**A Comparative Analysis of Merge Sort and Counting Sort**

Soscho Samuel (2003005) Nafis Raihan (2003005)

CSE, RUET CSE, RUET

**Abstract:** In this research work, Merge Sort and Counting Sort, two well-known sorting algorithms, are thoroughly compared. In computer science, sorting algorithms are essential because they affect how well data is manipulated in a variety of applications. The paper explores the fundamentals, benefits, and drawbacks of each algorithm, providing insight into how well each performs. Through an analysis of the time and space complexity, applicability, and use cases of each, this study seeks to give readers important insights into the advantages and disadvantages of Merge Sort and Counting Sort. The results enhance our comprehension of the optimal conditions for each algorithm, assisting researchers and practitioners in selecting the best option for sorting tasks optimization across various computational contexts.

**Introduction:**

**Merge Sort:** Divide and conquer problem solving is a natural strategy. When it comes to sorting, we could think about dividing the list into smaller parts, processing each part separately, and then finding a way to put them back together. The list can be divided in half, the halves sorted, and the sorted halves merged as a straightforward method of doing this. This is Merge Sort’s concept. its Time Complexity is O(n\*log n).

Advantages:

* Having Time complexity of O(n\*log n).
* It used both internal and external sorting.
* A Stable sort algorithm.

Disadvantages:

* As a minimum, the memory necessities of the further sorts since it is recursive.
* Merge sort required high space complexity.

Applications of Merge Sort:

* Sorting Big Datasets: Because Merge Sort guarantees a worst-case time complexity of O(n log n), it is especially well-suited for sorting huge datasets.
* External sorting: When the amount of data to be sorted is too big to fit into memory, merge sort is frequently utilized in external sorting.
* Custom sorting: Data that has been partially, or totally unsorted can be managed via merge sort, which can be modified to handle various input distributions.

**Counting Sort:**

[4]A non-comparison-based sorting method that performs well in situations where the input value range is constrained is counting sort. When there is a tiny difference between the number of elements to be sorted and the range of input values, it is especially effective. The fundamental principle of Counting Sort is to sort the elements in the input array according to their proper locations by counting the frequency of each distinct member. Complexity of Counting Sort is where N and M is the size of input list or array and counting array.

Advantage of Counting Sort:

* Counting sort generally performs faster than all comparison-based sorting algorithms, such as merge sort if the range of input is of the order of the number of inputs.
* Counting sort is easy to code.
* Counting sort is a stable algorithm.

Disadvantage of Counting Sort:

* Counting sort is inefficient if the range of values to be sorted is very large.
* Counting sort is not an In-place sorting algorithm, It uses extra space for sorting the array elements.

Applications of Counting Sort:

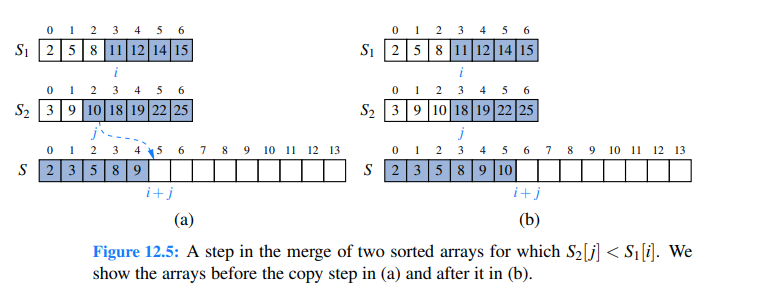
* Radix Sort Enhancement: Integral in Radix Sort for sorting integers with multiple digits.
* Data Preprocessing: Used in tasks like preparing data for analysis or visualization.
* Parallel Processing: Can be effectively parallelized due to its simplicity.
* Sorting in Linear Time: Excels in scenarios where linear time complexity is crucial.
* Educational Purposes: Utilized in educational settings to illustrate non-comparative.

**Background Study:**

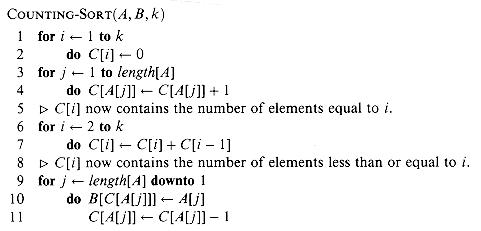
**A screenshot of a computer code

Description automatically generatedA white background with black text

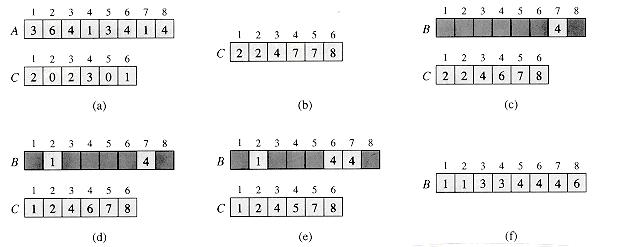
Description automatically generatedMerge Sort Algorithm:**

**Breakdown:**

We begin by focusing on the case when a sequence of items is represented with an array. The merge method (Code Fragment 12.1) is responsible for the subtask of merging two previously sorted sequences, S1 and S2, with the output copied into S. We copy one element during each pass of the while loop, conditionally determining whether the next element should be taken from S1 or S2. The divide-and-conquer merge-sort algorithm is given in Code Fragment 12.2. We illustrate a step of the merge process in Figure 12.5. During the process, index i represents the number of elements of S1 that have been copied to S, while index j represents the number of elements of S2 that have been copied to S. Assuming S1 and S2 both have at least one uncopied element, we copy the smaller of the two elements being considered. Since i + j objects have been previously copied, the next element is placed in S[i+ j]. (For example, when i+ j is 0, the next element is copied to S[0]). If we reach the end of one of the sequences, we must copy the next element from the other.

**Counting Sort Algorithm:** 

Implementation:



In the code for counting sort, we assume that the input is an array A[1 . . n], and thus length[A] = n. We require two other arrays: the array B[1 . . n] holds the sorted output, and the array C[1 . . k] provides temporary working storage.

Counting sort is illustrated in Figure 9.2. After the initialization in lines 1-2, we inspect each input element in lines 3-4. If the value of an input element is *i*, we increment *C*[*i*]. Thus, after lines 3-4, *C*[*i*] holds the number of input elements equal to *i* for each integer *i* = 1, 2, . . . , *k*. In lines 6-7, we determine for each *i*= 1, 2, . . . , *k*, how many input elements are less than or equal to *i*; this is done by keeping a running sum of the array *C*.

Finally, in lines 9-11, we place each element *A*[*j*] in its correct sorted position in the output array *B*. If all *n* elements are distinct, then when we first enter line 9, for each *A*[*j*], the value *C*[*A*[*j*]] is the correct final position of *A*[*j*] in the output array, since there are *C*[*A*[*j*]] elements less than or equal to *A*[*j*]. Because the elements might not be distinct, we decrement *C*[*A*[*j*]] each time we place a value*A*[*j*]into the B array; thiscauses the next input element with a value equal to*A*[*j*], if one exists, togo to the position immediately before*A*[*j*] in the output array.

**Results and Analysis:**

Time Taken by these algorithms in nanoseconds :

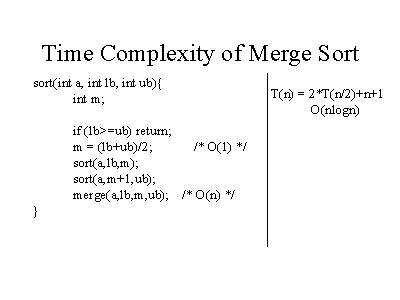
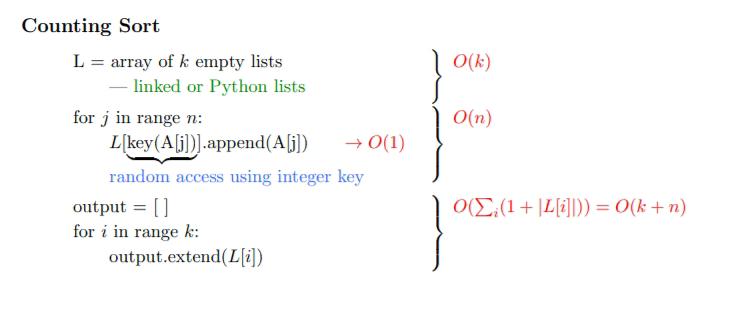
|  |  |  |
| --- | --- | --- |
| Amount of Data | Merge Sort | Counting Sort |
| 10 | 22769 | 134538 |
| 100 | 504000 | 116000 |
| 1000 | 2941077 | 196693 |
| 10000 | 35461770 | 961000 |
| 100000 | 393788231 | 8678924 |

Graphs:

Merge Sort

The Graph of merge sort looks like and Counting Sort .

Counting Sort:

Complexity Analysis:

**Conclusion:**

Counting Sort is particularly efficient for sorting integers with a limited and known range. Its linear time complexity makes it well-suited for scenarios where the range of input values is relatively small, outperforming comparison-based algorithms like Merge Sort in such cases. However, Counting Sort is specialized and may not be as effective when dealing with a broader range of data or non-integer inputs.

Merge Sort, on the other hand, is a versatile comparison-based algorithm with a stable O(n log n) time complexity. It excels in general-purpose sorting tasks and is not limited by the specific characteristics of the input data, making it a reliable choice for diverse sorting scenarios.

Ultimately, the choice between Counting Sort and Merge Sort depends on the nature and range of the data being sorted.

References:

1. Google Images.
2. GeekforGeeks.
3. Data Structures and Algorithms in Java Sixth Edition Michael T. Goodrich, Michael H. Goldwasser, Roberto Tamassia.
4. SimpliLearn.
5. Introduction to Algorithms By Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest.