FACULTY OF ENGINEERING AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE
COMP2421
DATA STRUCTURES AND ALGORITHMS
PROJECT No. 4

{ Sorting algorithm }

Student's name: Sara Issa Date: 24/May/2021
Teacher's name: Dr. Ahmed Abusnaina Section: 1

Table of Contents:

4.1 Introduction:

Sorting algorithm:

A sorting Algorithm is one of the most important types of algorithms in data structure, it's used to rearrangement array or list of items depending to a comparison operator on the elements, from it the new arrangement is deduced in data structure (either numerical order or lexicographical order). It has a lot of methods, although the methods of types of sorting algorithms differ, the goal of all of them is the same, which is the arrangement of data. The importance of the sorting algorithm is highlighted while using other algorithms that require that the data be arranged such as search, insert and merge algorithms. In this report, three new types of sorting algorithm will be illustrated with a description of the properties of each type (time complexity and space complexity, various situations, and stability).

An example of this, as shown in the following figure, the array will be arranged in ascending order based on the ASCII code values for each letter.

4.2 Three types of sorting algorithm:

1. Counting sort:

1.1 Definition:

Counting sort: is a sorting technique based on keys between a specific range, it is a positive integer sorting algorithm. But it doesn't work as negative integer sorting algorithm (because index value should not equal negative number in second array "count Array"), so the solution is find the minimum input value and store count of that minimum input value at index = 0.

1.2 Steps of sorting:

create an array whose size depends on the range of the numbers to be sorted and is equal to (the largest value - the lowest value +1) and make the counter value inside each element in it = 0.
Then it works on the number of values that have distinct key values through a counter, where it passes all the values to be arranged and one increment of the counter for each given number according to the value of the input number. Next, another matrix of the same size is created to find the counter array, which must equal the number of inputs values, by increasing the value of the previous counter on the next counter and storing in it.
Finally, an array is created whose size is equal to the value of the previous counter array and the place is numbered from 1 to the counter array number. All the inputs values are passed in order, and return to the second array depending on the value of the index. The value of the counter is obtained inside it, then the input value is stored in the last array in the place equal to the counter value, then one decrement of counter in the second matrix.
This process continues until all the entered values are passed, and thus the ascending order is obtained to entered values through the last array.

1.3 Counting Sort Example:

Figure No.1: Counting Sort Example

1.4 Counting sort properties:

1-Time complexity: Best case (if the input data array sorted ascending) = Average case (if the input data array not sorted) = Worst case (if the input data array sorted descending) = O (n+k), **where n is the number of elements in input array and k is the range of input.** If the difference between the largest value & the lowest value is large, this will take long time.
2-Space complexity: Extra Space = O (n+k), there are three arrays (input array, count array, result array), it is sorted elements not In-place (out-place).
3-Stability: counting sort is a stable sort, so multiple keys with the same value are placed in the sorted array in the same order that they appear in the input array.

1.5 Algorithm: in pseudocode.

```
function CountingSort(array[x]) is
count = array of k+1 zeros // k = the largest value-the lowest value

for x in input do // For all input value x
    count[array[x]] = count[array[x]] + 1 // Count the instances

for i in 1,2,3,…,k do
    count[i] += count[i-1] // Modifies it into count array

output = array of the same length as the number of input value
for x in input do // x from 0 to length(array)
 /*
Places the element at it's correct position and decrements the value of
count by one
*/
    output[count[array[x]] = array[x]
    --count[array[x]]


/*
 Return the output array after the elements of inputs array is sorted
(sorted array)
```

```
*/
return output
```

2. Bucket sort:

2.1 Definition:

Bucket sort: also known as bin sort, is a type of sorting algorithm that works by putting the elements of inputs array into several groups called buckets. Then, each bucket is sorted individually, either by using the bucket sorting repeatedly or by using other appropriate sorting methods. After completing the sorting of each bucket, they are grouped together while preserving the order in the sorting array, from which we will have followed the required from input array.

The time and space complexity depends on the algorithm used to sort each bucket, the number of buckets to use, and whether the input is uniformly distributed.

2.2 Steps of sorting:

First, it was created empty bucket or list with the same size of input array. Then, insert for every inputs array element array[i] into bucket [n*array[i]], where n is number of elements & i: number of index of array.

Note: if we need sort int value for example then insert into bucket[array[i]].

Next, sort individual buckets using the bucket sorting repeatedly or by using other appropriate sorting methods (In the following example, insertion sort is used). Finally, concatenate all sorted buckets.

2.3 Bucket Sort Example:

sorted array: 0.12, 0.17, 0.21, 0.23, 0.26, 0.39, 0.68, 0.72, 0.78, 0.94

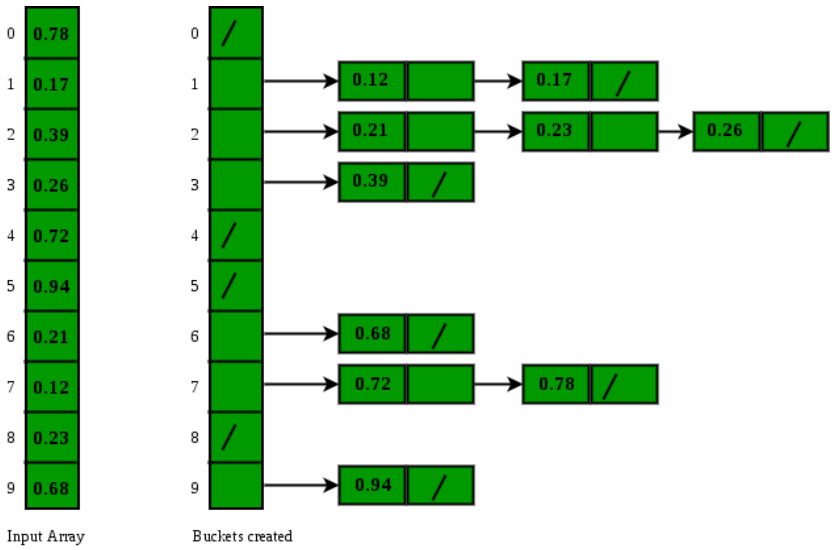

Figure No.2: Bucket Sort Example

2.4 Bucket sort properties:

1-Time complexity:

A. Worst case (if the input data array sorted descending): the worst condition when the elements of inputs array are close to each other, which leads to them being placed in the same bucket, and it is possible that some buckets contain more values than the average number of input values. The worst-case when all the elements are placed in a one bucket, the value of time complexity in this case = O(n^2).

B. Average case (if the input data array not sorted): the average condition when the elements are distributed randomly in the input array. Even if the elements are not distributed uniformly, the value of time complexity in this case = O(n).

C. Best case (if the input data array sorted ascending): the best condition when all the items are distributed uniformly and evenly in different buckets and it is really best when the elements are distributed with the correct sorting in each bucket, the value of time complexity in this case = O(n+k), where O(n) is the complexity for making the buckets and O(k) is the complexity for sorting the elements of the bucket using algorithms having linear time complexity at the best case (such as insertion sort).

2- Space Complexity: O(n+k) because bucket Sort is out-place sorting. This means that more memory is required for the buckets, and this makes it impractical for huge data.

3- Stability: Bucket sort stability depends on the sort used in sorting the buckets.

2.5 Algorithm: in pseudocode.

```
function BucketSort(array[index], n) is
 // n the number of elements in inputs array
 //K the number of elements in each bucket

 // Create n empty buckets, buckets[n]
    buckets ← new array of k empty lists
    M ← the maximum key value in the array (bucket)

// Put elements of inputs array in different buckets
    for index = 0 to length(array) do
        insert array[index] into buckets[n × array[index]]

// Sort each bucket individual
    for index = 0 to k do
        nextSort(buckets[index])

// Concatenate all buckets into sorted array
    return the concatenation of buckets[0],…,buckets[k-1]
```

3. Comb Sort:

3.1 Definition:

Comb sort: it develops on Bubble Sort by using gap (starts with number of elements in inputs array "large value", even it become equal the value 1). This happen by shrink the value of gap of 1.3 in every step. Then comb Sort removes more than one inversion counts with one swap and performs better than bubble sort.

NOTE: by testing comb sort on more 200000 random inputs array, it was concluded that the value of shrink factor = 1.3.

3.2 Steps of sorting:

If that the size of inputs array = n. The starting gap value = n, then the next gap value is calculated by dividing the previous value by the shrink factor = 1.3.

Then the elements in the inputs array are compared through the gap value. The comparison is made between index = 0 and index = the value of gap (If the first value is greater than the second value, the switching process will take place), then increment to the value of each previous index, and this process continues until the last element in the array.

Repeat all previous steps, until the value of gap = 1 is reached, the sorting process is complete and a sorted array is obtained.

3.3 Comb Sort Example:

Figure No.3: Comb Sort Example (inputs array)

The following table shows steps of sorting:

Gap value	Run NO.	Comments
10	1	// No change
10/1.3 = 7	2	// Compare & swap value
7/1.3 = 5	3	// Compare & swap value
5/1.3 = 3	4	// Compare & swap value
3/1.3 = 2	5	// Compare & swap value
2/1.3 = 1	6	// Compare & swap value, after that stop

Figure No.4: Comb Sort Example (sorted array)

3.4 Comb sort properties:

1-Time complexity:

A. Worst case (if the input data array sorted descending): the worst condition when the elements of inputs array is in reversed order, the value of time complexity in this case = O(n^2).
B. Average case (if the input data array not sorted): the value of time complexity in this case = O(nlogn).
C. Best case (if the input data array sorted ascending): the best condition when the elements of inputs array is already sorted, the value of time complexity in this case = O(n).

2-Space complexity: O(1), it doesn't need a space to sort the elements of inputs array; because it uses the current array (In-place).

3- Stability: comb sort is an unstable sorting algorithm as it doesn't sort the repeated elements in the same order as they appear in the inputs array.

3.5 Algorithm: in pseudocode.

```
function CombSort(array[index], n) is
    // n the number of elements in inputs array

    gap := n // Initialize gap size
    shrink := 1.3 // Set the gap shrink factor
    swapped := true //Boolean swapped = true;

    loop while swapped = true
        // Update the gap value for a next comb
        gap := int value of (gap / shrink)

        if gap ≤ 1 then
            gap := 1
            swapped := false // If there are no swaps this pass, we
are done(STOP)

        end if

        // A single "comb" over the input list
        index := 0
        loop while index + gap < n

// Compare all elements with current gap

            if array[index] > array[index + gap] then
                swap(array[index], array[index + gap])
                swapped := true
 /*
 If this assignment never happens within the loop, then there have
been no swaps and the list is sorted
*/
             end if
```

```
// Moving to the next index, while preserving the gap value
            index := index + 1
        end loop
    end loop
end function
```

4.3 Summary:

Sorting algorithm	Time complexity			Space complexity	Stability	in-place/out-place
	Worst case	**Average case**	**Best case**			
Counting sort	O(n+k)	O(n+k)	O(n+k)	O(n+k), Extra space	Stable	Out-place
Bucket sort	O(n^2)	O(n)	O(n+k)	O(n+k), Extra space	Dependable	Out-place
Comb Sort	O(n^2)	O(nlogn)	O(n)	O(1), No extra space	Unstable	In-place

Where n: number of elements in inputs array for all sorting types mentioned before.

K: the range of elements in inputs array in Counting sort, and K: the number of elements in each bucket in Bucket sort.

4.4 References:

1. Issa, S. S. (2021, May 21). *Bucket_sort*. Retrieved from wikipedia: https://en.wikipedia.org/wiki/Bucket_sort

2. Issa, S. S. (2021, May 21). *bucket-sort*. Retrieved from geeksforgeeks: https://www.geeksforgeeks.org/bucket-sort-2/

3. Issa, S. S. (2021, May 23). *Comb_sort*. Retrieved from wikipedia: https://en.wikipedia.org/wiki/Comb_sort

4. Issa, S. S. (2021, May 23). *comb-sort*. Retrieved from geeksforgeeks: https://www.geeksforgeeks.org/comb-sort/

5. Issa, S. S. (2021, May 19). *Counting_sort*. Retrieved from wikipedia: https://en.wikipedia.org/wiki/Counting_sort

6. Issa, S. S. (2021, May 19). *counting-sort*. Retrieved from geeksforgeeks: https://www.geeksforgeeks.org/counting-sort/

7. Issa, S. S. (2021, May 19). *sorting-algorithms*. Retrieved from geeksforgeeks: https://www.geeksforgeeks.org/sorting-algorithms/