|  |
| --- |
| aCOMP2421 (Fall2022/2023) |
| Project #4 |
| **Prepared By:**  Khalid Sami "1193137"  **Instructor :**  Radi jarar  **Section :** 2  Contents  [Introduction 3](#_Toc127642687)  [Counting Sort 4](#_Toc127642688)  [Properties: 4](#_Toc127642689)  [Running time: 4](#_Toc127642690)  [Advantages 5](#_Toc127642691)  [Disadvantages 5](#_Toc127642692)  [Bucket Sort 6](#_Toc127642693)  [Properties: 6](#_Toc127642694)  [Running time: 6](#_Toc127642695)  [Advantages 7](#_Toc127642696)  [Disadvantages 7](#_Toc127642697)  [Comb Sort 8](#_Toc127642698)  [Properties 8](#_Toc127642699)  [Running time: 9](#_Toc127642700)  [Algorithm 9](#_Toc127642701)  [Advantages: 9](#_Toc127642702)  [Disadvantages: 9](#_Toc127642703)  [Gnome Sort 10](#_Toc127642704)  [Properties: 10](#_Toc127642705)  [Running time: 11](#_Toc127642706)  [Advantages 11](#_Toc127642707)  [Disadvantages 11](#_Toc127642708)  [Cocktail Sort 12](#_Toc127642709)  [Algorithm: 12](#_Toc127642710)  [Properties 13](#_Toc127642711)  [Running time: 13](#_Toc127642712)  [Advantages: 13](#_Toc127642713)  [Disadvantages: 13](#_Toc127642714)  [Summary 14](#_Toc127642715)  [Reference 15](#_Toc127642716) Introduction Sorting algorithms play a crucial role in computer science, and they find extensive use in various fields, including databases, search engines, and scientific simulations. The primary objective of a sorting algorithm is to organize a set of data into a particular order, commonly in ascending or descending order. Sorting algorithms differ in their properties, including time complexity, space complexity, stability, and in-place sorting, which can influence their efficiency under different conditions. This research paper examines and compares five sorting algorithms: Counting Sort, Bucket Sort, Comb Sort, Gnome Sort, and Cocktail Sort. Each algorithm is described in detail, and its properties are discussed. Moreover, their performance is analyzed for various input data arrays. Finally, a summary of the outcomes and a comparison of the algorithms based on their properties is provided. Counting Sort Counting Sort is a simple, non-comparison sorting algorithm that operates by counting the number of elements that have distinct key values and calculating their positions in the final sorted array. This algorithm is good when the range of input data is small rather to the input size. It has a time complexity of O(n+k), where n is the number of elements to be sorted, and k is the range of input data.  The Counting Sort algorithm works as follows:   1. Find the maximum value in the input data array. 2. Create a new array with size equal to the maximum value plus one and initialize all elements to zero. 3. Traverse the input data array, and for each element, increment the count of the corresponding element in the new array. 4. Modify the count array so that each element at index i contains the sum of the previous counts. 5. Traverse the input data array again in reverse order, and for each element, place the element in its correct sorted position in the output array, based on the element's count.   Counting Sort is a stable sorting algorithm, meaning that it preserves the relative order of equal elements in the sorted output. It is also an in-place sorting algorithm, meaning that it sorts the input data array without requiring any additional storage.  If the input data array is already sorted (ascending or descending), Counting Sort has a time complexity of O(n), as it only needs to traverse the input data array once to count the elements. If the input data array is not sorted, Counting Sort has a time complexity of O(n+k). However, this algorithm requires a large amount of additional memory to store the count array, making it impractical for large input sizes or large range of input data.  In summary, Counting Sort is a simple and efficient algorithm for small input sizes and small range of input data. It has a linear time complexity and is stable and in-place. However, it is not practical for large input sizes or large range of input data, as it requires a lot of additional memory. Properties: Time complexity: O(n+k), where n is the number of elements in the input array, and k is the range of the input data.  Space complexity: O(n+k).  Stable: Yes.  In-place: No. Running time: Sorted (ascending): O(n+k).  Sorted (descending): O(n+k).  Not sorted: O(n+k).  Example of how the data will look before and after applying the Counting Sort algorithm on an input array.  Let's say we have the following unsorted array of integers:  [9, 6, 2, 7, 2, 1, 0, 8, 5, 3, 4, 9]  We can apply the Counting Sort algorithm on this array to sort it in ascending order. After applying the Counting Sort algorithm, the array will be sorted as follows:  [0, 1, 2, 2, 3, 4, 5, 6, 7, 8, 9, 9]  As you can see, the output array is sorted in ascending order, and all the duplicate elements in the input array are preserved in their original order. The time complexity of the Counting Sort is linear, O(n+k), where n is the size of the input array and k is the range of values in the array. the range of values is 10 (from 0 to 9), so the time complexity is O(n+10), which simplifies to O(n). The space complexity of the algorithm is also O(n+k), which in this case is O(n+10), or simply O(n), since k is a constant value.  Advantages:   * Counting Sort has a time complexity of O(n+k), where n is the number of elements in the input data array and k is the range of the input data values. This makes it very fast for input data arrays with a small range of values, particularly if k is much smaller than n. * Counting Sort is a stable sorting algorithm, meaning that it retain the relative order of equal elements in the input data array. * Counting Sort is an in-place sorting algorithm, meaning that it sorts the input data array without requiring additional memory for intermediate values.   Disadvantages:   * Counting Sort requires additional memory to store the counts of each input data value, as well as an output array to hold the sorted elements. This can be a significant disadvantage for big input data arrays or input data arrays with a large range of values. * Counting Sort is not suitable for sorting input data arrays with negative values or non-integer values, since the counting array requires non negative number indices. * Counting Sort can be slow and inefficient for input data arrays with a large range of values, since it requires counting each occurrence of every input data value.   Counting Sort is a very fast and efficient algorithm for sorting input data arrays with a small range of values, particularly if the input data values are non-negative integers. However, it may not be the best choice for sorting input data arrays with a large range of values or input data arrays with negative values. |
| Bucket Sort Bucket Sort is a comparison-based sorting algorithm that works by dividing an input data array into several "buckets" based on their values, sorting the buckets individually using a different sorting algorithm (usually Insertion Sort), and then concatenating the sorted buckets to obtain the final sorted array. Bucket Sort is typically used when the input data is uniformly distributed over a range. It has a time complexity of O(n+k), where n is the number of elements to be sorted, and k is the number of buckets.  The Bucket Sort algorithm works as follows:   1. Create an array of empty buckets. 2. Traverse the input data array, and for each element, assign the element to the appropriate bucket based on its value. 3. Sort each bucket individually using a different sorting algorithm (usually Insertion Sort). 4. Concatenate the sorted buckets to obtain the final sorted array.   Bucket Sort is a stable sorting algorithm if the sorting algorithm used to sort the buckets is stable. It is not an in-place sorting algorithm, as it requires additional memory to store the buckets.  If the input data array is already sorted (ascending or descending), Bucket Sort has a time complexity of O(n), as it only needs to traverse the input data array once to assign the elements to their respective buckets. If the input data array is not sorted and the buckets are small, Bucket Sort can have a time complexity close to O(n). However, if the input data is not uniformly distributed over the range, some buckets may be significantly larger than others, causing Bucket Sort to be slower. In this case, it may be better to use a different sorting algorithm.  In summary, Bucket Sort is a useful algorithm for uniformly distributed input data, with a linear time complexity and stability. However, it requires additional memory and may be slower if the input data is not uniformly distributed or if the buckets are large. Properties:  * Time complexity: O(n+k), where n is the number of elements in the input array, and k is the number of buckets. * Space complexity: O(n+k). * Stable: Yes. * In-place: No.  Running time:  * Sorted (ascending): O(n). * Sorted (descending): O(n). * Not sorted: O(n+k).  1. Bucket Sort is a good choice for data that is uniformly distributed over a range, because it can reduce the number of comparisons required to sort the data. 2. Bucket Sort is useful when the input data contains integers or floating-point numbers, but it may not be practical for other types of data, such as strings. 3. The performance of Bucket Sort can be impacted by the choice of sorting algorithm used to sort the individual buckets. For example, if we use Insertion Sort to sort the buckets, the time complexity of Bucket Sort becomes O(n + k^2). However, other sorting algorithms such as Merge Sort or Quick Sort can be used instead to reduce the time complexity to O(n log n). 4. Bucket Sort is not a stable sorting algorithm if the sorting algorithm used to sort the individual buckets is not stable. This means that equal elements in the input array may not be sorted in the same order in the output array.   Advantages**:**   * Bucket Sort is a linear time sorting algorithm for uniformly distributed input data, making it much faster than other comparison-based sorting algorithms such as Quick Sort and Merge Sort. * It is efficient for sorting input data arrays with a small range of values, where each value in the input data array is within a specific range or bucket. This can result in a significant reduction in the number of comparisons required to sort the input data array. * Bucket Sort is easy to implement and understand, making it a good choice for small to medium-sized input data arrays. * It is an in-place sorting algorithm, meaning that it does not require any additional storage beyond the input data array and the buckets.   Disadvantages**:**   * Bucket Sort is not suitable for sorting input data arrays with non-uniform distributions, where some buckets may contain a large number of elements while others may be empty or contain very few elements. In such cases, the time complexity of Bucket Sort can approach O(n^2), making it inefficient. * Bucket Sort requires prior knowledge of the range and distribution of the input data, which may not always be available or practical. * Bucket Sort may not be suitable for sorting large input data arrays, as the number of buckets required for sorting the input data can become very large, which can result in a significant increase in the memory requirements of the algorithm. |
| Comb Sort Comb Sort is a simple comparison-based sorting algorithm that improves upon the Bubble Sort algorithm by introducing the concept of "shrink factor." This factor reduces the gap between adjacent elements being compared, which increases the efficiency of the algorithm. The time complexity of Comb Sort is O(n^2) in the worst case, but it can perform much better than Bubble Sort for certain types of input data.  The Comb Sort algorithm works as follows:   1. Set the initial gap value to the length of the input data array. 2. Traverse the input data array, comparing adjacent elements that are separated by the gap value. 3. If two adjacent elements are out of order, swap them. 4. Reduce the gap value by a shrink factor (typically 1.3) and repeat steps 2-4 until the gap value is less than or equal to 1. 5. Traverse the input data array one last time to ensure that all elements are in their correct positions.   Comb Sort is not a stable sorting algorithm, meaning that the relative order of equal elements in the input data array is not preserved in the sorted output. However, it is an in-place sorting algorithm, meaning that it sorts the input data array without requiring any additional storage.  If the input data array is already sorted (ascending or descending), Comb Sort has a time complexity of O(n). If the input data array is in reverse order, Comb Sort has a time complexity of O(n^2). However, for random input data, Comb Sort can perform much better than Bubble Sort and have a time complexity of O(n^2/2^p), where p is the number of comparisons made before the gap value is reduced to 1.  In summary, Comb Sort is a simple algorithm that can be more efficient than Bubble Sort for certain types of input data. It has a time complexity of O(n^2), is not stable, but is in-place. However, it may not be the best choice for large input data arrays, as its performance can be unpredictable.  Properties:   * Time complexity: O(n^2), in the worst case, where n is the number of elements in the input array. * Space complexity: O(1). * Stable: No. * In-place: Yes.  Running time:  * Sorted (ascending): O(n). * Sorted (descending): O(n^2). * Not sorted: O(n^2).  Algorithm  * input = [8, 4, 1, 6, 9, 2, 7, 5, 3]            output=[1,2,3,4,5,6,7,8,9]   In the above example, the input array is [8, 4, 1, 6, 9, 2, 7, 5, 3]. The Comb Sort algorithm is then applied to this input array, resulting in the sorted output array [1, 2, 3, 4, 5, 6, 7, 8, 9]. Advantages:  * Comb Sort is an improvement over Bubble Sort and can be faster for certain types of input data. * Comb Sort is an in-place sorting algorithm, meaning that it sorts the input data array without requiring any additional storage. * Comb Sort is easy to implement and understand.  Disadvantages:  * Comb Sort has a worst-case time complexity of O(n^2), making it less efficient than some other sorting algorithms for large input data arrays. * Comb Sort is not a stable sorting algorithm, meaning that the relative order of equal elements in the input data array is not preserved in the sorted output. * Comb Sort can be less predictable than some other sorting algorithms, as its performance can vary depending on the input data.  Gnome Sort Gnome Sort is a simple comparison-based sorting algorithm that works by repeatedly comparing adjacent elements and swapping them if they are in the wrong order. It is named after the gnome sorters who sort the books in a library by repeatedly moving books to their correct position, similar to how Gnome Sort works.  The Gnome Sort algorithm works as follows:   1. Set the current position to 0. 2. If the current element is greater than the next element, swap them and move back one position. Otherwise, move forward one position. 3. Repeat steps 2 until the current position is at the end of the array. 4. If the current position is at the end of the array, the array is sorted.   Gnome Sort has a time complexity of O(n^2), where n is the number of elements to be sorted. It has a space complexity of O(1), as it only requires a constant amount of additional storage.  Gnome Sort is a stable sorting algorithm, meaning that it preserves the relative order of equal elements in the sorted output. It is also an in-place sorting algorithm, meaning that it sorts the input data array without requiring any additional storage.  If the input data array is already sorted (ascending or descending), Gnome Sort has a time complexity of O(n), as it only needs to traverse the input data array once. However, if the input data array is not sorted, Gnome Sort has a time complexity of O(n^2).  Gnome Sort is not as efficient as other sorting algorithms like Merge Sort or Quick Sort, especially for large input sizes. However, it is a simple algorithm to implement and can be useful for small input sizes or when the input data is almost sorted.  In summary, Gnome Sort is a simple, stable, and in place sorting algorithm that is not hard to implement However, it has a time complexity of O(n^2) and is not as efficient as other sorting algorithms, especially for large input sizes. Properties:  * Time complexity: O(n^2), in the worst case, where n is the number of elements in the input array. * Space complexity: O(1). * Stable: Yes. * In-place: Yes.  Running time:  * Sorted (ascending): O(n). * Sorted (descending): O(n^2). * Not sorted: O(n^2).   Advantages of Gnome Sort:   * Simple to implement and understand. * Only requires one pointer to traverse the array, making it a space-efficient sorting algorithm.   Disadvantages of Gnome Sort:   * Worst case time complexity is O(n^2), making it inefficient for large data sets. * Performs poorly on nearly sorted arrays, requiring many passes to complete the sort. * Not a stable sort, meaning that the original order of equal elements may not be preserved.   Overall, Gnome Sort is a relatively simple sorting algorithm that may be useful for small data sets or situations where memory usage is a concern. However, its poor time complexity and lack of stability make it less suitable for larger and more complex sorting tasks.   * Gnome Sort is similar to Insertion Sort in that it moves elements forward or backward in the array until they are in their correct position. However, Gnome Sort only compares adjacent elements while Insertion Sort compares elements in both forward and backward directions. * Gnome Sort gets its name from the way it moves elements up or down the array like a garden gnome moving along a garden path. * While Gnome Sort is not as fast as some other sorting algorithms, it can be a good choice for small to medium-sized data sets or in situations where simplicity and low memory usage are more important than speed. * Gnome Sort can be modified to become a stable sort by adding a check to see if an element swap occurred in the previous iteration, and then only moving back one position if a swap did not occur. This modified version is called the "Stupid Gnome Sort."   This implementation uses a while loop to iterate over the input array and swap adjacent elements that are out of order, effectively moving each out-of-order element back to its correct position in the sorted portion of the array. The **i** variable keeps track of the current position in the array, and is incremented each time two adjacent elements are in order. If two adjacent elements are out of order, they are swapped and **i** is decremented to compare the newly swapped element with the one before it. Cocktail Sort Cocktail Sort, also known as Bidirectional Bubble Sort, is a variation of Bubble Sort that sorts the input data array bidirectionally, moving both the smallest elements to the beginning and the largest elements to the end. This algorithm works by repeatedly traversing the input data array from both ends and swapping adjacent elements if they are in the wrong order.  Cocktail Sort is a stable sorting algorithm, meaning that it preserves the relative order of equal elements in the sorted output. It is also an in-place sorting algorithm, meaning that it sorts the input data array without requiring any additional storage.  The time complexity of Cocktail Sort is O(n^2), where n is the number of elements to be sorted. This algorithm's worst-case time complexity occurs when the input data array is reverse sorted, and the best-case time complexity occurs when the input data array is already sorted. In terms of space complexity, Cocktail Sort has a constant space complexity, as it does not require any additional storage beyond the input data array.  If the input data array is already sorted (ascending or descending), Cocktail Sort has a time complexity of O(n), as it only needs to traverse the input data array once to check if any elements are out of order. If the input data array is not sorted, Cocktail Sort has a time complexity of O(n^2), making it inefficient for large input sizes.  In summary, Cocktail Sort is a variation of Bubble Sort that sorts the input data array bidirectionally. It is stable, in-place, and has a constant space complexity. However, it has a worst-case time complexity of O(n^2), making it inefficient for large input sizes. Algorithm:  1. Initialize the start and end indices of the input data array as start = 0 and end = n-1, where n is the number of elements in the input data array. 2. While start is less than end, do the following: a. Traverse the input data array from start to end-1, and if the current element is greater than the next element, swap them. b. Decrement end by 1. c. Traverse the input data array from end to start+1, and if the current element is smaller than the previous element, swap them. d. Increment start by 1. 3. Repeat step 2 until start is no longer less than end. 4. The input data array is now sorted in ascending order.   **Example Input:** [6, 2, 9, 1, 5, 3]  **Example Output:** [1, 2, 3, 5, 6, 9]  In the example input, the input data array is [6, 2, 9, 1, 5, 3]. After applying Cocktail Sort, the input data array is sorted in ascending order, resulting in the output [1, 2, 3, 5, 6, 9].  Properties**:**   * Time Complexity: O(n^2), where n is the number of elements to be sorted * Space Complexity: O(1) * Stability: Stable * In-place: Yes  Running time:  * Best-case time complexity: O(n) - when the input data array is already sorted in ascending order * Worst-case time complexity: O(n^2) - when the input data array is reverse sorted * Average-case time complexity: O(n^2) * Cocktail Sort is a relatively slow sorting algorithm and can become inefficient for large input sizes.   Cocktail Sort is an in-place, stable sorting algorithm with a worst-case time complexity of O(n^2). It has a constant space complexity, making it suitable for sorting large arrays. However, its performance is not as good as other sorting algorithms, such as Merge Sort and Quick Sort, which have better average-case and worst-case time complexities. Advantages:  * Cocktail Sort is a stable sorting algorithm, meaning that it preserves the relative order of equal elements in the sorted output. * It is an in-place sorting algorithm, meaning that it does not require any additional storage beyond the input data array. * Cocktail Sort is easy to implement and understand, making it a good choice for small input sizes. * It is suitable for sorting partially sorted input data arrays, as it can take advantage of any existing order in the input data array.  Disadvantages:  * Cocktail Sort has a worst-case time complexity of O(n^2), making it inefficient for large input sizes. This can be a significant drawback if the input data array is very large. * It is not as efficient as other sorting algorithms, such as Merge Sort and Quick Sort, which have better average-case and worst-case time complexities. * Cocktail Sort is a relatively slow sorting algorithm and may not be suitable for time-critical applications.  Summary  | **Algorithm** | **Time Complexity** | **Space Complexity** | **Stability** | **In-Place?** | **Suitable for** | | --- | --- | --- | --- | --- | --- | | Counting Sort | O(n + k) | O(k) | Stable | Yes | Small input data arrays, non-negative integer values | | Bucket Sort | O(n + k) | O(n + k) | Stable | Yes | Small to medium-sized input data arrays, uniformly distributed input data | | Comb Sort | O(n^2) | O(1) | Unstable | Yes | Large input data arrays, partially sorted input data | | Gnome Sort | O(n^2) | O(1) | Stable | Yes | Small to medium-sized input data arrays, partially sorted input data | | Cocktail Sort | O(n^2) | O(1) | Stable | Yes | Small to medium-sized input data arrays, partially sorted input data |   As we can see, all of the sorting algorithms have different time and space complexities, stabilities, and in-place properties, making them suitable for different types of input data arrays and applications.  Counting Sort and Bucket Sort are both linear time sorting algorithms that are suitable for small to medium-sized input data arrays with specific properties. Counting Sort is efficient for sorting small input data arrays with non-negative integer values, while Bucket Sort is efficient for sorting small to medium-sized input data arrays with uniformly distributed input data.  Comb Sort, Gnome Sort, and Cocktail Sort are all comparison-based sorting algorithms that have worst-case time complexities of O(n^2). They are suitable for partially sorted input data arrays and can take advantage of any existing order in the input data. Comb Sort is an efficient in-place sorting algorithm suitable for large input data arrays, while Gnome Sort and Cocktail Sort are stable in-place sorting algorithms suitable for small to medium-sized input data arrays.  Overall, the choice of sorting algorithm depends on the properties of the input data array and the specific requirements of the application. It is important to consider the time and space complexity, stability, and in-place properties of each sorting algorithm to select the most appropriate one for a given task. Reference  1. GeeksforGeeks. (2021). Counting Sort. Retrieved from <https://www.geeksforgeeks.org/counting-sort/> 2. GeeksforGeeks. (2021). Bucket Sort. Retrieved from <https://www.geeksforgeeks.org/bucket-sort-2/> 3. GeeksforGeeks. (2021). Comb Sort. Retrieved from <https://www.geeksforgeeks.org/comb-sort/> 4. GeeksforGeeks. (2021). Cocktail Sort. Retrieved from <https://www.geeksforgeeks.org/cocktail-sort/> 5. Rosetta Code. (2021). Sorting algorithms/Gnome sort. Retrieved from <https://rosettacode.org/wiki/Sorting_algorithms/Gnome_sort> 6. Tutorialspoint. (2021). Counting Sort. Retrieved from <https://www.tutorialspoint.com/data_structures_algorithms/counting_sort_algorithm.htm> |

2/14/2023