Suppose Mergesort Routine Is Applied On The Following Input Arrays: 5,3,8,9, 1,7,0,2,6,4. Fast Forward

Suppose Mergesort Routine Is Applied On The Following Input Arrays: 5,3,8,9, 1,7,0,2,6,4. Fast Forward

---

Introduction to Mergesort Algorithm

Mergesort is a highly efficient, comparison-based, divide-and-conquer sorting algorithm. It was invented by John von Neumann in 1945 and is renowned for its predictable performance and stability. Unlike simpler algorithms like Bubble Sort or Selection Sort, mergesort guarantees a time complexity of O(n log n) in the worst case, making it suitable for large datasets.

This algorithm works by recursively dividing the input array into smaller subarrays until each subarray contains a single element. Then, it merges these subarrays in a manner that results in a sorted array. The process involves two main phases: divide and conquer, followed by the merge step.

Understanding the Input Array

Given the input array:

`5, 3, 8, 9, 1, 7, 0, 2, 6, 4`

This array contains 10 elements, which makes it an ideal candidate for demonstrating the step-by-step operation of mergesort. The goal is to sort this array in ascending order using the mergesort routine.

Step-by-Step Mergesort Process

Initial Division of the Array

The first step involves dividing the array into two halves:

- Left subarray: `5, 3, 8, 9, 1`
- Right subarray: `7, 0, 2, 6, 4`

Each of these subarrays will then be recursively subdivided until each contains only a single element.

Recursive Division of the Left Subarray

Left Subarray: `5, 3, 8, 9, 1`

1. Divide into two parts:
- Left: `5, 3, 8`
- Right: `9, 1`

2. Further divide `5, 3, 8`:
- Left: `5, 3`
- Right: `8`

3. Divide `5, 3`:
- Left: `5`
- Right: `3`

4. `8` is already a single element.

Similarly, the right part `9, 1` divides into:
- Left: `9`
- Right: `1`

Now, at this point, all subarrays are reduced to single elements:
`5`, `3`, `8`, `9`, `1`

Recursive Division of the Right Subarray

Right Subarray: `7, 0, 2, 6, 4`

1. Divide into:
- Left: `7, 0, 2`
- Right: `6, 4`

2. Divide `7, 0, 2`:
- Left: `7, 0`
- Right: `2`

3. Divide `7, 0`:
- Left: `7`
- Right: `0`

4. `2`, `6`, and `4` are already single elements.

Now, all subdivided into single elements:
`7`, `0`, `2`, `6`, `4`

Merge Phase: Combining Sorted Subarrays

After successfully dividing the array into individual elements, the next phase involves merging these elements back together in sorted order.

Merging the Left Subarray

1. Merge `5` and `3`:
- Sorted result: `3, 5`

2. Merge `3, 5` with `8`:
- Compare:
- `3` vs `8` → `3`
- `5` vs `8` → `5`
- Result after merge: `3, 5, 8`

3. Merge `9` and `1`:
- Sorted result: `1, 9`

4. Merge `3, 5, 8` with `1, 9`:
- Compare `3` and `1` → `1`
- Compare `3` and `9` → `3`
- Compare `5` and `9` → `5`
- Compare `8` and `9` → `8`
- Remaining elements: `9`
- Final merged sorted left subarray: `1, 3, 5, 8, 9`

Merging the Right Subarray

1. Merge `7` and `0`:
- Sorted result: `0, 7`

2. Merge `0, 7` with `2`:
- Compare `0` and `2` → `0`

- Compare `7` and `2` → `2`
- Remaining elements: `7`
- Result: `0, 2, 7`

3. Merge `6` and `4`:
- Sorted result: `4, 6`

4. Merge `0, 2, 7` with `4, 6`:
- Compare `0` and `4` → `0`
- Compare `2` and `4` → `2`
- Compare `7` and `4` → `4`
- Compare `7` and `6` → `6`
- Remaining element: `7`
- Final merged sorted right subarray: `0, 2, 4, 6, 7`

Final Merge: Combining Both Halves

Now, merge the two sorted halves:

- Left: `1, 3, 5, 8, 9`
- Right: `0, 2, 4, 6, 7`

Merging process:

1. Compare `1` and `0` → `0`
2. Compare `1` and `2` → `1`
3. Compare `3` and `2` → `2`
4. Compare `3` and `4` → `3`
5. Compare `5` and `4` → `4`
6. Compare `5` and `6` → `5`
7. Compare `8` and `6` → `6`
8. Compare `8` and `7` → `7`
9. Remaining elements: `8`, `9`

Final sorted array:

`0, 1, 2, 3, 4, 5, 6, 7, 8, 9`

Benefits of Using Mergesort

Mergesort offers several advantages, making it a preferred sorting algorithm in various applications:

- Consistent Performance: Guarantees O(n log n) complexity regardless of input data distribution.
- Stability: Maintains the relative order of equal elements, which is essential in certain sorting contexts.
- External Sorting: Suitable for sorting large datasets that do not fit entirely into memory, as it can be implemented with external storage.
- Predictability: The recursive division and merging process makes its performance predictable, unlike algorithms like Quicksort that can degrade to O(n^2) in worst-case scenarios.

Implementation Details and Optimization Tips

Implementing mergesort efficiently requires awareness of certain best practices:

- Using Auxiliary Arrays: During merging, temporary arrays are used to hold sorted subarrays. Efficient memory management can improve performance.
- In-Place Mergesort: Though traditional mergesort requires extra space, advanced techniques can perform in-place merging to reduce memory usage.
- Hybrid Algorithms: Combining mergesort with other algorithms like insertion sort for small subarrays can optimize performance.
- Parallelization: Mergesort's divide-and-conquer nature lends itself well to parallel execution, significantly reducing sorting time on multi-core processors.

Applications of Mergesort

Mergesort is widely used across various domains owing to its efficiency and stability. Some common applications include:

- Database Sorting: Sorting large datasets stored on disk.
- External Sorting: Handling data that exceeds main memory capacity.
- Multithreaded Sorting: Parallel implementations for high-performance computing.
- Stable Sorting Needs: When the order of equal elements must be preserved, such as in multi-stage sorting processes.

Conclusion

Applying the mergesort routine to the input array `5, 3, 8, 9, 1, 7, 0, 2, 6, 4` demonstrates its systematic approach to sorting through recursive division and merging. This process ensures the array is sorted in ascending order efficiently and reliably, highlighting why mergesort remains a fundamental algorithm in computer science. Its predictable performance, stability, and adaptability

make it a versatile choice for sorting large or complex datasets across various applications.

Summary

- Mergesort is a divide-and-conquer algorithm with O(n log n) time complexity.
- It divides the array into smaller parts recursively until single elements are achieved.
- Sorted subarrays are merged in a manner that results in a fully sorted array.
- The algorithm is stable, predictable, and suitable for large datasets.
- Practical implementations can be optimized with techniques like in-place merging and parallel processing.

By

Frequently Asked Questions

What is the first step in applying Mergesort to the array [5, 3, 8, 9, 1, 7, 0, 2, 6, 4]?

The first step is to divide the array into smaller subarrays until each subarray contains only one element. So, the array is split into [5, 3, 8, 9, 1], [7, 0, 2, 6, 4].

How does Mergesort merge the subarrays during the sorting process?

Mergesort merges subarrays by comparing the elements of each subarray and combining them into a sorted array, repeatedly merging sorted subarrays until the entire array is sorted.

After the initial division, what are the sorted subarrays before the final merge in the given input?

Initially, the array is split into [5], [3], [8], [9], [1], [7], [0], [2], [6], [4], which are already sorted individually. Subsequent merging steps produce larger sorted subarrays.

What is the sorted array after applying Mergesort to [5, 3, 8, 9, 1, 7, 0, 2, 6, 4]?

The sorted array after applying Mergesort will be [0, 1, 2, 3, 4, 5, 6, 7, 8, 9].

Why is Mergesort considered an efficient sorting algorithm for

large datasets like the given array?

Mergesort has a consistent time complexity of O(n log n), making it efficient for large datasets. It also guarantees stable and predictable performance regardless of the initial order of elements.

Additional Resources

Suppose Mergesort Routine Is Applied On The Following Input Arrays: 5,3,8,9,1,7,0,2,6,4. Fast Forward

In the world of computer science, sorting algorithms serve as the backbone of efficient data processing, enabling rapid organization and retrieval of information. Among these algorithms, Mergesort stands out as a classic example of a divide-and-conquer approach, renowned for its stability and predictable performance. When applied to specific datasets, understanding how Mergesort operates not only demystifies its internal mechanics but also highlights its strengths and limitations. In this detailed exploration, we examine the step-by-step process of applying Mergesort to the input array: 5, 3, 8, 9, 1, 7, 0, 2, 6, 4. Fast forward through the process, we will analyze each phase—from dividing the array into smaller subarrays to the meticulous merging steps—providing insights into the algorithm's efficiency and behavior.

Understanding Mergesort: A Foundation

What Is Mergesort?

Mergesort is a comparison-based sorting algorithm that relies on recursively dividing an unsorted list into smaller sublists until each sublist contains a single element or is empty. Subsequently, these sublists are merged back together in a manner that results in a sorted array. Its key features include:

- Divide and Conquer Strategy: Breaking down the problem into smaller, manageable parts.
- Recursive Implementation: Applying the same process to subproblems.
- Stable Sorting: Preserving the relative order of equal elements.
- O(n log n) Time Complexity: Consistent performance regardless of input distribution.

Why Use Mergesort?

Mergesort's efficiency makes it suitable for large datasets, especially where stability is essential. Its predictable O(n log n) time complexity outperforms simpler algorithms like insertion sort or selection sort, which have quadratic complexities. Moreover, its ability to handle data stored on external storage efficiently makes it useful in systems with large files.

Step-by-Step Breakdown of Mergesort on the Input Array

The input array under consideration is:

`[5, 3, 8, 9, 1, 7, 0, 2, 6, 4]`

Applying Mergesort involves two main phases:

1. Dividing the array into smaller subarrays until each subarray contains a single element.
2. Merging subarrays back together in sorted order.

Below, we detail each phase, tracking the recursive divisions and subsequent merging.

Phase 1: Recursive Division of the Array

Initial Array:

`[5, 3, 8, 9, 1, 7, 0, 2, 6, 4]`

Step 1: Divide into two halves

- Left: `[5, 3, 8, 9, 1]`
- Right: `[7, 0, 2, 6, 4]`

Dividing the Left Subarray `[5, 3, 8, 9, 1]`

Step 2: Divide into two halves

- Left: `[5, 3]`
- Right: `[8, 9, 1]`

Step 3: Further divide `[5, 3]`

- Left: `[5]`
- Right: `[3]`

Base case reached: Single elements `[5]` and `[3]`

Step 4: Divide `[8, 9, 1]`

- Left: `[8]`
- Right: `[9, 1]`

Step 5: Divide `[9, 1]`

- Left: `[9]`
- Right: `[1]`

Base case: `[9]` and `[1]`

Dividing the Right Subarray `[7, 0, 2, 6, 4]`

Step 6: Divide into two halves

- Left: `[7, 0]`
- Right: `[2, 6, 4]`

Step 7: Divide `[7, 0]`

- Left: `[7]`
- Right: `[0]`

Base case: `[7]` and `[0]`

Step 8: Divide `[2, 6, 4]`

- Left: `[2]`
- Right: `[6, 4]`

Step 9: Divide `[6, 4]`

- Left: `[6]`
- Right: `[4]`

Base case: `[6]` and `[4]`

Phase 2: Merging Sorted Sublists

After reaching the base case of single elements, the algorithm proceeds to merge these into sorted subarrays, gradually building up to the sorted array.

Merging the Left Subarray `[5]` and `[3]`

- Compare 5 and 3
- 3 is smaller, so first element: `[3]`
- Remaining: `[5]`

Merged Result: `[3, 5]`

Merging `[8]` and `[9, 1]`

- First, merge `[9]` and `[1]`:

- Compare 9 and 1
- 1 is smaller, so `[1]`
- Remaining: `[9]`

- Now, merge `[8]` and `[1, 9]`:

- Compare 8 and 1
- 1 is smaller: `[1]`
- Next compare 8 and 9
- 8 is smaller: `[8]`
- Remaining: `[9]`

Merged Result: `[1, 8, 9]`

Merging `[5, 3]` and `[8, 9, 1]`

- Merge `[3, 5]` and `[1, 8, 9]`:

- Compare 3 and 1: 1 is smaller → `[1]`
- Compare 3 and 8: 3 is smaller → `[1, 3]`
- Compare 5 and 8: 5 is smaller → `[1, 3, 5]`
- Remaining `[8, 9]` are appended → `[1, 3, 5, 8, 9]`

Merged Left Subarray: `[1, 3, 5, 8, 9]`

Merging `[7]` and `[0]`

- Compare 7 and 0
- 0 is smaller → `[0]`
- Remaining: `[7]`

Merged Result: `[0, 7]`

Merging `[2]` and `[6, 4]`

- Merge `[6]` and `[4]`:

- Compare 6 and 4 → 4 is smaller → `[4]`
- Remaining: `[6]`

- Now, merge `[2]` and `[4, 6]`:

- Compare 2 and 4 → 2 is smaller → `[2]`
- Remaining `[4, 6]`

Merged Result: `[2, 4, 6]`

Merging `[0, 7]` and `[2, 4, 6]`

- Compare 0 and 2 → 0 is smaller → `[0]`
- Compare 7 and 2 → 2 is smaller → `[0, 2]`
- Compare 7 and 4 → 4 is smaller → `[0, 2, 4]`
- Compare 7 and 6 → 6 is smaller → `[0, 2, 4, 6]`
- Remaining: `[7]`

Merged Result: `[0, 2, 4, 6, 7]`

Final Merge: Combining Left and Right Sorted Arrays

Now, the two main sorted arrays are:

- Left: `[1, 3, 5, 8, 9]`
- Right: `[0, 2, 4, 6, 7]`

Merging `[1, 3, 5, 8, 9]` and `[0, 2, 4, 6, 7]`

- Compare 1 and 0 → 0 is smaller → `[0]`
- Compare 1 and 2 → 1 is smaller → `[0, 1]`
- Compare 3 and 2 → 2 is smaller → `[0, 1, 2]`
- Compare 3 and 4 → 3 is smaller → `[0, 1, 2, 3]`
- Compare 5 and 4 → 4 is smaller → `[0, 1, 2, 3, 4]`
- Compare 5 and 6 → 5 is smaller → `[0, 1, 2, 3, 4, 5]`
- Compare 8 and 6 → 6 is smaller → `[0, 1, 2, 3, 4, 5, 6]`
- Compare 8 and 7 → 7 is smaller → `[0, 1, 2, 3, 4, 5, 6, 7]`
- Remaining: `[8, 9]`

- Append remaining `[8,

Suppose Mergesort Routine Is Applied On The Following Input Arrays 5 3 8 9 1 7 0 2 6 4 Fast Forward

Suppose Mergesort Routine Is Applied On The Following Input Arrays 5 3 8 9 1 7 0 2 6 4 Fast Forward

Related Articles

- She Got Soaked Walking Home In The Rain Yesterday Because Her Umbrella Was In Her Gym Locker.what Part
- Suppose M,n Z1, That C, That A Is An Nn Matrix With Coefficients In C, And That Is An Eigenvalue Of A.
- True Or False, Otto Von Bismarck, The Prussian-born Leader Of German Unification,Practiced Realpolitik

Back to Home