Fatima Jinnah Women University

Department Of Software Engineering

PROJECT

Course Title

Design and Analysis of Algorithm

Submitted To

Dr. Amir Arsalan

Submitted By

Mariam Fatima

Section: A (4TH Semester)

Registration No: 2021-BSE-020

Date of Submission

July 14, 2023

Contents

COCKTAIL SORT

Cocktail Sort is a comparison-based sorting algorithm that works by repeatedly iterating through the list, comparing adjacent elements, and swapping them if they are in the wrong order.

Other names of Cocktail Sort are;

- Shaker Sort.
- Bi-Directional Sort.
- Cocktail Shaker Sort.
- Shuttle Sort.
- Happy Hour Sort.
- Ripple Sort.

BUBBLE SORT VS COCKTAIL SORT

Cocktail Sort is a variation of the Bubble Sort because it addresses one of its drawbacks—the inefficient performance when dealing with large lists.

Cocktail Sort differs from the Bubble Sort by introducing a bidirectional approach. Instead of always traversing the list from left to right, like in Bubble Sort, Cocktail Sort alternates the direction of its passes. It starts by performing a forward pass through the list, comparing and swapping adjacent elements if necessary. Then it reverses its direction and performs a backward pass through the remaining unsorted portion of the list. This bidirectional movement helps to more quickly move large elements towards their correct positions. It continues with these alternating forward and backward passes until no more swaps are made, indicating that the list is fully sorted. By moving in both directions, Cocktail Sort can potentially reduce the number of iterations required to sort the list compared to traditional Bubble Sort.

ADVANTAGES

1. Cocktail sort is more efficient than bubble sort in certain cases, especially when the array being sorted has a small number of unsorted elements near the end.
2. Cocktail sort is a simple algorithm to understand and implement, making it a good choice for educational purposes or for sorting small datasets.

DISADVANTAGES

1. Cocktail sort has a worst-case time complexity of O(n^2), which means that it can be slow for large datasets or datasets that are already partially sorted.

2. Cocktail sort requires additional bookkeeping to keep track of the starting and ending indices of the subarrays being sorted in each pass, which can make the algorithm less efficient in terms of memory usage than other sorting algorithms.
3. There are more efficient sorting algorithms available, such as merge sort and quicksort, that have better average-case and worst-case time complexity than cocktail sort.

STAGES

Cocktail Sort is divided into two stages;

Forward Pass:

The first stage loops through the array from left to right, just like the Bubble Sort. During the loop, adjacent items are compared and if the value on the left is greater than the value on the right, then values are swapped. At the end of the first iteration, the largest number will reside at the end of the array.

Backward Pass:

The second stage loops through the array in opposite direction, starting from the item just before the most recently sorted item, and moving back to the start of the array. Here also, adjacent items are compared and are swapped if required.

EXAMPLE

Let us consider an example array (5 1 4 2 8 0 2)

First Forward Pass:

(**5 1** 4 2 8 0 2) ? (**1 5** 4 2 8 0 2), Swap since 5 > 1
(1 **5 4** 2 8 0 2) ? (1 **4 5** 2 8 0 2), Swap since 5 > 4
(1 4 **5 2** 8 0 2) ? (1 4 **2 5** 8 0 2), Swap since 5 > 2
(1 4 2 **5 8** 0 2) ? (1 4 2 **5 8** 0 2)
(1 4 2 5 **8 0** 2) ? (1 4 2 5 **0 8** 2), Swap since 8 > 0
(1 4 2 5 0 **8 2**) ? (1 4 2 5 0 **2 8**), Swap since 8 > 2

After the first forward pass, the greatest element of the array will be present at the last index of the array.

First Backward Pass:

(1 4 2 5 **0 2** 8) ? (1 4 2 5 **0 2** 8)
(1 4 2 **5 0** 2 8) ? (1 4 2 **0 5** 2 8), Swap since 5 > 0
(1 4 **2 0** 5 2 8) ? (1 4 **0 2** 5 2 8), Swap since 2 > 0
(1 **4 0** 2 5 2 8) ? (1 **0 4** 2 5 2 8), Swap since 4 > 0
(**1 0** 4 2 5 2 8) ? (**0 1** 4 2 5 2 8), Swap since 1 > 0

After the first backward pass, the smallest element of the array will be present at the first index of the array.

Second Forward Pass:

(0 **1 4** 2 5 2 8) ? (0 **1 4** 2 5 2 8)
(0 1 **4 2** 5 2 8) ? (0 1 **2 4** 5 2 8), Swap since 4 > 2
(0 1 2 **4 5** 2 8) ? (0 1 2 **4 5** 2 8)
(0 1 2 4 **5 2** 8) ? (0 1 2 4 **2 5** 8), Swap since 5 > 2

Second Backward Pass:

(0 1 2 **4 2** 5 8) ? (0 1 2 **2 4** 5 8), Swap since 4 > 2
Now, the array is already sorted, but our algorithm doesn't know if it is completed. The algorithm needs to complete this whole pass without any swap to know it is sorted.
(0 1 **2 2** 4 5 8) ? (0 1 **2 2** 4 5 8)
(0 **1 2** 2 4 5 8) ? (0 **1 2** 2 4 5 8)

ALGORITHM

```
Procedure Cocktail_Sort(Array, n)
swapped = true
start = 0
end = n-1
while(swapped)
swapped = false
for i in range (start,end )
if (Array[i] > Array[i + 1])
swap(Array[i], Array[i+1])
swapped = true
end if
end for loop
if(!swapped)
break
end if
swapped = false
end = end - 1
for i in range(end - 1, start - 1, -1)
if (Array[i] > Array[i + 1])
swap(Array[i], Array[i+1])
swapped = true
end if
end for loop
```

start = start + 1
end while loop
return Array
end algorithm

CODE

```
#include <iostream>

using namespace std;

void Cocktail_Sort(int Array[], int n)

{

    bool swapped = true;

    int start = 0;

    int end = n - 1;


    while (swapped)

    {

        swapped = false;

        for (int i = start; i < end; ++i)

        {

            if (Array[i] > Array[i + 1])

            {

                swap(Array[i], Array[i + 1]);

                swapped = true;

            }

        }


        if (!swapped)
```

```
            break;

        swapped = false;
        --end;

        for (int i = end - 1; i >= start; --i)
        {
            if (Array[i] > Array[i + 1])
            {
                swap(Array[i], Array[i + 1]);
                swapped = true;
            }
        }
        ++start;
    }
}

void Print_Array(int Array[], int n)
{
    for (int i = 0; i < n; ++i)
    cout << Array[i] << " ";
    cout << endl;
}

int main()
```

```
{
    int Array[] = { 5, 1, 4, 2, 8, 0, 2 };
    int n = sizeof(Array) / sizeof(Array[0]);
    Cocktail_Sort(Array, n);
    cout << "Sorted Array:";
    Print_Array(Array, n);
    return 0;
}
```

OUTPUT

Sorted Array: 0 1 2 2 4 5 8

COMPLEXITY

CASE	COMPLEXITY
Best Case	O(n)
Average Case	$O(n^2)$
Worst Case	$O(n^2)$
Space	O(1)

Best Case Complexity:

It occurs when there is no sorting required, i.e., the array is already sorted. The best-case time complexity of cocktail sort is **O(n)**.

Average Case Complexity:

It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of cocktail sort is **$O(n^2)$**.

Worst Case Complexity:

It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of cocktail sort is **$O(n^2)$**.

Space Complexity:

The space complexity of Cocktail Sort is O(1), as it does not require any extra space to sort the elements of the array.

COMPARISON

Time complexities are the same, but Cocktail performs better than Bubble Sort. An example of a list that proves this point is the list (2,3,4,5,1), which would only need to go through one pass of Cocktail Sort to become sorted, but Bubble Sort would take four passes. So, one Cocktail Sort pass should be counted as two Bubble sort Passes. Typically Cocktail Sort is less than two times faster than Bubble Sort.

Another optimization can be that the algorithm remembers where the last actual swap has been done. In the next iteration, there will be no swaps beyond this limit and the algorithm has shorter passes. As the cocktail shaker sort goes bidirectional, the range of possible swaps, which is the range to be tested, will reduce per pass, thus reducing the overall running time slightly.