DATA STRUCTURES AND ALGORITHMS

NOTE02-SORTING PROBLEM (INSERTION/MERG SORT)

EDITED BY

ARSHIA GHAROONI

The University of central Tehran branch
Tehran

2023

0.1 Introduction

Sorting is the process of arranging a collection of items in a specific order, usually ascending or descending. Sorting is one of the most fundamental operations in computer science and is used in a wide range of applications, including data processing, searching, and machine learning.

Sorting algorithms are the algorithms that sort a given set of data into a specific order. There are various sorting algorithms available, each with its own advantages and disadvantages. In this lecture note, we will discuss the two popular sorting algorithms: Insertion Sort and Merge Sort.

0.2 Insertion Sort

Insertion Sort is a simple and efficient sorting algorithm that works by inserting each item in the proper place in a sorted sequence. Insertion Sort is a good choice for small arrays or for arrays that are mostly sorted, as it has a low overhead and performs well in these situations.

In this lecture note, we will discuss Insertion Sort in detail, including its algorithm, time complexity, space complexity, advantages, and disadvantages.

0.2.1 Insertion Sort Algorithm

The basic idea behind Insertion Sort is to divide the array into two parts: a sorted part and an unsorted part. Initially, the sorted part contains only the first element of the array, and the unsorted part contains the rest of the elements. The algorithm then iterates over the unsorted part, taking each element in turn and inserting it into the proper place in the sorted part.

The pseudocode for Insertion Sort is as follows:

```
for i = 1 to n-1
   j = i
   while j > 0 and array[j-1] > array[j]
      swap array[j] and array[j-1]
      j = j - 1
```

Let's break down the algorithm into steps:

1. Starting from the second element of the array, iterate over the unsorted part of the array.
2. Take the current element and compare it with the elements in the sorted part of the array.

3. Move the larger elements one position to the right to make room for the current element.

4. Insert the current element in the proper place in the sorted part of the array.

5. Repeat steps 2 to 4 for all elements in the unsorted part of the array.

0.2.2 Time Complexity of Insertion Sort

The time complexity of Insertion Sort is O(n2) in the worst case, where n is the number of elements in the array. This is because the algorithm needs to compare each element in the unsorted part of the array with each element in the sorted part of the array.

In the best case, when the array is already sorted, the time complexity of Insertion Sort is O(n), as the algorithm only needs to compare each element with its predecessor and do no swaps.

In the average case, the time complexity of Insertion Sort is also O(n2), as the array is usually not sorted and the algorithm needs to perform many comparisons and swaps.

0.2.3 Space Complexity of Insertion Sort

The space complexity of Insertion Sort is O(1), as the algorithm does not use any extra space apart from the input array.

0.2.4 Advantages of Insertion Sort

1. Simple and easy to implement: Insertion Sort is one of the simplest sorting algorithms to implement and understand, making it a good choice for small programs or teaching purposes.

2. Low overhead: Insertion Sort has a low overhead, as it does not use any extra space apart from the input array.

3. Good for small arrays or mostly sorted arrays: Insertion Sort performs well on small arrays or arrays that are mostly sorted, as it does not need to perform many comparisons and swaps.

0.2.5 Disadvantages of Insertion Sort

1. Slow for large arrays: Insertion Sort has a time complexity of O(n2), which makes it slow for large arrays.

2. Not suitable for complex data structures: Insertion Sort is not suitable for complex data structures, such as trees or graphs, as it requires a linear data structure

0.2.6 Improvements to Insertion Sort

Although Insertion Sort is not suitable for large arrays or complex data structures, there are several improvements that can be made to the algorithm to make it more efficient.

1. Binary Insertion Sort: Binary Insertion Sort is a variation of Insertion Sort that uses binary search to find the proper position for each element in the sorted part of the array, reducing the number of comparisons required.
2. Shell Sort: Shell Sort is a generalization of Insertion Sort that divides the array into smaller subarrays and applies Insertion Sort to each subarray, reducing the number of comparisons required.
3. Comb Sort: Comb Sort is a variation of Insertion Sort that uses a larger gap between elements to compare, reducing the number of swaps required.

0.3 Merg Sort

Merge Sort is a widely used sorting algorithm that uses a divide-and-conquer approach to sort an array of elements. In Merge Sort, the array is divided into two halves, each half is recursively sorted, and then the two sorted halves are merged to produce the final sorted array. In this lecture note, we will

discuss Merge Sort in detail, including its algorithm, time complexity, space complexity, advantages, and disadvantages.

0.3.1 Merge Sort Algorithm

The Merge Sort algorithm consists of two main steps: divide and conquer. In the divide step, the array is divided into two halves, and each half is recursively sorted using the same algorithm. In the conquer step, the two sorted halves are merged together to produce the final sorted array. The pseudocode

for Merge Sort is as follows:

```
function merge_sort(array)
   if length(array) <= 1
      return array
   else
      middle = length(array) / 2
      left_half = merge_sort(array[0:middle])
      right_half = merge_sort(array[middle:length(array)])
      return merge(left_half, right_half)

function merge(left_half, right_half)
   result = []
   i = 0
   j = 0
   while i < length(left_half) and j < length(right_half)
      if left_half[i] < right_half[j]
         result.append(left_half[i])
         i = i + 1
      else
         result.append(right_half[j])
         j = j + 1
   result.extend(left_half[i:])
   result.extend(right_half[j:])
   return result
```

Let's break down the algorithm into steps:

1. Check if the length of the array is less than or equal to 1. If it is, return the array as it is already sorted.
2. Divide the array into two halves, using the middle index.
3. Recursively call Merge Sort on each half of the array.
4. Merge the two sorted halves using the Merge function.
5. Return the sorted array.

The Merge function takes two sorted arrays as input and produces a single sorted array as output. The function works by iterating over the elements of the two arrays and inserting them in the proper order in a new array.

0.3.2 Time Complexity of Merge Sort

The time complexity of Merge Sort is O(n log n) in the worst, best, and average cases, where n is the number of elements in the array. This is because the array is recursively divided into halves until the base case is reached, resulting in a logarithmic number of divisions. The merge operation takes linear time, as each element is compared only once and inserted into the result array.

0.3.3 Space Complexity of Merge Sort

The space complexity of Merge Sort is O(n), where n is the number of elements in the array. This is because the algorithm creates a temporary array of the same size as the input array to store the result of the merge operation.

0.3.4 Advantages of Merge Sort

1. Efficient for large arrays: Merge Sort is efficient for large arrays as its time complexity is O(n log n).
2. Stable Sorting: Merge Sort is a stable sorting algorithm, which means that the order of equal elements is preserved during the sorting process.
3. Suitable for linked lists: Merge Sort is suitable for linked lists as it does not require random access to elements in the array.

0.3.5 Disadvantages of Merge Sort

1. Space complexity: Merge Sort has a space complexity of O(n), which can be a disadvantage for systems with limited memory.
2. Recursive: Merge Sort is a recursive algorithm, which means that it requires additional overhead to manage the recursive calls. This can be a disadvantage in certain situations.
3. Not in-place: Merge Sort requires additional memory to store the result of the merge operation. This can be a disadvantage in situations where memory is limited.

0.3.6 Improvements to Merge Sort

Although Merge Sort is a highly efficient sorting algorithm, there are several improvements that can be made to the algorithm to make it even more efficient.

1. Iterative Merge Sort: Iterative Merge Sort is a variation of Merge Sort that uses an iterative approach instead of a recursive approach. This can reduce the overhead of managing recursive calls.
2. In-Place Merge Sort: In-Place Merge Sort is a variation of Merge Sort that performs the merge operation in-place, without requiring additional memory. This can be useful in situations where memory is limited.
3. Hybrid Merge Sort: Hybrid Merge Sort is a variation of Merge Sort that uses a combination of Merge Sort and another sorting algorithm, such as Insertion Sort or Quick Sort, to improve efficiency.

0.4 Divide and Conquer

Divide and Conquer is a problem-solving technique that involves breaking down a problem into smaller subproblems, solving each subproblem recursively, and then combining the results of each subproblem to solve the original problem. This technique is used in a wide range of algorithms, including sorting algorithms, searching algorithms, and algorithms for mathematical problems such as matrix multiplication.

0.4.1 The Divide and Conquer Process

The Divide and Conquer process consists of three main steps: Divide, Conquer, and Combine.

1. Divide: The first step is to divide the problem into smaller subproblems that are easier to solve. This is done by breaking the problem down into smaller pieces, often recursively, until the subproblems are small enough to be solved directly.

2. Conquer: The second step is to solve each subproblem recursively. This is done by applying the same algorithm to each subproblem, which should be easier to solve than the original problem.

3. Combine: The final step is to combine the results of each subproblem to solve the original problem. This is done by merging the results of each subproblem, often in a specific order or pattern.

0.4.2 Examples of Divide and Conquer Algorithms

1. Merge Sort: Merge Sort is a sorting algorithm that uses the Divide and Conquer technique to sort an array of elements. It works by dividing the array into two halves, recursively sorting each half, and then merging the two sorted halves together.

2. Quick Sort: Quick Sort is another sorting algorithm that uses the Divide and Conquer technique. It works by selecting a pivot element from the array, dividing the array into two subarrays based on the pivot element, recursively sorting each subarray, and then combining the results.

3. Binary Search: Binary Search is a searching algorithm that uses the Divide and Conquer technique to find a target element in a sorted array. It works by dividing the array in half, comparing the target element to the middle element, and recursively searching either the left or right half of the array based on the comparison result.