Course name: Data structure

Course code: COMP2421

Research Report (1)

{Sorting Algorithms}

Instructor: Dr. Ahmad Abusnaina

Name: Dana Imam

Student ID: 1200121

Section number: 2

Contents:

1) Counting Sort:

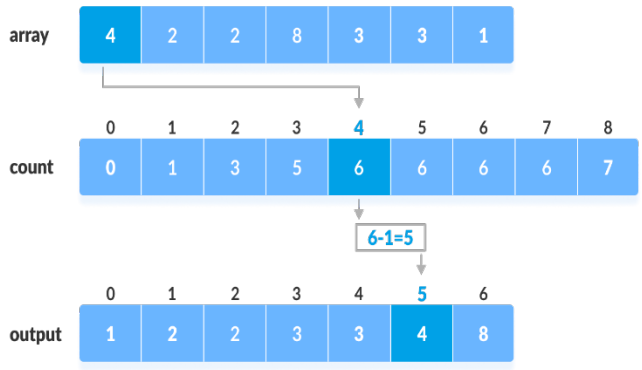
Counting sort method sorts the elements of an array according to its key values without comparing between them, unlike other sort methods like Merge sort or Quick sort. According to the name of this sort method, it sorts by counting how many times each key value occurs in the array, and using those counts to compute an item's index in the final, sorted array. The key values in the array often are nonnegative integers, but this sorting method can be extended to work for negative inputs also.

- **The Algorithm of Counting Sort**

```
countingSort(array, n) // 'n' is the size of array
 max = find maximum element in the given array
create count array with size maximum + 1
Initialize count array with all 0's
for i = 0 to n
find the count of every unique element and
store that count at ith position in the count array
for j = 1 to max
Now, find the cumulative sum and store it in count array
for i = n to 1
Restore the array elements
Decrease the count of every restored element by 1
end countingSort
```

- **How does it work?**

Firstly, the maximum element is found in the given array. This maximum element + 1 will be the length of a new counting array initialized with zeros for each index. The counting array is used to store the number of occurrences of each key value in the given array; by adding one at their respective index in the counting array. After that, the counting array is modified by cumulative count (adding the previous count). Then, an output array with the same length of input array is created, and these cumulative counts in the counting array indicate to the index of each input; by decreasing the count by one, and the



result will be the place for the key value from the original array into the output array, as shown in this figure.

- **Properties:**

  - **Time complexity:**

Worst case	Average case	Best case
O(n)	O(n)	O(n)

  - **Other properties:**

Stability	Space complexity	Space
yes	O(n)	Out-place

the complexity is the same because no matter how the elements are placed in the array, the algorithm goes through n+k ->(n) times.

-The weakness of counting sort:

The inputs in counting sort are restricted. Counting sort only works when the range of potential items in the input is known ahead of time.

If the range of potential values is big, then counting sort requires a lot of space, so there is a space cost.

2) Cocktail Sort:

Cocktail sort is also called as **bi-directional** bubble sort. It depends on comparing between each two adjacent elements in the array, like quick, bubble or selection sort, by traversing through a given array in both directions alternatively. Therefore, it does not only guarantee to move the largest elements in its correct place (at the end of the array) like bubble sort, but also guarantees to put the smallest element in its correct place in the array (at the beginning).

- **The Algorithm of Cocktail Sort**

```
cocktailSort(a, n) // 'a' is the given array, 'n'
is the size of given array
swapped = true
beg = 0
end = n-1
while(swapped)
swapped = false
for i in range from beg to end
if (a[i] > a[i + 1])
swap(a[i], a[i+1])
swapped = true
End if
End for loop
```

```
if(!swapped)
break
end if
swapped = false
end = end - 1
for i in range from end - 1 to beg
if (a[i] > a[i + 1])
swap(a[i], a[i+1])
swapped = true
End if
End for loop
```

- **How does it work?**

Cocktail sort uses two loops: the first one is from left to right exactly like the bubble sort; it compares each two adjacent elements and if the one on the left is greater than the one on the right, they will be swapped, and so on until the end of the array. Hence, the largest element will be at the end of the array by the end of this loop. For the second loop, it follows as the same mechanism as the first loop but reversely; starting from the item just before the most recently sorted

item, and traversing backward the array with comparing between the adjacent elements and swapping if needed.

- ## Properties:

  - **Time complexity:**

Worst case	Average case	Best case
$O(n^2)$	$O(n^2)$	O(n)

  - **Other properties:**

Stability	Space complexity	Space
yes	O(1)	in-place

- **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 **$O(n^2)$**.
-

-The weakness of Cocktail Sort:

Although Cocktail Sort is very intuitive and easy to understand and implement, it is highly impractical for solving most problems.
It has an average and worst-case running time of $O(n^2)$, and can only run on its best-case running time of $O(n)$ when the array is already sorted.

3) Bucket Sort:

Bucket sort is a sorting algorithm which depends on dividing a given unsorted array elements into groups of specific ranges called buckets. This sorting algorithm depends on other sorting algorithms in its sorting process; because when elements of the original array are moved in their suitable buckets, each bucket needs to be sorted using another sorting algorithm, commonly the insertion sort. After that, these elements are gathered again from the sorted buckets and being concatenated into one sorted array. Bucket sort is useful when input is uniformly distributed over a range. For example: a range from (0.0) to (1.0).

- **The Algorithm of Bucket Sort**

```
bucketSort(a[], n)
Create 'n' empty buckets
Do for each array element a[i]
Put array elements into buckets, i.e. insert a[i] into bucket[n*a[i]]
Sort the elements of individual buckets by using the insertion sort.
At last, gather
the sorted buckets.
End bucketSort
```

```
Bucket Sort(A[])
1. Let B[0....n-1] be a new array
2. n=length[A]
3. for i=0 to n-1
4. make B[i] an empty list
5. for i=1 to n
6. do insert A[i] into list B[n a[i]]
7. for i=0 to n
8. do sort list B[i] with insertion-sort
9. Concatenate lists B[0], B[1],....., B[n-1] together in order
   End
```

- **How does it work?**

First, an array of size n is being created, which n is the number of the elements in the given array. Each slot of this new array is considered to be a bucket of a

specific range, for example, if the given array contains numbers with range between 0.0 and 1.0, and this array contains ten element , so n will be ten and each slot indicates to the range of these elements if they are multiplied by n, another example: if the given array contains numbers in range(0 – 23) , we can create five buckets such that first bucket has a range of (0 - 5), and the second one has the range (5-10) and so on. After that, the elements from the original array are being divided into these buckets according to their ranges. The next step is to sort each bucket alone using another method of sorting. And finally, these sorted buckets are gathered, starting from the smallest range to the largest one, and concatenated into one new sorted array with the same size of the original one.

- **Properties:**

  - **Time complexity:**

Worst case	Average case	Best case
$O(n^2)$	O(n)	O(n)

  - **Other properties:**

Stability	Space complexity	Space
yes	O(n)	out-place

  - **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. In Bucket sort, best case occurs when the elements are uniformly distributed in the buckets. The complexity will be better if the elements are already sorted in the buckets.
  - **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending.
  - **Worst case Complexity** when there are elements of close range in the array, they are likely to be placed in the same bucket. This may result in some buckets having a greater number of elements than others, and It makes the complexity depend on the sorting algorithm used to sort the elements of the bucket.

The complexity becomes even worse when the elements are in reverse order. If insertion sort is used to sort elements of the bucket, then time complexity will be $O(n^2)$.

-The weakness of Bucket Sort:

It is not useful if we have a large array because it increases the cost.

Summary:

Property / Sort method	Counting sort	Cocktail sort	Bucket sort
Worst Case	O(n)	$O(n^2)$	$O(n^2)$
Average Case	O(n)	$O(n^2)$	O(n)
Best Case	O(n)	O(n)	O(n)
Stability	yes	yes	yes
Space Complexity	O(n)	O(1)	O(n)
Space	Out-place	In-place	Out-place

-According to the time complexity, it is noticeable that counting sort is better than the others in this report in general; because in its all cases, the time complexity is Big O of n, while the others might take longer time in worst and average cases.

-According to the space and space complexity, Cocktail sort is the best, because it does not use external memory to sort the given array.

References:

- https://www.javatpoint.com/counting-sort
- https://www.geeksforgeeks.org/counting-sort/.
- https://www.programiz.com/dsa/counting-sort
- https://www.geeksforgeeks.org/cocktail-sort/
- https://www.javatpoint.com/cocktail-sort
- https://www.javatpoint.com/bucket-sort
- https://www.programiz.com/dsa/bucket-sort