Faculty of Engineering & Technology Electrical & Computer Engineering Department

COMP2421 - DATA STRUCTURES AND ALGORITHMS
Project #4

New Sorting Algorithms

Prepared By:

Aya Dahbour 1201738

Instructor:

Dr. Ahmed Abusnaina

Section: 1

Date: Feb-2023

1.Counting Sort

Sorting is a fundamental problem in computer science, and numerous sorting algorithms have been developed over the years to address this issue. One such algorithm is Counting Sort, which is a non-comparison based sorting algorithm that can sort an array of integers in linear time complexity. Counting Sort is particularly useful when the range of input elements is known in advance, and this range is not too large in comparison to the number of elements.

Algorithm Overview

The basic idea behind Counting Sort is to count the number of occurrences of each distinct element in the input array and then place them in order. To accomplish this, the algorithm performs the following steps:

1. Firstly, it finds the maximum element in the input array and creates a count array of size max+1 with all elements initialized to zero.
2. Then it traverses the input array and counts the number of occurrences of each distinct element by incrementing the count array at the corresponding index.
3. It then traverses the count array and computes the prefix sum by adding the current element with the previous element.
4. Next, it creates a sorted array of the same size as the input array and traverses the input array in reverse order. For each element, it finds its position in the sorted array by looking up the count array and decrementing the prefix sum of the element's value.
5. Finally, it places the element at the computed position in the sorted array and decrements the count array at the corresponding index.

Algorithm Properties

Counting Sort has several useful properties:

- Firstly, its time complexity is O(n+k), where n is the number of elements in the input array, and k is the range of input data. This makes Counting Sort a linear-time sorting algorithm, which is much faster than comparison-based sorting algorithms such as Quick Sort and Merge Sort.
- Secondly, its space complexity is O(k), where k is the range of input data. This means that Counting Sort requires extra space for the count array and the sorted array, but the space required is proportional to the range of input data, which is often much smaller than the number of elements in the input array.
- Thirdly, Counting Sort is a stable sorting algorithm, which means that elements with the same value will appear in the same order in the sorted output as they do in the input array. This property is useful when sorting objects based on multiple keys in a specific order.
- Finally, Counting Sort is not an in-place sorting algorithm, as it requires extra space for the count array and the sorted array.

How Input Data Affects the Running Time of Sorting Algorithms?

The running time of sorting algorithms can vary depending on the input data. Here's how the running time of sorting algorithms is affected by the input data array:

- Sorted (ascending): If the input data array is already sorted in ascending order, sorting algorithms will still perform the standard sorting operation but with fewer comparisons and swaps. For comparison-based sorting algorithms like Quick Sort and Merge Sort, the running time will decrease to O(n log n) due to fewer comparisons. However, for non-comparison based sorting algorithms like Counting Sort, the running time will remain unaffected and will remain O(n + k).
- Sorted (descending): When the input data array is sorted in descending order, sorting algorithms take longer to sort compared to randomly ordered arrays. For comparison-based sorting algorithms such as Quick Sort and Merge Sort, the worst-case time complexity of O(n^2) applies. On the other hand, non-comparison based sorting algorithms like Counting Sort have a running time that is not affected by the input data order and will remain at O(n + k).
- Not sorted: If the input data array is unsorted, the standard worst-case time complexity will be the running time of sorting algorithms. Comparison-based sorting algorithms like Quick Sort and Merge Sort have a worst-case time complexity of O(n^2), but their running time is typically closer to O(n log n) for random data. Non-comparison based sorting algorithms like Counting Sort have a running time of O(n + k) that is not influenced by the input data distribution.

Algorithm Pseudocode:

```
countingSort(array, size)
  maxVal = maximum value in array
  countArray = array of (maxVal + 1) zeros
  output = empty array of size size
  for i = 0 to size - 1 do
    countArray[array[i]] += 1
  for i = 1 to maxVal do
    countArray[i] += countArray[i-1]
  for i = size - 1 down to 0 do
    output[countArray[array[i]]-1] = array[i]
    countArray[array[i]] -= 1
  for i = 0 to size - 1 do
      array[i] = output[i]
```

Situations for Using Counting Sort

Counting Sort is suitable for situations where the input element range is pre-known and not large compared to the number of elements. It can be used as a subroutine in other sorting algorithms like Radix Sort. Counting Sort is stable and can sort objects based on multiple keys in a specific order, such as sorting people by age and then by name.

2. Cocktail Sort:

also known as Bidirectional Bubble Sort or Shaker Sort, is a sorting algorithm that is a variation of Bubble Sort. It works by repeatedly iterating through the list in both directions, swapping adjacent elements if they are in the wrong order.

Algorithm Description:

Cocktail Sort is a variation of Bubble Sort that works by repeatedly iterating through the list in both directions, swapping adjacent elements if they are in the wrong order. The algorithm starts by iterating through the list from left to right, swapping adjacent elements if they are in the wrong order. Then it iterates from right to left, again swapping adjacent elements if they are in the wrong order. This process continues until the list is fully sorted, with no more swaps needed.

How works in more details:

1. The first pass starts at the beginning of the array and compares each pair of adjacent elements. If the elements are in the wrong order, they are swapped.
2. The second pass starts at the end of the array and compares each pair of adjacent elements in the opposite direction. If the elements are in the wrong order, they are swapped.
3. The algorithm continues to alternate between passes until no more swaps are necessary, indicating that the array is now sorted.

Algorithm Properties:

- Time Complexity: Cocktail Sort has a worst-case time complexity of O(n^2), where n is the number of elements in the list. This is the same as the worst-case time complexity of Bubble Sort.
- Space Complexity: The space complexity of Cocktail Sort is O(1), as it only requires a constant amount of additional memory to store the indices used in the algorithm.
- Stability: Cocktail Sort is a stable sorting algorithm, meaning that it preserves the relative order of equal elements in the list. This is achieved by only swapping adjacent elements if they are in the wrong order, rather than swapping elements based on their values alone.
- In-Place Sorting: Cocktail Sort is an in-place sorting algorithm, meaning that it does not require any additional memory to sort the list.

Advantages and Disadvantages:

Advantages:

- Simple to implement
- In-place sorting
- Stable sorting

Disadvantages:

- Worst-case time complexity is O(n^2), making it inefficient for large lists
- Cocktail Sort performs more swaps than Bubble Sort, making it slower in practice

The running time of the Cocktail Sort algorithm depends on the input data array:

Here's how the running time is affected by different input data types:

1. Sorted (ascending): If the input data array is already sorted in ascending order, the Cocktail Sort algorithm will make one pass through the array and terminate, as no swaps will be made. Therefore, the time complexity of Cocktail Sort for a sorted (ascending) input data array is O(n), which is the best-case scenario for the algorithm.

2. Sorted (descending): If the input data array is sorted in descending order, the Cocktail Sort algorithm will perform similarly to the standard Bubble Sort algorithm. The algorithm will make n-1 passes through the array, swapping adjacent elements until the largest element "bubbles up" to the end of the array. Then, the algorithm will make n-2 passes through the array, and so on, until the array is fully sorted. The time complexity of Cocktail Sort for a sorted (descending) input data array is O(n^2), which is the worst-case scenario for the algorithm.

3. Not sorted: If the input data array is not sorted, the Cocktail Sort algorithm will perform similarly to the standard Bubble Sort algorithm. The algorithm will make passes through the array in both directions, swapping adjacent elements that are in the wrong order. After each pass, the algorithm will identify the largest and smallest elements in the unsorted part of the array, and only swap elements within that range on the next pass. The number of passes required to sort the array depends on the initial state of the array. The time complexity of Cocktail Sort for an unsorted input data array is also O(n^2), which is the worst-case scenario for the algorithm.

Algorithm Pseudocode

```
function cocktailSort(array):
    n = length(array)
    left = 0
    right = n - 1
    while left < right:
        # Traverse the array from left to right
        for i from left to right:
            if array[i] > array[i+1]:
                swap(array, i, i+1)
        right = right - 1
        # Traverse the array from right to left
        for i from right to left:
            if array[i] > array[i+1]:
                swap(array, i, i+1)
        left = left + 1
    return array
```

3.Comb Sort

Sorting is a fundamental operation in computer science that involves arranging a set of data in a specific order. There are several algorithms for sorting data, and each has its strengths and weaknesses. One such algorithm is the Comb Sort algorithm, which is an improvement over the Bubble Sort algorithm.

What is the Comb Sort Algorithm?

The Comb Sort algorithm is a comparison-based sorting algorithm that works by performing a series of bubble sort-like steps. The algorithm gets its name from the way it shrinks the gap between elements being compared, which resembles the action of combing hair. The algorithm works by comparing elements that are a certain distance apart and then gradually reducing this distance until it reaches one, which is the final comparison.

Algorithm Properties

- Time Complexity: The time complexity of the Comb Sort algorithm is O(n^2), which means that it takes quadratic time to sort n elements. However, the algorithm's time complexity can be improved by using a shrinking factor to reduce the gap between elements being compared. This can result in a time complexity of O(n log n) in some cases.
- Space Complexity: The Comb Sort algorithm has a space complexity of O(1), which means that it sorts elements in place without requiring any additional space.
- Stability: The Comb Sort algorithm is not a stable sorting algorithm. This means that it does not maintain the relative order of elements with equal keys. If two elements have the same key, their relative order may be changed after sorting.
- In-Place Sorting: The Comb Sort algorithm is an in-place sorting algorithm. This means that it sorts elements without requiring additional memory for storing the sorted elements.

How does the Comb Sort Algorithm work?

The Comb Sort algorithm works by comparing elements that are a certain distance apart and swapping them if they are in the wrong order. The distance between elements being compared starts off as the length of the list and is gradually reduced by a factor known as the shrink factor. The shrink factor is typically between 1.3 and 1.4 and helps to reduce the time complexity of the algorithm.

The algorithm starts off by comparing elements that are distance d apart, where d is the length of the list. It then compares elements that are d - 1 apart and continues this process until it compares adjacent elements. At this point, the algorithm is essentially performing a bubble sort. The process continues until the gap between elements being compared is one, at which point the algorithm terminates.

Algorithm Pseudocode

```
function combSort(array):
    gap = array.length
    shrink = 1.3
    swapped = true
    while gap > 1 or swapped:
        gap = floor(gap / shrink)
        if gap < 1:
            gap = 1
        i = 0
        swapped = false
        while i + gap < array.length:
            if array[i] > array[i + gap]:
                array[i], array[i + gap] = array[i + gap], array[i]
                swapped = true
            i = i + 1
    return array
```

How the Initial Order of Input Data Affects the Running Time of the Comb Sort Algorithm?

The running time of the Comb Sort algorithm depends on the initial order of the input data array. Here's how the algorithm behaves in different scenarios:

1. Sorted (ascending): If the input data array is already sorted in ascending order, the Comb Sort algorithm will still perform a full pass through the array, but it will not swap any elements. This means that the time complexity will be O(n) for this case, which is a best-case scenario for the algorithm.
2. Sorted (descending): If the input data array is sorted in descending order, the Comb Sort algorithm will still work, but it will be much slower than in the best-case scenario. In fact, it will perform as poorly as it would for an unsorted array, since the algorithm always swaps adjacent elements. Therefore, the time complexity will be O(n^2) for this case, which is the worst-case scenario for the algorithm.
3. Not sorted: If the input data array is not sorted, the Comb Sort algorithm will need to perform several passes through the array, swapping adjacent elements that are out of order. The time complexity for this case will be O(n^2), which is the average-case scenario for the algorithm.

4. Gnome Sort:

Gnome Sort is a simple, easy-to-understand sorting algorithm that sorts an array of elements by comparing adjacent elements and swapping them if they are in the wrong order. The algorithm gets its name from the behavior of a garden gnome, who sorts a line of flower pots by picking up the pots and placing them in the correct order, one by one.

Algorithm Steps:

1. Start at the first element of the array.
2. Compare the current element with the previous element.
3. If the current element is greater than or equal to the previous element, move to the next element.
4. If the current element is smaller than the previous element, swap them and move to the previous element.
5. Repeat steps 2-4 until you reach the beginning of the array or the current element is greater than or equal to the previous element.
6. Move to the next element and repeat steps 2-5 until the end of the array is reached.

Algorithm Properties:

- Time Complexity: The time complexity of Gnome Sort is O(n^2), where n is the number of elements in the array. This makes it an inefficient algorithm for large arrays.
- Space Complexity: The space complexity of Gnome Sort is O(1), as it sorts the array in-place without requiring any extra memory.
- Stability: Gnome Sort is a stable sorting algorithm, which means that it preserves the relative order of elements with equal values. This makes it useful in situations where the original order of elements needs to be preserved.
- In-Place Sorting: Gnome Sort is an in-place sorting algorithm, which means that it sorts the array without requiring any extra memory or creating a new array. This makes it useful in situations where memory is limited and creating a new array is not feasible.

Advantages and Disadvantages:

Advantages:

• Simple and easy to implement.
• Stable sorting algorithm.
• In-place sorting algorithm.

Disadvantages:

• Not efficient for large arrays.
• Not suitable for real-time applications.
• Requires multiple passes over the array.

The running time of the Gnome Sort algorithm depends on the input data array:

1. Sorted (ascending): If the input array is already sorted in ascending order, Gnome Sort will take O(n) time to traverse the array, but no swaps will be made, as each element is already in its correct position.
2. Sorted (descending): If the input array is sorted in descending order, Gnome Sort will take O(n^2) time to sort the array. In this case, the pointer will always move to the beginning of the array, as each element is smaller than the previous element, and each element will need to be swapped with the previous element.
3. Not sorted: If the input array is not sorted, Gnome Sort will take O(n^2) time to sort the array. In this case, the pointer will move back to the beginning of the array several times, swapping elements to their correct position. The number of swaps will depend on how much the array is out of order.

Algorithm Pseudocode

```
function gnomeSort(arr):
    i = 1
    j = 2
    n = length(arr)

    while i < n:
        if arr[i-1] <= arr[i]:
            i = j
            j = j + 1
        else:
            temp = arr[i-1]
            arr[i-1] = arr[i]
            arr[i] = temp
            i = i - 1
            if i == 0:
                i = j
                j = j + 1
    return arr
```

5. Pancake Sort:

Pancake sort is an algorithm used for sorting a list of elements by flipping subarrays of the list. The algorithm repeatedly searches the list for the largest element, then flips the subarray from the beginning of the list to the position of the largest element, and then flips the entire list. This process is repeated until the list is sorted.

Algorithm Steps:

1. Search the input list for the largest element.
2. Flip the subarray from the beginning of the list to the position of the largest element, so that the largest element is now at the beginning of the list.
3. Flip the entire list, so that the largest element is now at the end of the list.
4. Repeat steps 1-3 for the remaining unsorted elements, until the entire list is sorted.

Algorithm Properties:

Time Complexity: The worst-case time complexity of pancake sort is O(n^2), where n is the number of elements in the input list. The average-case time complexity is also O(n^2). However, the best-case time complexity is O(n), when the input list is already sorted.

• Space Complexity: The space complexity of pancake sort is O(1), as the algorithm sorts the input list in place without requiring any additional memory.

• Stability: Pancake sort is stable, as it only swaps adjacent elements in the list.

• In-Place Sorting: Pancake sort is an in-place sorting algorithm, as it sorts the input list in place without requiring any additional memory.

How the Initial Order of Input Data Affects the Running Time of the Pancake Sort Algorithm?

- Sorted (ascending or descending): If the input data array is already sorted in ascending or descending order, pancake sort will have a best-case scenario where no flips are necessary, and the time complexity will be O(n).

- Not sorted: If the input data array is not sorted, the performance of pancake sort will depend on the number of inversions in the list. An inversion is defined as a pair of elements in the list that are out of order. If the number of inversions is small, pancake sort can sort the list in O(n) time. However, if the number of inversions is large, the time complexity of pancake sort can degrade to O(n^2).

Algorithm Pseudocode

```
function pancakeSort(array)
    for i from n to 1
        maxIndex = findMaxIndex(array, i)
        if maxIndex != i
            flip(array, maxIndex)
            flip(array, i)
    return array

function findMaxIndex(array, n)
    maxIndex = 1
    for i from 1 to n
        if array[i] > array[maxIndex]
            maxIndex = i
    return maxIndex

function flip(array, k)
    i = 1
    while i < k
        temp = array[i]
        array[i] = array[k]
        array[k] = temp
        i++
        k--
```

Conclusion

Sorting algorithms are important tools used in computer science for efficient data organization. Counting Sort is suitable for integer data values in a defined range and has a time complexity of O(n+k). Cocktail Sort and Gnome Sort are simple algorithms but may be inefficient for larger arrays. Comb Sort is an improvement over bubble sort and Cocktail Sort that reduces the number of swaps needed to sort an array. Pancake Sort is an inefficient sorting algorithm and is rarely used in practice. Selecting the right sorting algorithm depends on the data type, size of the input array, and required time complexity. Counting Sort, Cocktail Sort, Comb Sort, Gnome Sort, and Pancake Sort each have their own strengths and weaknesses, making them suitable for different scenarios.

References

[1] https://www.geeksforgeeks.org/counting-sort/

[2] https://en.wikipedia.org/wiki/Counting_sort

[3] https://www.geeksforgeeks.org/cocktail-sort/

[4] https://en.wikipedia.org/wiki/Cocktail_sort

[5] https://www.tutorialspoint.com/articles/index.php

[6] https://www.geeksforgeeks.org/comb-sort/

[7] https://rosettacode.org/wiki/Sorting_algorithms/Gnome_sort

[8] https://en.wikipedia.org/wiki/Gnome_sort

[9] https://rosettacode.org/wiki/Sorting_algorithms/Pancake_sort

[10] https://www.geeksforgeeks.org/pancake-sorting/