CS221 Final Project: Rosdav Algorithm

Tanedo, Roseler T. Jr., Buenaventura, John David C.

BatStateU - Alangilan

May 19, 2023

Introduction

Rosdav algorithm is a unique sorting algorithm designed to sort an array of integers. It combines the principles of divide and conquer with a distinctive merging method to achieve its sorting objective.

The algorithm operates by recursively dividing the input array into smaller subarrays until the base case is reached, which is when the subarrays contain only one element or are empty. Then, it merges the sorted subarrays in a unique manner. The merging process ensures that the second half of the array correctly based on whether the length of the final sorted array is even or odd.

Rosdav's balanced splitting strategy helps distribute the workload evenly between recursive calls, contributing to its efficiency on certain types of input data. While its time complexity is comparable to other efficient sorting algorithms.

Problem Statement

The problem statement of Rosdav's algorithm is to sort an input array in ascending order. The algorithm follows a divide-and-conquer approach, recursively splitting the input array into smaller subarrays, sorting them, and then merging the sorted subarrays to obtain the final sorted result. It also includes additional logic to handle odd-sized arrays by splitting the merged result into two halves and sorting them independently. The goal is to efficiently and correctly sort the input array, ensuring that the resulting array is in ascending order.

Algorithm Design

The algorithm design for the Rosdav sorting algorithm involves the following steps:

- Define the base case: If the length of the input array ‘arr‘ is less than or equal to 1, return the array as it is already sorted.
- Find the midpoint: Calculate the midpoint index, ‘mid‘, of the array ‘arr‘ by performing integer division of the length of ‘arr‘ by 2.
- Recursively sort the two subarrays: Split the array ‘arr‘ into two subarrays: ‘left‘ consisting of elements from index 0 to ‘mid-1‘, and ‘right‘ consisting of elements from index ‘mid‘ onwards. Recursively apply Rosdav sort on ‘left‘ and ‘right‘ to obtain sorted subarrays.
- Merge the two sorted subarrays: Create an empty list, ‘result‘, to store the merged subarrays. Initialize three pointers: ‘i‘ to traverse ‘left‘, ‘j‘ to traverse ‘right‘, and ‘k‘ to keep track of the current index in ‘result‘.

Algorithm Design

- Compare and merge elements: Iterate while both ‘i‘ and ‘j‘ are within their respective subarrays. Compare the elements at ‘left[i]‘ and ‘right[j]‘. If ‘left[i]‘ is smaller, append it to ‘result‘ and increment ‘i‘. Otherwise, append ‘right[j]‘ to ‘result‘ and increment ‘j‘. Also, increment ‘k‘.
- Handle remaining elements: If there are any remaining elements in either ‘left‘ or ‘right‘, append them to ‘result‘ and update ‘k‘ accordingly.
- Sort the second half of ‘result‘: Check if the length of ‘result‘ is even or odd. If it is even, obtain the index ‘mid_idx‘ by performing integer division of the length of ‘result‘ by 2. Sort the elements from index ‘mid_idx‘ onwards in ascending order using the ‘sorted()‘ function. If it is odd, sort the elements from index ‘mid_idx‘ + 1 onwards in ascending order.
- Return the final sorted list: Return the ‘result‘ list as the final sorted list obtained using the Rosdav algorithm.

Implementation

Rosdav algorithm is created using the programming language Python, which is one of the easiest programming language to utilize to create algorithms. Here are the snippets of the code made.

Recursive Call on the left and right half

```
n = len(arr)

if n <= 1:
    return arr
else:
    mid = n // 2
    left = rosdav(arr[:mid])
    right = rosdav(arr[mid:])
    i, j, k = 0, 0, 0
    result = []
```

Implementation

Merging of the two subaarays

```
i, j, k = 0, 0, 0
   result = []
   while i < len(left) and j < len(right):
       if left[i] < right[j]:
           result.append(left[i])
           i += 1
       else:
           result.append(right[j])
           j += 1
       k += 1

   while i < len(left):
       result.append(left[i])
       i += 1
       k += 1

   while j < len(right):
       result.append(right[j])
       j += 1
       k += 1
```

Implementation

Sorting of the merged result array

```
if len(result) % 2 == 0:
        mid_idx = len(result) // 2
        return result[:mid_idx] + sorted(result[mid_idx:])
   else:
        mid_idx = len(result) // 2
        return result[:mid_idx+1] + sorted(result[mid_idx+1:])
```

Analysis

Time Complexity:

- The Rosdav algorithm has a time complexity of $O(nlogn)$, where n is the length of the input array.
- The recursion in Rosdav involves dividing the array into two subarrays, each with approximately half the size of the original array. This process takes $O(logn)$ time due to the recursive calls.
- The merging step, where the two sorted subarrays are combined, takes linear time proportional to the length of the subarrays being merged. Since each element of the input array is involved in exactly one merge operation, the overall merging time complexity is $O(n)$.
- Therefore, the total time complexity of the Rosdav algorithm is $O(nlogn)$.

Analysis

Space Complexity:

- The Rosdav algorithm has a space complexity of $O(n)$, where n is the length of the input array.
- In the recursive calls, the algorithm creates new stack frames for each level of recursion. The maximum depth of the recursion tree is log n, which contributes to the space complexity of $O(nlogn)$.
- The merging process in Rosdav does not require any additional space beyond the input array, as the merging is done in place. Hence, the merging step does not add to the space complexity.
- Overall, the space complexity of the Rosdav algorithm is $O(n)$.

Analysis

Performance and Comparisons:

- The algorithm efficiently balances the workload between the recursive calls by splitting the array into two subarrays with approximately equal sizes. This balanced splitting contributes to better performance on certain inputs compared to other sorting algorithms like merge sort, which may result in unbalanced subarrays.
- However, in terms of average-case time complexity, Rosdav has the same time complexity as quicksort and mergesort, which are widely used and efficient sorting algorithms.
- When compared to less efficient sorting algorithms like bubble sort or insertion sort, Rosdav generally performs better due to its time complexity of $O(nlogn)$.

Analysis

Performance and Comparisons:

- In Rosdav's algorithm, the merge step compares elements from the left and right subarrays. However, instead of using explicit pointers, it uses three counters: i, j, and k. It iterates over the elements of the subarrays and appends the smaller element to the result list. The counters i and j keep track of the current positions in the left and right subarrays, while k keeps track of the position in the result list.
- In this sorting algorithm, the remaining elements are appended to the result list with additional sorting. The list is split into two halves, and the second half is sorted separately before concatenating the two halves. This additional sorting step is required to maintain the specific ordering property of Rosdav's algorithm, where the last element of the array is guaranteed to be the largest.

Conclusion

The project implements Rosdav's algorithm, a recursive sorting algorithm that divides an input array into smaller subarrays, sorts them, and then merges them to produce a sorted output array in it's own way.The provided implementation recursively applies the algorithm, merging the subarrays and handling remaining elements appropriately.

Conclusion

Possible work or Improvements

- **Input Validation:** Add input validation to handle different scenarios, such as checking for ascending order, even length, and repeated elements.
- **Performance Optimization:** Analyze and optimize the algorithm for better time and space efficiency.
- **Algorithm Generalization:** Modify the algorithm to handle odd lengths, repetitions, or non-ascending order.
- **Comparative Analysis:** Compare Rosdav's algorithm with other sorting algorithms to evaluate their strengths and weaknesses.
- **Code Refactoring and Documentation:** Improve code readability, maintainability, and provide comprehensive documentation.
- **Testing and Validation:** Develop thorough test cases and automated tests to validate correctness and performance.