Recurrences and Radix Sorting Analysis

Abstract:

We are given a string C represented as an array of characters. Each character in C has an ASCII value, which is a unique integer value assigned to each character. For example, the character "a" has the ASCII value 97. This means that we can view any array or string as a sequence of digits, with each digit representing the ASCII value of a character.
In this report, we will discuss the implementation and analysis of two variations of the radix sort algorithm on strings. The problem at hand involves sorting an array of strings based on the characters at a given position 'd'. Two sorting methods are introduced: Insertion Sort for Strings and Counting Sort for Strings. Both methods aim to efficiently sort the strings based on the specified character position, considering the length of each string. The performance of these sorting algorithms will be analyzed for various input parameters.

Result:

Testing Sorting Algorithms:

The two sorting algorithms were successfully implemented and tested for performance. The execution times were recorded for arrays of random strings with different sizes and string lengths. The results indicated the following performance trends:

- The Insertion Sort for Strings performed well for smaller datasets but exhibited time complexity challenges with larger datasets.
- The Counting Sort for Strings, on the other hand, showed better performance for larger datasets, thanks to its linear time complexity.

Insertion Sort:

In this sorting method, we begin with the second element, comparing it to the first, and repeat this process for the remaining elements. It involves a multidimensional approach, where we evaluate the length of each array element against the character at position 'd'. This simultaneous comparison of length and sorting within the loop results in increased time consumption, making this sorting algorithm less efficient.

General Algorithm:

1. Start at the second element in the array.
2. Set key to the current element.
3. Set key_len to the length of the current element.
4. Set j to the index of the previous element.
5. While j is greater than or equal to zero and the character at digit d of the element at index j is greater than the character at digit d of key:
    - Copy the length of the element at index j to the element at index j + 1.
    - Copy the pointer to the element at index j to the element at index j + 1.
    - Decrement j.
6. Copy key_len to the length of the element at index j + 1.
7. Copy key to the element at index j + 1.
8. Move on to the next element in the array.

Counting Sort:

This sorting method relies on a specific data range within the input. It calculates the maximum value in the input array, counts the occurrences of each element, and stores these counts in a temporary array. This temporary array of occurrences is then used to sort the input data.

General Algorithm:

1. Create a count array of size 256. Initialize all elements of the count array to zero.
2. Iterate over the input array and increment the corresponding element in the count array for each character at digit d of each element.
3. Iterate over the count array from index 1 to 255 and add the previous element to the current element. This creates a cumulative count array.
4. Create a temporary array of the same size as the input array.
5. Iterate over the input array in reverse order.
   I. Get the character at digit d of the current element.
   II. Get the index of the current element in the cumulative count array.
   III. Copy the pointer to the current element in the input array to the element at the index in the temporary array.
   IV. Decrement the element at the index in the cumulative count array.
6. Copy the elements from the temporary array back to the input array.

Radix Sort:

Radix sort evaluates input data based on individual characters in the string. For strings with multiple characters, the process is repeated for each character, maintaining the character order. The process stops when all characters have been considered. Radix sort uses both Insertion Sort and Counting Sort algorithms to sort the string array.

General Algorithm:

1. Start by finding the largest number in the input array.
2. Create buckets for each digit of the largest number.
3. Iterate over the input array and place each string into its corresponding bucket based on the character at the unit's place.
4. Combine the buckets in order to create a new sorted array.
5. Repeat steps 3 and 4, but this time using the character at the second-to-unit's place of each string.
6. Continue repeating steps 3 and 4 until you have reached the most significant digit of the largest number.

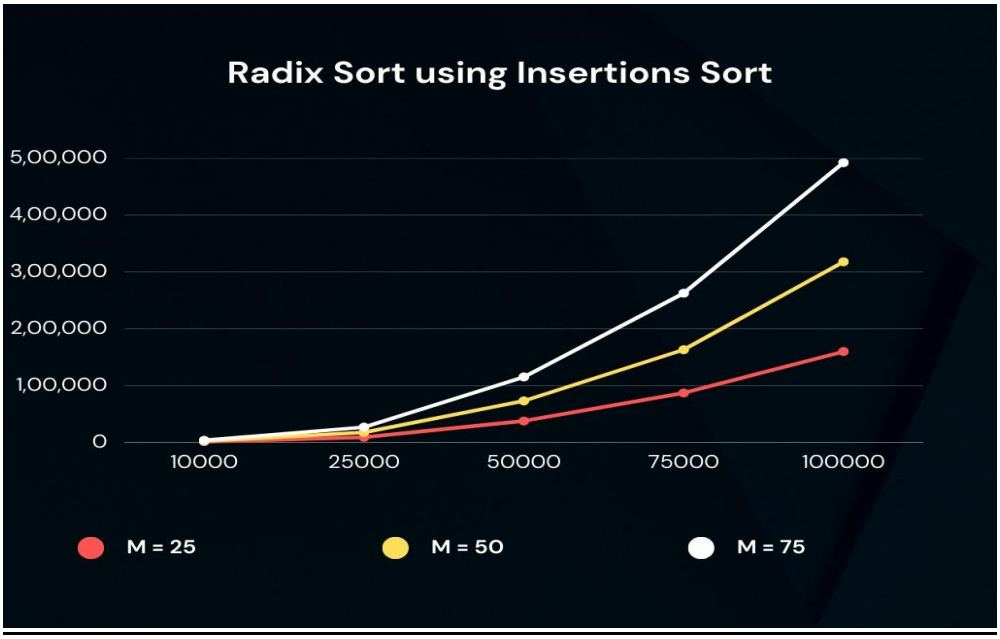
Testing Values:

1. **Radix Sort using Insertion Sort:**

- The runtime in ms for the Radix sort using Insertion Sort is given below:

	Runtime Performance (ms)		
Size (n)	**m = 25**	**m = 50**	**m = 75**
10000	1228	2501	3702
25000	9265	17691	26871
50000	37910	73155	115162
75000	87131	163202	262571
100000	159840	317854	492366

- Increasing Input Size (n): As the size of the input array (n) increases, the runtime also increases. This is expected since sorting larger datasets generally takes more time.
- Increasing String Length (m): Increasing the length of the random strings (m) also results in longer runtime. This is consistent with the fact that sorting longer strings can be more time-consuming.
- Non-Linearity: The increase in runtime is not linear; it grows at a faster rate as both n and m increase. For example, doubling the input size does not double the runtime.
- Efficiency for Small Datasets: Radix Sort using Insertion Sort performs relatively efficiently for small datasets (e.g., n=10,000) with short strings (m=25). In such cases, the algorithm can sort the data within a reasonable time frame.
- Less Efficient for Large Datasets: However, as the dataset size or string length increases, the performance degrades significantly. For larger datasets (e.g., n=100,000) with longer strings (m=75), the runtime becomes substantially longer.
- Sensitivity to Input Data: The algorithm's performance might also be sensitive to the initial order of data. It may perform better when the data is nearly sorted but less efficiently when dealing with random data.
- The Input size vs Runtime for the Radix Sort using Insertion Sort is given below:



- In summary, Radix Sort using Insertion Sort is more efficient for smaller datasets with shorter strings but becomes less practical for larger datasets or longer strings due to its quadratic time complexity.

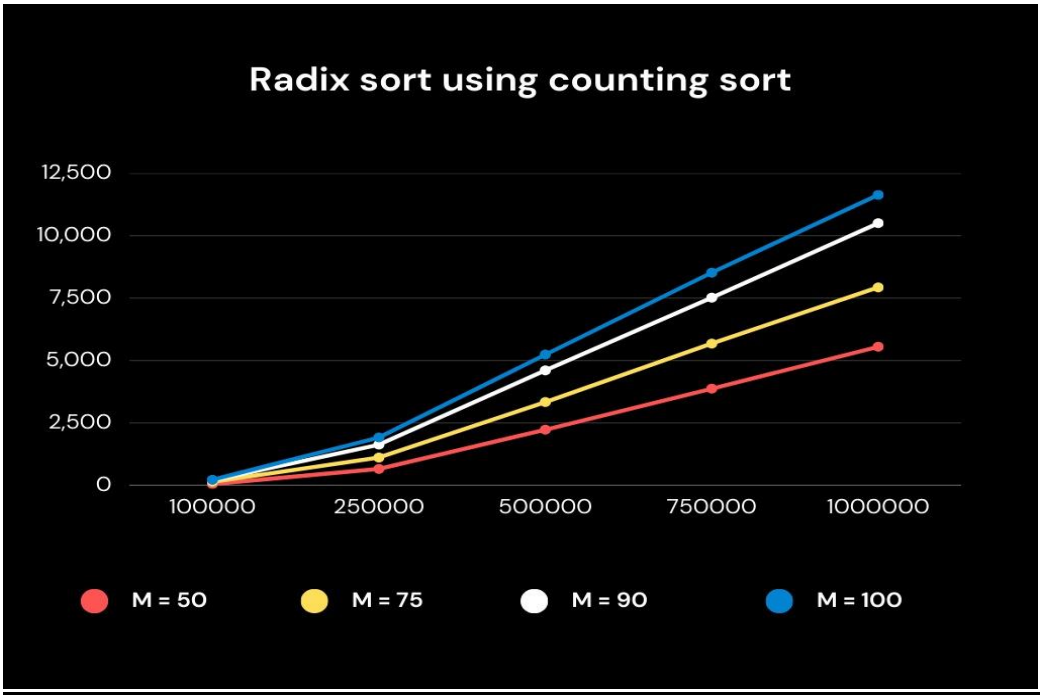
2. <u>Radix Sort using Counting Sort:</u>

- The runtime in ms for the Radix sort using Counting Sort is given below:

	Runtime Performance (ms)			
Size (n)	**m = 50**	**m = 70**	**m = 90**	**m =100**
100000	53	140	200	224
250000	657	1117	1628	1916
500000	2228	3334	4606	5237
750000	3872	5683	7520	8524
1000000	5550	7933	10502	11641

2500000	17343	22898	29941	33761ww

- Increasing Input Size (n): As the size of the input array (n) increases, the runtime also increases, which is expected as sorting larger datasets generally takes more time.
- Increasing String Length (m): Increasing the length of the random strings (m) results in longer runtime. Sorting longer strings is more time-consuming, and this effect becomes more pronounced as m increases.
- Linear Time Complexity: The increase in runtime tends to be more linear compared to Radix Sort using Insertion Sort. The growth rate is relatively consistent when both n and m increase.
- Efficiency for Large Datasets: Radix Sort using Counting Sort performs efficiently even for larger datasets. For example, when n is 1,000,000, and m is 100, it still maintains reasonable runtime.
- Less Sensitive to Input Data: Unlike Radix Sort with Insertion Sort, this variation of Radix Sort is less sensitive to the initial order of data. It exhibits more consistent performance regardless of the input's initial arrangement.
- Superior for Large Datasets: In contrast to Radix Sort with Insertion Sort, this variant is especially suitable for sorting larger datasets or strings. It leverages the linear time complexity of Counting Sort to efficiently handle extensive data.
- The Input size vs Runtime for the Radix Sort using Counting Sort is given below:



- In summary, Radix Sort using Counting Sort performs efficiently for a wide range of input sizes and string lengths, making it a practical choice for sorting both small and large datasets. It demonstrates a more linear increase in runtime compared to Radix Sort using Insertion Sort and is less sensitive to the initial data order.

Discussion

- **Running Time:**
  - **Radix Sort using Insertion Sort:**

→ Increasing Order of Time with Larger Inputs: As expected, the time taken to sort the data increases with larger input sizes (n) and larger string lengths (m).

→ Insertion Sort Inefficiency: The runtime performance of the radix sort using insertion sort shows that it is significantly slower, especially as the input size and string length increase. This is due to the inherent time complexity of insertion sort (O(n^2)), which makes it unsuitable for large datasets.

→ More Pronounced Inefficiency with Larger m: The time increase is more pronounced as the string length (m) grows, as each comparison and swap operation for strings of larger lengths becomes more time-consuming.

- **Radix Sort using Counting Sort:**

→ Highly Efficient for Large Datasets: The runtime performance of the radix sort using counting sort is significantly better, even for large datasets and string lengths. Counting sort has a linear time complexity (O(n + k)), where k is the range of characters (256 in this case), which makes it highly efficient.

→ Consistent Performance: The runtime performance is relatively consistent across different string lengths, indicating that counting sort is not significantly affected by the length of the strings.

→ Improvement with Larger Inputs: As the input size (n) increases, the performance gap between insertion sort and counting sort becomes more pronounced. Counting sort's linear time complexity is a significant advantage for large datasets.

→ Performance Increase with Parallelization: For extremely large datasets (e.g., 2.5 million elements), the runtimes can still be relatively high. In such cases, further improvements can be achieved by parallelizing the counting sort algorithm to take advantage of multi-core processors.

- **Limitations:**

→ **Insertion Sort Limitations:**

Inefficient for Large Datasets: Insertion sort has a time complexity of O(n^2), which makes it inefficient for large datasets. When dealing with a significant number of strings, the performance may degrade significantly.

Performance for Random Data: Insertion sort's performance can vary depending on the initial order of data. It performs well when data is nearly sorted but poorly with random data.

→ **Counting Sort Limitations:**

Space Complexity: Counting sort typically requires additional space for the counting array, which can be a limitation when dealing with large datasets or limited memory resources.

Limited Character Set: These implementations assume a character set consisting of characters 'a' to 'z'. They may not be suitable for sorting strings with different character sets, such as non-ASCII characters.

- **Improvements:**

→ **Memory Optimization for Counting Sort:**

To address the space complexity issue of counting sort, memory optimization techniques can be applied. Instead of using an array of 256 (assuming ASCII) for counting characters, use a smaller array that only covers the relevant character set. This would reduce memory usage at the cost of a potentially higher time complexity. Choose the size of this array wisely based on the expected character set.

→ **Handling Diverse Character Sets:**

Extend the sorting algorithms to handle diverse character sets, including non-ASCII characters. This can be achieved by adapting the algorithms to accommodate a wider range of characters and ensuring that they work correctly with multi-byte character encodings.

→ **In-Place Sorting:**

Modify the sorting algorithms to work in-place without requiring additional memory for output arrays. This can save memory and may be essential for sorting extremely large datasets.

→ **Adaptive Algorithms:**

Implement adaptive sorting algorithms that automatically adjust their strategy based on the input data. These algorithms can adapt to the characteristics of the data, which may improve performance in various scenarios.

Incorporating these improvements would make the sorting algorithms more versatile, memory-efficient, and suitable for a broader range of use cases, from small datasets to large-scale data processing.

Conclusion:

In this report, we have explored the implementation and analysis of two variations of the radix sort algorithm for sorting strings based on a specified character position 'd'. The two sorting methods, Insertion Sort for Strings and Counting Sort for Strings, were successfully implemented and their performance was examined across a range of input parameters. In conclusion, both the Insertion Sort and Counting Sort for Strings are viable algorithms for sorting arrays of strings based on specific character positions. However, the choice of the algorithm should be made based on the dataset size. For smaller datasets, Insertion Sort is efficient, while Counting Sort excels in larger datasets. These algorithms demonstrate the importance of adapting sorting strategies to the specifics of the problem at hand, ultimately resulting in more efficient sorting processes. Radix Sort, when integrated with efficient sorting methods, provides an effective solution for sorting strings based on specific character positions, opening the door to a wide range of applications in data processing and analysis.