Practical - 5

Aim:

Implement a program of Counting Sort & analyse it.

Code:

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

int count_countingSort = 0;
int count_maxElement = 0;

int maxElement(int arr[], int size) {
    int max = arr[0];
    for (int i = 1; i < size; i++) {
        count_countingSort++;
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    count_maxElement = max;
    return max;
}

int arrCopy(int source[], int dest[], int size) {
    for (int i = 0; i < size; i++) {
        dest[i] = source[i];
    }
    return 0;
}

int countingSort(int arr[], int size) {
    int max = maxElement(arr, size);
```

```
    // Create count array and initialize as 0
    int count[max + 1];
    for (int i = 0; i <= max; i++) {
        count[i] = 0;
    }

    // Store count (Frequency) of each element
    for (int i = 0; i < size; i++) {
        count_countingSort++;
        count[arr[i]]++;
    }

    // prefix sum of count array
    for (int i = 1; i <= max; i++) {
        count_countingSort++;
        count[i] += count[i - 1];
    }

    // Build the output array
    int ans[size];

    // To make it stable we are running the loop from size-1 to 0
    for (int i = size - 1; i >= 0; i--) {
        count_countingSort++;
        ans[count[arr[i]] - 1] = arr[i];
        count[arr[i]]--;
    }

    // transfer the sorted elements back to original array
    arrCopy(ans, arr, size);

    return 0;
}
```

```
int main(){
    FILE *fp = freopen("input.txt", "r", stdin);
    int size;
    scanf("%d", &size);
    int *arr = malloc(size * sizeof(int));
    for (int i = 0; i < size; i++) {
        scanf("%d", &arr[i]);
    }

    countingSort(arr, size);

    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\nSize of input: %d", size);
    printf("\nMax Element: %d", count_maxElement);
    printf("\nCount: %d\n", count_countingSort);
    return 0;
}
```

Output:

```
PS C:\Users\Lenovo\Desktop\Sem4\DAA\Prac5> .\count.exe
1 1 1 2 2 2 4 7 9 10
Size of input: 10
Max Element: 10
Count: 39
```

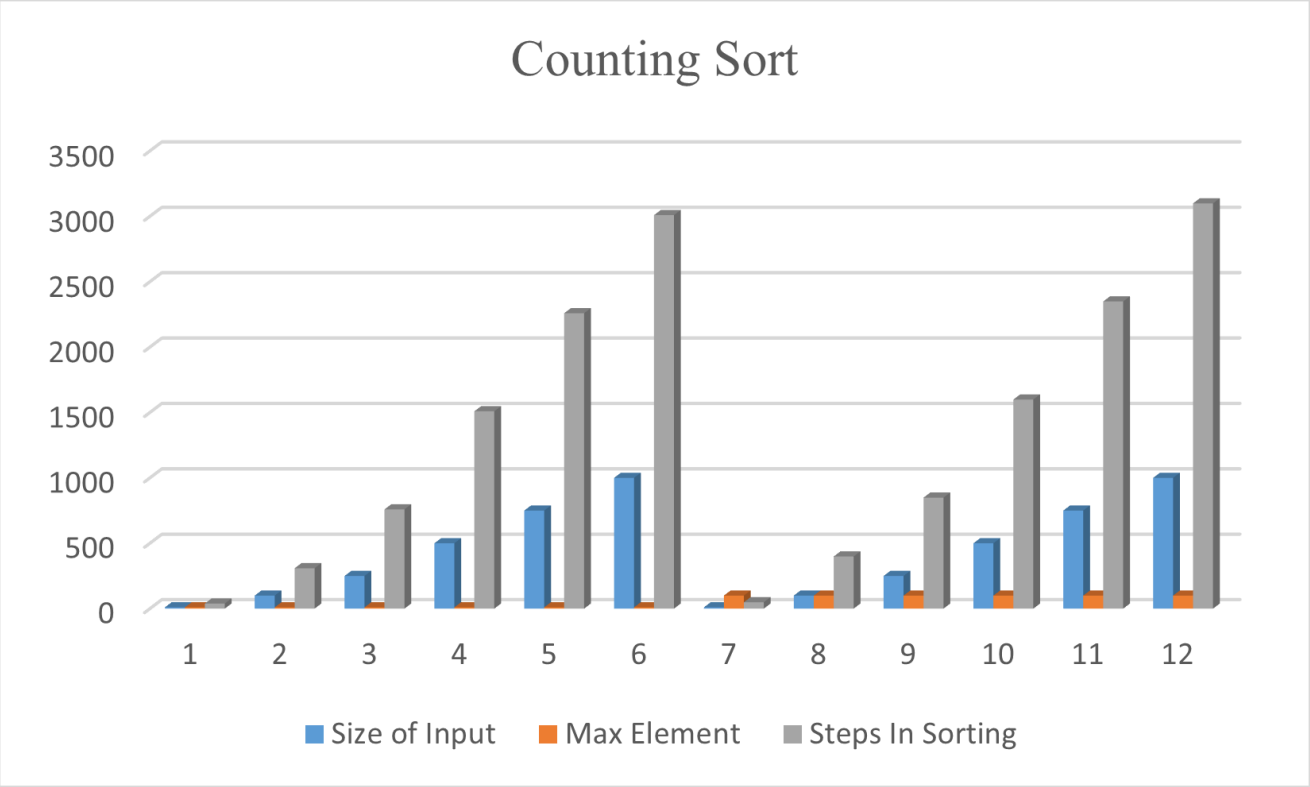
Analysis:

For Size 10:

Size of Input	Max Element	Steps In Sorting
10	10	39
100	10	309
250	10	759
500	10	1509
750	10	2259
1000	10	3009

For Size 100:

Size of Input	Max Element	Steps In Sorting
10	100	49
100	100	399
250	100	849
500	100	1599
750	100	2349
1000	100	3099



Conclusion:

- Best Use Case: It works best when the range of input values (k) is not significantly larger than the number of elements (n).
- Performance: It is a linear time algorithm, O(n+k), often used as a sub-routine in Radix Sort.
- Stability: It is a stable sort, meaning it maintains the relative order of equal elements, which is critical for specific applications.
- Limitations: It cannot be used for floating-point numbers and is memory-intensive for large ranges of input, as it requires O(k) auxiliary space.
- Summary: It is a specialized tool, offering superior speed for specific, constrained datasets.