Data structures and Algorithms

Note03-Sorting problem P2

Edited By

Arshia Gharooni

2023

0.1 Introduction

Sorting is a fundamental operation in computer science and is used extensively in various applications such as searching, data compression, and database management. There are many sorting algorithms available, and each algorithm has its own advantages and disadvantages. Two popular sorting algorithms are merge sort and insertion sort. In this lecture note, we will compare the two algorithms and analyze their performance characteristics.

0.2 Insertion Sort vs. Merge Sort: A Comparative Analysis

0.2.1 Insertion Sort

Insertion sort is a simple and efficient algorithm that sorts a list by repeatedly inserting unsorted elements into a sorted portion of the list. The algorithm works by iteratively inserting each unsorted element into its correct position within the sorted portion of the list.

Best Case Time Complexity

In the best case, the input list is already sorted. In this case, the algorithm simply iterates over the list once to confirm that the list is sorted. The best-case time complexity of insertion sort is therefore O(n).

Worst Case Time Complexity

In the worst case, the input list is in reverse order. In this case, each unsorted element must be compared to every element in the sorted portion of the list, resulting in a time complexity of O(n^2).

Space Complexity

Insertion sort requires only a constant amount of extra memory to sort an input list. Specifically, it uses a single variable to hold a temporary copy of the current element being inserted into the sorted portion of the list.

Therefore, the space complexity of insertion sort is O(1), which means that the amount of extra memory required by the algorithm does not depend on the size of the input list.

Optimality

Insertion sort is not an optimal sorting algorithm in terms of worst-case time complexity. However, it can be useful in certain situations where the input is almost sorted or when the size of the input is small.

0.2.2 Merge Sort

Merge sort is a divide-and-conquer algorithm that recursively divides the input list into smaller sub-lists, sorts each sub-list, and then merges the sorted sub-lists back together.

Best Case Time Complexity

In the best case, the input list is already sorted. Even in this case, merge sort still divides the input list into smaller sub-lists and merges them back together, resulting in a time complexity of O(n log n).

Worst Case Time Complexity

In the worst case, merge sort still has a time complexity of O(n log n). This is because the algorithm recursively divides the input list into halves until each sub-list contains only one element, and then merges the sorted sub-lists back together.

Space Complexity

Merge sort requires extra memory to store the sub-lists during the recursive calls. The amount of memory required by merge sort depends on the implementation, but it is typically O(n) or O(log n).

Optimality

Merge sort is an optimal sorting algorithm in terms of worst-case time complexity. However, it has a higher space complexity than some other sorting algorithms, which can be a consideration in certain applications.

0.2.3 Comparison

Best Case Time Complexity

Both insertion sort and merge sort have a best-case time complexity of O(n) when the input list is already sorted. However, in practice, insertion sort may be faster for small input sizes due to its lower constant factors.

Worst Case Time Complexity

Insertion sort has a worst-case time complexity of O(n^2), while merge sort has a worst-case time complexity of O(n log n). Merge sort is therefore generally faster for larger input sizes.

Space Complexity

Insertion sort has a space complexity of O(1), which means that it uses a constant amount of extra memory regardless of the size of the input list. This makes insertion sort a good choice for situations where memory usage is a concern, especially for large input sizes.

Merge sort, on the other hand, has a space complexity that depends on the implementation. In the simplest implementation, merge sort requires O(n) extra memory, which can be a concern for very large input sizes. However, other implementations can reduce the amount of extra memory required to O(log n). This makes merge sort a good choice for situations where worst-case time complexity is a concern, and memory usage is not a major constraint.

Overall, the choice between insertion sort and merge sort depends on the specific requirements of the application. If memory usage is a major concern, and the input size is large, then insertion sort may be a better choice. If worst-case time complexity is a major concern, and memory usage is not a constraint, then merge sort may be a better choice.

Optimality

Insertion sort is not an optimal sorting algorithm in terms of worst-case time complexity, while merge sort is optimal. However, insertion sort may be useful in certain situations due to its simplicity and low constant factors. Overall, the choice between insertion sort and merge sort depends on the specific requirements of the application.

0.3 Other Sorting algorithms

0.3.1 Quick Sort

Quick Sort is an efficient, comparison-based sorting algorithm invented by Tony Hoare in 1959. It is a Divide and Conquer algorithm that divides the input array into smaller sub-arrays, then recursively sorts those sub-arrays. Quick Sort has an average case time complexity of O(n log n) and a worst-case time complexity of O(n^2).

Algorithm Overview

The Quick Sort algorithm follows the divide and conquer paradigm. The algorithm takes an unsorted array and divides it into two sub-arrays based on a pivot element. The pivot element is chosen such that all elements to the left of the pivot are smaller than the pivot, and all elements to the right of the pivot are larger than the pivot. This process is repeated recursively for the left and right sub-arrays until the entire array is sorted.

Algorithm Steps

1. Choose a pivot element from the array. There are different strategies for selecting a pivot element, such as choosing the first element, the last element, or a random element. In this example, we will choose the last element as the pivot.

2. Partition the array such that all elements to the left of the pivot are smaller than the pivot, and all elements to the right of the pivot are larger than the pivot. We can do this using two pointers, one starting from the beginning of the array, and the other starting from the end of the array. If the element at the left pointer is larger than the pivot and the element at the right pointer is smaller than the pivot, we swap the two elements. We continue this process until the two pointers meet. At this point, we know that all elements to the left of the pivot are smaller than the pivot, and all elements to the right of the pivot are larger than the pivot.

3. Recursively apply steps 1 and 2 to the left and right sub-arrays until the entire array is sorted.

Example

Let's take the following array as an example: [8, 4, 2, 9, 3, 6]

1. Choose the last element (6) as the pivot.

2. Partition the array: [4, 2, 3, 6, 9, 8] The left pointer starts at the beginning of the array (8), and the right pointer starts at the end of the array (4). The left pointer moves to the right until it reaches 9, which is larger than the pivot. The right pointer moves to the left until it reaches 2, which is smaller than the pivot. The two elements are swapped, and the process continues until the two pointers meet.

3. Apply quick sort recursively to the left and right sub-arrays: [4, 2, 3] [9, 8] Pivot is 3: [2, 3, 4] [9, 8] Pivot is 8: [2, 3, 4, 8, 9, 6] Pivot is 6: [2, 3, 4, 6, 8, 9]

Time Complexity

The time complexity of Quick Sort can be analyzed in the best case, worst case, and average case scenarios.

1. Best Case: The best case scenario for Quick Sort occurs when the pivot element divides the input data into two equal parts. In this case, the algorithm requires log(n) iterations to sort the data, resulting in a time complexity of O(nlogn).

2. Worst Case: The worst case scenario for Quick Sort occurs when the pivot element is chosen as the smallest or largest element in the input data. In this case, the algorithm requires n iterations to sort the data, resulting in a time complexity of O(n^2).

3. Average Case: The average case scenario for Quick Sort occurs when the pivot element is chosen randomly, and the input data is uniformly distributed. In this case, the algorithm requires log(n) iterations to sort the data, resulting in a time complexity of O(nlogn).

Space Complexity

The space complexity of Quick Sort can be analyzed in terms of the amount of extra memory required by the algorithm. Quick Sort is an in-place sorting algorithm, meaning that it does not require any extra memory to store the sorted data. However, the algorithm requires extra memory for the recursive function calls.

1. Stack Space: The recursive function calls in Quick Sort use a stack to store the return addresses and the parameters of the function calls. The amount of stack space required by Quick Sort is proportional to the maximum depth of the recursion tree. In the worst case scenario, when the input data is already sorted or reverse sorted, the maximum depth of the recursion tree is n, resulting in a space complexity of O(n).

2. Heap Space: Quick Sort also requires heap space for the temporary variables used in the partitioning process. The amount of heap space required by Quick Sort is proportional to the size of the input data. In the worst case scenario, when the input data is already sorted or reverse sorted, Quick Sort requires maximum heap space to store the temporary variables, resulting in a space complexity of O(n).

0.3.2 Bubble Sort

Bubble Sort is a simple sorting algorithm that works by repeatedly swapping adjacent elements if they are in the wrong order. It is named as Bubble Sort because the smaller or larger elements gradually bubble up to the top of the list. Bubble Sort is a comparison-based algorithm and has a time complexity of O(n^2) in the worst case. It is not recommended for large datasets, but it is easy to understand and implement, making it a good choice for small datasets.

Algorithm

The Bubble Sort algorithm can be described in the following steps:

1. Start with the first element of the list.
2. Compare the first and second elements, if they are in the wrong order, swap them.
3. Move to the next pair of elements, and continue comparing and swapping until the end of the list is reached.
4. Repeat steps 1-3 until no more swaps are needed, i.e., the list is sorted.

Time complexity

The time complexity of an algorithm is a measure of the amount of time it takes to run as a function of the size of the input. The time complexity of Bubble Sort can be analyzed as follows:

1. Worst Case: The worst case occurs when the list is in reverse order, i.e., the largest element is at the beginning of the list and the smallest element is at the end of the list. In this case, the algorithm needs to make n-1 passes through the list, and each pass requires n-1 comparisons and swaps in the worst case. Therefore, the total number of comparisons and swaps is:

   $(n-1)+(n-2)+...+1 = n*(n-1)/2$

   Hence, the time complexity of Bubble Sort in the worst case is O(n^2).

2. Best Case: The best case occurs when the list is already sorted, and no swaps are needed. In this case, the algorithm only needs to make one pass and perform n-1 comparisons. Therefore, the time complexity of Bubble Sort in the best case is O(n).

3. Average Case: The average case occurs when the list is randomly ordered. In this case, the algorithm needs to make about n/2 passes, and each pass requires about n/2 comparisons and swaps on average. Therefore, the total number of comparisons and swaps is:

   $(n/2)*(n/2) = n^2/4$ Hence, the time complexity of Bubble Sort in the average case is also O(n^2).

4. Space Complexity: The space complexity of an algorithm is a measure of the amount of memory it requires as a function of the size of the input. The space complexity of Bubble Sort can be analyzed as follows:

   (a) In-Place Bubble Sort is an in-place sorting algorithm, which means that it sorts the list by swapping adjacent elements without using any additional memory. Therefore, the space complexity of Bubble Sort is O(1).

(b) Not In-Place If we modify Bubble Sort to create a new list for the sorted elements, the space complexity would be O(n) because we would need to allocate memory for the new list.

0.3.3 Tim Sort

Tim Sort is a sorting algorithm that was designed by Tim Peters for use in the Python programming language. It is a hybrid sorting algorithm that uses both insertion sort and merge sort algorithms in its implementation. Tim Sort has been widely adopted as the default sorting algorithm in many programming languages, including Python, Java, and C++. In this article, we will provide a detailed explanation of how Tim Sort works, including its implementation, time complexity, and examples.

Algorithm Overview

Tim Sort is a divide-and-conquer algorithm that first divides the input into small runs or subarrays. It then sorts each run using Insertion Sort and merges the runs using Merge Sort. The algorithm is designed to take advantage of the fact that many real-world datasets have already partially sorted runs.

The first step of Tim Sort is to divide the input into runs. A run is defined as a subarray of the input that is already sorted in either ascending or descending order. Tim Sort uses a modified version of Insertion Sort to identify and create runs in the input.

Once the runs have been created, Tim Sort uses a modified version of Merge Sort to merge them. The algorithm merges the runs in a way that takes advantage of the fact that many of the runs are already partially sorted.

The main idea behind Tim Sort is to use Insertion Sort to create runs of size between 32 and 64. These runs are then merged using Merge Sort. The algorithm is designed to minimize the number of comparisons and swaps performed during the merge phase.

The key features of Tim Sort are:

1. It is a hybrid sorting algorithm that combines the best features of insertion sort and merge sort.
2. It uses a "run" data structure to divide the input array into smaller chunks.
3. It employs a modified merge sort algorithm to merge the sorted chunks back together.
4. It includes several optimizations to improve performance, such as the use of galloping mode and binary search.

Time Complexity Analysis

To analyze the time complexity of Tim Sort, we need to consider two main factors: the time complexity of creating the runs and the time complexity of merging the runs.

The time complexity of creating the runs is O(n log n). This is because the algorithm uses a modified version of Insertion Sort to create runs of size between 32 and 64. The worst-case time complexity of Insertion Sort is O(n^2), but in practice, the runs are often much smaller than n, so the time complexity is usually much lower.

The time complexity of merging the runs is also O(n log n). This is because the algorithm uses a modified version of Merge Sort to merge the runs. Merge Sort has a worst-case time complexity of O(n log n), and the modified version used by Tim Sort is designed to minimize the number of comparisons and swaps performed during the merge phase.

Therefore, the overall time complexity of Tim Sort is O(n log n). This means that the algorithm is very efficient and performs well on many different kinds of input.

Space Complexity

The space complexity of an algorithm is the amount of memory it requires to run. In the case of Tim sort, the space complexity is O(n), where n is the size of the input array.

This is because Tim sort requires extra memory to store the temporary arrays that are used during the sorting process. Specifically, Tim sort uses a temporary array of size n/2 to merge two sorted sub-arrays, and it also uses a stack to keep track of the sub-arrays that need to be merged.

Let's examine each of these in more detail:

1. Temporary Array: When merging two sorted sub-arrays, Tim sort creates a temporary array of size n/2. This temporary array is used to store the merged result before copying it back to the original array. The size of the temporary array is determined by the size of the input array, which means that the space complexity of Tim sort is proportional to the size of the input array.

2. Stack: Tim sort uses a stack to keep track of the sub-arrays that need to be merged. Each time Tim sort merges two sub-arrays, it pushes the resulting merged sub-array onto the stack. The algorithm then pops the top two sub-arrays from the stack and merges them. This process continues until all sub-arrays have been merged.

   The size of the stack is proportional to the logarithm of the size of the input array. This is because the number of times Tim sort needs to merge sub-arrays is proportional to the number of times the size of the sub-arrays can be halved before reaching a sub-array of size 1. Since the size of the sub-arrays is halved each time, the number of times this can be done before reaching a sub-array of size 1 is logarithmic in the size of the input array.