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:

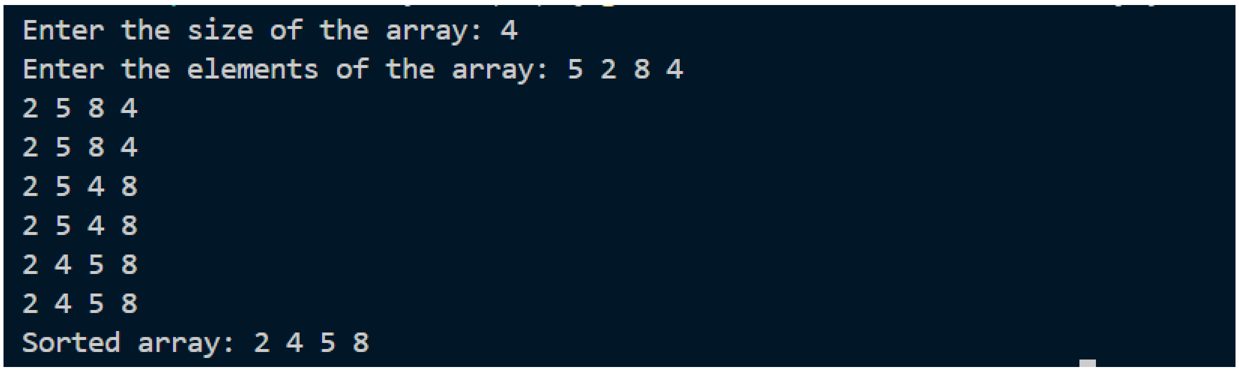
```
Step 1: Start
Step 2: Input the size of the array.
Step 3: For each index i from 0 to size - 2:
    Step 3.1: For each index j from 0 to size - i - 2:
        Step 3.1.1: IF arr[j] > arr[j + 1] THEN:
            Step 3.1.1.1: Swap arr[j] and arr[j + 1].
            Step 3.1.1.2: Print the current state of the array.
        [END OF IF]
    [END OF FOR j]
Step 4: [END OF FOR i]
Step 5: Print the sorted array.
Step 6: Exit.
```

Program:

```
#include <stdio.h>


void bubblesort(int arr[], int size){

  int i,j,temp;

  for(i=0;i<size-1;i++){

    for(j=0;j<size-i-1;j++){

    if(arr[j]>arr[j+1]){

        temp=arr[j];
```

```
          arr[j]=arr[j+1];
          arr[j+1]=temp;
        }
        for(int k=0;k<size;k++){
          printf("%d ",arr[k]);
        }
        printf("\n");
      }
    }
}

int main(){
  int size,i,j,temp;
  printf("Enter the size of the array: ");
  scanf("%d",&size);
  int arr[size];
  printf("Enter the elements of the array: ");
  for(i=0;i<size;i++){
    scanf("%d",&arr[i]);
  }
  bubblesort(arr,size);
  for(i=0;i<size;i++){
  printf("%d ",arr[i]);
  }
}
```

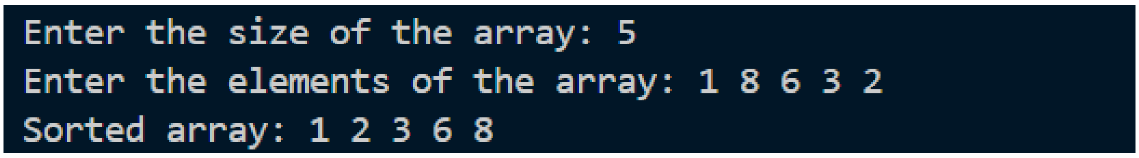
Output screenshots:



```
Enter the size of the array: 4
Enter the elements of the array: 5 2 8 4
2 5 8 4
2 5 8 4
2 5 4 8
2 5 4 8
2 4 5 8
2 4 5 8
Sorted array: 2 4 5 8
```

Optimized

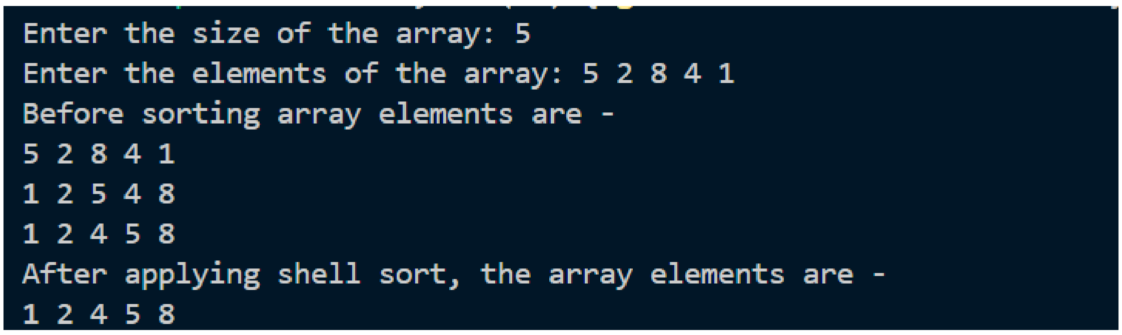
```
#include <stdio.h>

#include <stdbool.h>

void optimizedbubbleSort(int array[], int n)

{

   for(int i=0; i<n; i++)

   {

     bool flag = false;

      for(int j=0; j<n-i-1; j++)

      {

         if(array[j]>array[j+1])

         {

           flag = true;

            int temp = array[j+1];

            array[j+1] = array[j];

            array[j] = temp;

         }

      }
      if(!flag){
        return;
      }
    }
  }

  int main(){
    int size,i,j,temp;
    printf("Enter the size of the array: ");
    scanf("%d",&size);
    int arr[size];
    printf("Enter the elements of the array: ");
    for(i=0;i<size;i++){
      scanf("%d",&arr[i]);
    }
    optimizedbubbleSort(arr,size);
    printf("Sorted array: ");
    for(i=0;i<size;i++){
    printf("%d ",arr[i]);
    }
  }
```



```
Enter the size of the array: 5
Enter the elements of the array: 1 8 6 3 2
Sorted array: 1 2 3 6 8
```

Shell Sort

```
#include <stdio.h>


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


int shell(int a[], int n)
{
    for (int interval = n/2; interval > 0; interval /= 2)
    {
        for (int i = interval; i < n; i += 1)
        {
            int temp = a[i];
            int j;
            for (j = i; j >= interval && a[j - interval] > temp; j -= interval)
                a[j] = a[j - interval];

            a[j] = temp;
        }
        printf("\n");
        printArr(a,n);
    }
    return 0;
}


int main()
{
    int size,i,j,temp;
  printf("Enter the size of the array: ");
  scanf("%d",&size);
  int arr[size];
  printf("Enter the elements of the array: ");
  for(i=0;i<size;i++){
    scanf("%d",&arr[i]);
  }
    printf("Before sorting array elements are - \n");
    printArr(arr, size);
    shell(arr, size);
    printf("\nAfter applying shell sort, the array elements are - \n");
    printArr(arr, size);
    return 0;
}
```



```
Enter the size of the array: 5
Enter the elements of the array: 5 2 8 4 1
Before sorting array elements are -
5 2 8 4 1
1 2 5 4 8
1 2 4 5 8
After applying shell sort, the array elements are -
1 2 4 5 8
```

Conclusion:-
Demonstrated sorting and searching methods

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:

1. Reducing Comparisons: Instead of only comparing adjacent elements like Bubble Sort, Shell Sort compares elements spaced apart (using gaps), sorting subarrays first and reducing the need for many comparisons.

2. Handling Long-Range Displacements: Shell Sort efficiently moves elements over long distances early on, whereas Bubble Sort slowly shifts elements one step at a time through adjacent swaps.

3. Fewer Passes: Shell Sort uses progressively smaller gaps, leading to fewer overall passes compared to Bubble Sort, which performs multiple passes for every adjacent element comparison.

Main Difference: Bubble Sort only swaps adjacent elements, while Shell Sort uses gaps to make larger strides, speeding up the sorting process.

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

In Shell Sort, the gap determines the distance between compared elements. Larger gaps allow faster sorting of distant elements, bringing the array closer to a sorted state early on. As the gap reduces, the algorithm efficiently fine-tunes the list, transitioning into Insertion Sort for the final pass.

Changing the gap sequence affects performance:

- Larger gaps move elements faster, reducing long-range displacements.
- Smaller gaps optimize the final sorting phase.
- Optimized sequences (like Knuth or Sedgewick) significantly improve runtime compared to simple halving (n/2).

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 preferred over Bubble Sort when:

- Efficiency is needed for larger datasets, as Shell Sort is much faster.
- Partially sorted data benefits from Shell Sort's gap-based comparisons.
- Memory constraints require in-place sorting with minimal extra space.
- Small to medium datasets where its simplicity and decent performance make it a good choice.

Shell Sort performs better on:

- Nearly sorted data (fewer passes are needed).
- Moderately sized random data (better time complexity than Bubble Sort).

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

Bubble Sort:

1. Educational Purposes: Used to teach sorting algorithms due to its simplicity.
2. Small Datasets: Suitable for sorting small lists, like user inputs or scores.
3. Nearly Sorted Data: Effective for minor adjustments in almost sorted data.

Shell Sort:

1. Database Management: Used for sorting records in moderate-sized datasets.
2. System Programming: Ideal for memory-constrained environments with small to medium datasets.
3. Image Processing: Applies to sorting pixel values in moderate datasets.
4. Gaming: Useful for sorting player scores or rankings quickly.
5. Simulation: Employed in scientific simulations needing efficient sorting.