Continue

Sorting algorithms are used to sort a data structure according to a specific order relationship, such as numerical order or lexicographical order. This operation is one of the most important and widespread in computer science. For a long time, new methods have been developed to make this procedure faster and faster. There are currently hundreds of different sorting algorithms, each with its own specific characteristics. They are classified according to two metrics: space complexity and time complexity. Those two kinds of complexity are represented with asymptotic notations, mainly with the symbols O, , , representing respectively the upper bound, the tight bound, and the lower bound of the algorithm's complexity, specifying in brackets an expression in terms of n, the number of the elements of the data structure. Most of them fall into two categories: Logarithmic The complexity is proportional to the binary logarithm (i.e to the base 2) of n. An example of a logarithmic sorting algorithm is Quick sort, with space and time complexity O(n log n). Quadratic The complexity is proportional to the square of n. An example of a quadratic sorting algorithm is Bubble sort, with a time complexity of O(n2). Space and time complexity can also be further subdivided into 3 different cases: best case, average case and worst case. Sorting algorithms can be difficult to understand and it's easy to get confused. We believe visualizing sorting algorithms can be a great way to better understand their functioning while having fun! Sorting is an essential data organization technique that plays a significant role in various fields, such as computer science, mathematics, and data analysis. It involves arranging a collection of items in a specific order, making it easier to search, retrieve, and analyze data efficiently. This article will delve into the fundamental concepts of sorting, exploring its principles, methods, and applications, providing readers with a comprehensive understanding of this crucial data organization technique.The Importance Of Sorting In Data OrganizationSorting is an essential technique used in data organization that plays a crucial role in various aspects of life, ranging from everyday tasks to complex technical applications. Sorting allows us to arrange data in a specific order, making it easier to search, retrieve, and analyze information efficiently.One of the primary reasons why sorting is important in data organization is its ability to enhance data retrieval speed. When data is sorted, it can be searched and accessed more quickly, as the ordered structure facilitates optimized search algorithms. This is particularly useful in large datasets where finding specific information would otherwise be time-consuming and cumbersome.Moreover, sorting enables us to identify patterns and trends within the data. By arranging the data in a specific order, we can easily identify the highest or lowest values, outliers, or sequences that would otherwise remain hidden. This plays a vital role in data analysis, decision-making processes, and identifying anomalies or errors within datasets.Furthermore, sorting is extensively utilized in various industries and applications, including finance, e-commerce, healthcare, inventory management, and more. For example, in e-commerce, sorting allows customers to view products based on their preferences, such as price, popularity, or customer ratings.In conclusion, sorting is a fundamental technique in data organization that significantly improves data retrieval speed, facilitates data analysis, and finds wide-ranging applications across industries. Understanding the importance of sorting is crucial for efficient data management and information processing.Different Types Of Sorting Algorithms And Their ApplicationsSorting algorithms are essential in organizing data efficiently. There are various types of sorting algorithms, each with its own strengths and weaknesses, making them suitable for different applications.One popular sorting algorithm is Bubble Sort, an easy-to-understand algorithm that repeatedly compares adjacent elements and swaps them if they are in the wrong order. While Bubble Sort is simple, it is not efficient for large datasets due to its time complexity of O(n^2).Another widely used sorting algorithm is Merge Sort, which follows a divide and conquer approach. It divides the dataset into smaller sub-arrays, sorts them individually, and then merges them back into one sorted array. Merge Sort has a time complexity of O(n log n), making it suitable for large datasets.Quicksort is another efficient sorting algorithm. It selects a pivot element and partitions the dataset around that pivot, recursively sorting the subsets. Quicksort has an average time complexity of O(n log n) but can degrade to O(n^2) in the worst-case scenario.Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure. It repeatedly extracts the maximum element from the heap and places it in the sorted array. Heap Sort has a time complexity of O(n log n) and is useful when a stable sort is not required.Different sorting algorithms have applications depending on the specific needs of the data. Some sorting algorithms may be better suited for small datasets, while others excel at sorting large amounts of data efficiently. Understanding the various types of sorting algorithms and their strengths helps in selecting the most appropriate sorting technique for a given scenario.Understanding The Basic Principles Behind Sorting TechniquesSorting techniques are essential for data organization as they enable the arrangement of information in a structured and accessible manner. To comprehend how sorting works, it is crucial to understand the basic principles behind these techniques.At its core, sorting involves reordering a collection of elements in a specific order, such as ascending or descending. The fundamental principle behind sorting is the comparison of elements based on certain criteria, such as numerical value or alphabetical order.One of the key concepts in sorting is the notion of a key, which is a value associated with each element that determines its position in the final sorted sequence. Sorting algorithms use these keys to compare elements and arrange them accordingly.Two primary approaches employed in sorting algorithms are comparison-based and non-comparison-based techniques. Comparison-based sorting algorithms compare pairs of elements and decide their order based on the results of those comparisons. On the other hand, non-comparison-based techniques utilize additional information, such as the distribution of elements or their positions in the data structure, to optimize the sorting process.Understanding these basic principles of sorting techniques is vital as it enables us to appreciate the various sorting algorithms in greater depth and comprehend their applications and efficiency.1. The importance of sorting in data organization2. Different types of sorting algorithms and their applications3. Understanding the basic principles behind sorting techniquesStep-by-step Process Of Sorting Data In Ascending Or Descending OrderSorting data is an essential technique in data organization that arranges information in a specific order to make it more manageable and accessible. The process of sorting generally involves taking a collection of data elements and arranging them in either ascending or descending order based on a specific comparison criterion.The step-by-step process of sorting data can be summarized as follows:1. Select a suitable sorting algorithm: There are numerous sorting algorithms available, each with its own strengths and weaknesses. Commonly used algorithms include Bubble Sort, Insertion Sort, Selection Sort, Merge Sort, Quick Sort, and Heap Sort.2. Define the comparison function: Sorting relies on comparing elements to determine their relative order. The comparison function specifies the criteria for this comparison, such as numerical or alphabetical order.3. Divide the data: If using a divide-and-conquer algorithm like Merge Sort or Quick Sort, the data is divided into smaller subsets for easier processing.4. Perform comparisons and swaps: Iterate through the data, comparing adjacent elements, and swapping them if necessary to establish the desired order.5. Repeat until sorted: Continue the iteration and swapping process until the data is completely sorted.By following these steps, data can be efficiently sorted, allowing for easier search, analysis, and manipulation of the information. Sorting plays a crucial role in various fields, from organizing large datasets in computer science to alphabetizing lists in everyday applications.Overview Of Common Challenges And Considerations In Sorting Large DatasetsSorting large datasets can be a complex task that requires careful consideration of several challenges. One of the main challenges is the limited memory capacity available for sorting. When dealing with large datasets, the amount of memory required for sorting can surpass the available memory, leading to performance issues or even program crashes. To overcome this challenge, sorting algorithms often employ external sorting techniques, which involve using external storage such as hard drives.Another consideration when sorting large datasets is the time complexity of the algorithm being used. Sorting large datasets can be time-consuming, and choosing an algorithm with an efficient time complexity becomes crucial. Different sorting algorithms have different time complexities, such as O(n^2) for algorithms like Bubble Sort and Insertion Sort, or O(n log n) for more efficient algorithms like Merge Sort and Quick Sort.Additionally, sorting large datasets may require considering other factors, such as stability, which ensures that records with equal keys maintain their relative order. Other considerations include adaptability to different data types or distributions, the need for parallel processing, and the ability to handle duplicate values efficiently.Therefore, when sorting large datasets, it is essential to consider memory limitations, time complexity, stability, adaptability to different data types, and other relevant factors for selecting the most suitable sorting algorithm.Performance Analysis: Comparing The Efficiency Of Different Sorting AlgorithmsSorting algorithms are not created equal, and their performance can vary widely depending on the size and nature of the dataset being sorted. In this section, we will delve into the importance of performance analysis and compare the efficiency of different sorting algorithms.Performance analysis allows us to quantify and compare the time complexity and space complexity of different sorting algorithms. Time complexity measures how the running time of an algorithm grows as the input size increases, while space complexity determines the amount of memory required by an algorithm.Some of the most common sorting algorithms include Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, and Heap Sort. Each algorithm has its strengths and weaknesses, making them suitable for different scenarios.By analyzing the time and space complexity of these algorithms, we can identify the most efficient sorting algorithm for a specific dataset. A sorting algorithm with a lower time complexity and space complexity will generally outperform others, especially when sorting large datasets.In this section, we will provide a comparative analysis of these sorting algorithms, discussing their strengths, weaknesses, and recommended use cases. Understanding their efficiency will equip you with the knowledge needed to make informed decisions when sorting data.# Real-world examples of sorting in everyday applications and industriesSorting data is a fundamental technique that finds applications in various fields and industries. From everyday tasks to critical systems, sorting plays a vital role in organizing data efficiently. Real-world examples of sorting can be found in a multitude of applications, including e-commerce, finance, healthcare, logistics, and more.In the e-commerce sector, sorting is used to arrange products based on customer preferences, price range, popularity, or other criteria. It enables users to easily navigate through catalogs, facilitating a seamless shopping experience. Similarly, financial institutions utilize sorting to arrange transactions and create accurate reports, enabling efficient auditing and analysis.In the healthcare industry, sorting is crucial for managing patient records and medical imaging data. Sorting algorithms help arrange patient data based on parameters such as age, medical history, or urgency, allowing healthcare professionals to quickly access relevant information for diagnosis and treatment.Sorting is also employed in the logistics industry to arrange shipments efficiently. By sorting packages based on factors like destination, weight, or size, companies can streamline their delivery processes, minimize errors, and ensure timely deliveries.These real-world examples only scratch the surface of the countless applications of sorting. From databases to web applications, sorting is an essential tool for organizing and processing data effectively across various industries.FAQsQ1: What is sorting?Sorting is a fundamental data organization technique used to arrange data elements in a specified order. It involves reordering elements of a collection, such as an array or a list, according to a defined criterion, such as numerical or alphabetical order.Q2: Why is sorting an essential data organization technique?Sorting plays a crucial role in various computer applications and algorithms. It enables efficient searching, easier data retrieval, and faster processing of information. Sorting also facilitates better data analysis and optimization in a wide range of fields, including databases, computer graphics, and scientific computing.Q3: How does sorting work?Sorting involves comparing elements of a collection and rearranging them based on the comparison results. Common sorting algorithms, such as bubble sort, selection sort, and merge sort, employ different techniques to compare and rearrange the elements until the desired order is achieved. The choice of the sorting algorithm depends on factors like the size of the data set and the desired efficiency.Q4: What are some popular sorting algorithms?Several popular sorting algorithms have been developed over the years. Some commonly used ones include: Bubble Sort: Iteratively compares adjacent elements and swaps them if they are in the wrong order. Insertion Sort: Builds the sorted array gradually by repeatedly inserting the next element in its proper place. Quick Sort: Divides the array into smaller sub-arrays, based on a chosen pivot element, and recursively sorts them. Merge Sort: Divides the array into smaller sub-arrays, recursively sorts them, and then merges them back together in order.Remember, the choice of a sorting algorithm depends on factors like the type of data, its size, and the desired efficiency for a particular application.ConclusionIn conclusion, sorting is a fundamental technique used in data organization that involves arranging data in a particular order. It plays a crucial role in various applications such as searching, indexing, and efficient data retrieval. Understanding the basics of sorting and its various algorithms can greatly enhance the efficiency and effectiveness of data processing, ultimately leading to improved system performance and user satisfaction. A Sorting Algorithm is used to rearrange a given array or list of elements in an order. For example, a given array [10, 20, 5, 2] becomes [2, 5, 10, 20] after sorting in increasing order and becomes [20, 10, 5, 2] after sorting in decreasing order. There exist different sorting algorithms for different different types of inputs, for example a binary array, a character array, an array with a large range of values or an array with many duplicates or a small vs large array.The algorithms may also differ according to output requirements. For example, stable sorting (or maintains original order of equal elements) or not stable. Sorting is provided in library implementation of most of the programming languages. These sorting functions typically are general purpose functions with flexibility of providing the expected sorting order (increasing or decreasing or by a specific key in case of objects).Basics Introduction to Sorting Applications of Sorting Sorting Algorithms:Comparison Based : Selection Sort, Bubble Sort, Insertion Sort, Merge Sort, Quick Sort, Heap Sort, Cycle Sort, 3-way Merge SortNon Comparison Based : Counting Sort, Radix Sort, Bucket Sort, TimSort, Comb Sort, Pigeonhole SortHybrid Sorting Algorithms : IntroSort, Tim SortLibrary Implementations:Easy ProblemsMedium Problems Hard ProblemsQuick Links : DSA Tutorial - Learn Data Structures and Algorithms Getting Started with Array Data Structure String in Data Structure Hashing in Data Structure Linked List Data Structure Stack Data Structure Queue Data Structure Tree Data Structure Graph Data Structure Trie Data Structure Searching Algorithms Sorting Algorithms Introduction to Recursion Greedy Algorithms Graph Algorithms Dynamic Programming or DP Bitwise Algorithms Segment Tree Pattern Searching Geometry After reading this chapter you willunderstand the problem of sorting a set of numbers (or letters) in a defined order.be able to implement a variety of well-known sorting algorithms.be able to evaluate the efficiency and relative advantages of different algorithms given different input cases.be able to analyze sorting algorithms to determine their average-case and worst-case time and space complexity. Applying an order to a set of objects is a common general problem in life as well as computing. You may open up your email and see that the most recent emails are at the top of your inbox. Your favorite radio station may have a top-10 ranking of all the new songs based on votes from listeners. You may be asked by a relative to put a shelf of books in alphabetical order by the authors names. All these scenarios involve ordering or ranking based on some value. To achieve these goals, some form of sorting algorithm must be used. A key observation is that these sorting problems rely on a specific comparison operator that imposes an ordering (a comes before b in alphabetical order, and 10 < 12 in numerical order). As a terminology note, alphabetical ordering is also known as lexical, lexicographic, or dictionary ordering. Alphabetical and numerical orderings are usually the most common orderings, but date or calendar ordering is also common.In this chapter, we will explore several sorting algorithms. Sorting is a classic problem in computer science. These algorithms are classic not because you will often need to write sorting algorithms in your professional life. Rather, sorting offers an easy-to-understand problem with a diverse set of algorithms, making sorting algorithms an excellent starting point for the analysis of algorithms.To begin our study, let us take a simple example sorting problem and explore a straightforward algorithm to solve it.An Example Sorting ProblemSuppose we are given the following array of 8 values and asked to sort them in increasing order:Figure 3.1How might you write an algorithm to sort these values? Our human mind could easily order these numbers from smallest to largest without much effort. What if we had 20 values? 200? We would quickly get tired and start making mistakes. For these values, the correct ordering is 22, 24, 27, 35, 43, 45, 47, 48. Give yourself some time to think about how you would solve this problem. Dont consider arrays or indexes or algorithms. Think about doing it just by looking at the numbers. Try it now.Reflect on how you solved the problem. Did you use your fingers to mark the positions? Did you scan over all the values multiple times? Taking some time to think about your process may help you understand how a computer could solve this problem.One simple solution would be to move the smallest value in the list to the leftmost position, then attempt to place the next smallest value in the next available position, and so on until reaching the last value in the list. This approach is called Selection Sort.Selection SortSelection Sort is an excellent place to start for algorithm analysis. This sort can be constructed in a very simple way using some bottom-up design principles. We will take this approach and work our way up to a conceptually simple sorting algorithm. Before we get started, let us outline the Selection Sort algorithm. As a reminder, we will use 0-based indexing with arrays:Start by considering the first or 0 position of the array.Find the index of the smallest value in the array from position 0 to the end.Exchange the value in position 0 with the smallest value (using the index of the smallest value).Repeat the process by considering position 1 of the array (as the smallest value is now in position 0).The algorithm works by repeatedly selecting the smallest value in the given range of the array and then placing it in its proper position along the array starting in the first position. With a little thought for our design, we can construct this algorithm in a way that greatly simplifies its logic.Selection Sort ImplementationLet us start by creating the exchange function. Our exchange function will take an array and two indexes. It will then swap the value in the given positions within the array. For example, exchange(array, 1, 3) will take the value in position 1 and place it in position 3 as well as taking the value in position 3 and placing it in position 1. Let us look at what might happen by calling it on our previous array.Here is our previous array with indexes added:Figure 3.2After calling exchange(array, 1, 3), we get this:Figure 3.3This function is the first tool we need to build Selection Sort. Lets explore one implementation of this function.This function will do nothing if the indexes are identical. When we have separate indexes, the corresponding values in the array are exchanged. This evolves the state of the array by switching two values. In this implementation, we do not make any checks to see if an exchange could be made. It may be worth checking if the indexes are identical. It also may be worth checking if the indexes are valid (for example, between 0 and n 1), but this exercise is left to the reader.Now we need another tool to help us select the next smallest value to put in its correct order. For this task, we need something that is conceptually the same as a findMin function. For our algorithms, we would need to make a few modifications to the regular findMin. The two additions we need to make are as follows: (1) We need to get the index of the smallest value, not just its value. (2) We want to search only within a given range. The last modification will let us choose the smallest value for position 0 and then choose the second smallest value for position 1 (chosen from positions 1 to n 1).Let us look at one implementation for this algorithm.For this algorithm, we can set any start coordinate and find the index of the smallest value from the start to the end of the array. This simple procedure gives us a lot of power, as we will learn.Now that we have our tools created, we can write Selection Sort. This leads to a simple implementation thanks to the design that decomposed the problem into smaller tasks.We now have our first sorting algorithm. This algorithm provides a great example of how design impacts the complexity of an implementation. Combining a few simple ideas leads to a powerful new tool. This practice is sometimes called encapsulation. The complexity of the algorithm is encapsulated behind a few functions to provide a simple interface. Mastering this art is the key to becoming a successful computing professional. Amazing things can be built when the foundation is functional, and good design removes a lot of the difficulty of programming. Try to take this lesson to heart. Good design gives us the perspective to program in a manner that is closer to the way we think. Context that improves our ability to think about problems improves our ability to solve problems.Selection Sort ComplexityIt should be clear that the sorting of the array using Selection Sort does not use any extra space other than the original array and a few extra variables. The space complexity of Selection Sort is O(n).Analyzing the time complexity of Selection Sort is a little trickier. We may already know that the complexity of finding the minimum value from an array of size n is O(n) because we cannot avoid checking every value in the array. We might reason that there is a loop that goes from 1 to n in the algorithm, and our findMinIndex should also be O(n). This idea leads us to think that calling an O(n) function n times would lead to O(n2). Is this correct? How can we be sure? Toward the end of the algorithms execution, we are only looking for the minimum values index from among 3, 2, or 1 values. This seems close to O(1) or constant time. Calling an O(1) function n times would lead to O(n), right? Practicing this type of reasoning and asking these questions will help develop your algorithm analysis skills. These are both reasonable arguments, and they have helped establish a bound for our algorithms complexity. It would be safe to assume that the actual runtime is somewhere between O(n) and O(n2). Let us try to tackle this question more rigorously.When our algorithm begins, nothing is in sorted order (assume a random ordering). Our index from line 3 of selectionSort starts at 0. Next, findMinIndex searches all n elements from 0 to n 1. Then we have the smallest value in position 0, and index becomes 1. With index 1, findMinIndex searches n 1 values from 1 to n 1. This continues until index becomes n 1 and the algorithm finishes with all values sorted.We have the following pattern:With index at 0 n comparison operations are performed by findMinIndex.With index at 1 n 1 comparison operations are performed by findMinIndex.With index at 2 n 2 comparison operations are performed by findMinIndex.With index at n 2 2 comparison operations are performed by findMinIndex.With index at n 1 1 comparison operation is performed by findMinIndex.Our runtime is represented by the sum of all these operations. We could rewrite this in terms of the sum over the number of comparison operations at each step:n + (n 1) + (n 2) + 3 + 2 + 1.Can we rewrite this sequence as a function in terms of n to give the true runtime? One way to solve this sequence is as follows:Let S = n + (n 1) + (n 2) + 3 + 2 + 1.Multiplying S by 2 gives2 * S=[n + (n 1) + (n 2) + + 3 + 2 + 1] + [n + (n 1) + (n 2) + + 3 + 2 + 1].We can rearrange the right-hand side to highlight a useful pattern:2 * S=[n + (n 1) + (n 2) ++3+2+1]+[1 +2+3++ (n 2)+ (n 1) +n].We notice that lining them up with one sequence reversed leads to n terms of n + 1:2 * S=(n + 1) + (n + 1) + (n + 1) + + (n + 1) + (n + 1) + (n + 1)=n * (n + 1).Now we can divide by two to get an exact function for this summation sequence, which is also known as a variant of the arithmetic series: To view it in polynomial terms, we can distribute the n term and move the fraction:S=()*[n2 + n]=() n2 + () n.In Big-O terms, the time complexity of Selection Sort is O(n2). This is also known as quadratic time.Insertion SortInsertion Sort is another classic sorting algorithm. Insertion Sort orders values using a process like organizing books on a bookshelf starting from left to right. Consider the following shelf of unorganized books:Figure 3.4Currently, the books are not organized alphabetically. Insertion Sort starts by considering the first book as sorted and placing an imaginary separator between the sorted and unsorted books. The algorithm then considers the first book in the unsorted portion.Figure 3.5According to the image, the algorithm is now considering the book called Linear Algebra. The algorithm will now try to place this new book into its proper position in the sorted section of the bookshelf. The letter C comes before L, so the book should be placed to the right of the Calculus book, and the algorithm will consider the next book. This state is shown in the following image:Figure 3.6Now we consider the Algorithms book. In this case, the Algorithms book should come before both the Calculus and Linear Algebra books, but there is no room on the shelf to just place it there. We must make room by moving the other books over.The actual process used by the algorithm considers the book immediately to the left of the book under consideration. In this case, the Linear Algebra book should come after the Algorithms book, so the Linear Algebra book is moved over one position to the right. Next, the Calculus book should come after the Algorithms book, and the Calculus book is moved one position to the right. Now there are no other books to reorder, so the Algorithms book is placed in the correct position on the shelf. This situation is highlighted in the following image:Figure 3.7After adjusting the position of these books, we have the state displayed below:Figure 3.8Now we consider the textbook on Discrete Mathematics. First, we examine the Linear Algebra book to its left. The Linear Algebra book should go after the Discrete Mathematics book, and it is moved to the right by one position. Now examining the Calculus book informs us that no other sorted books should be after the Discrete Mathematics book. We will now place the Discrete Mathematics book in its correct place after the Calculus book. The Algorithms book at the far left is not even examined. This process is illustrated in the figure below:Figure 3.9Once the Discrete Mathematics book is placed in its correct position, the process begins again, considering the next book as shown below:Figure 3.10This brief illustration should give you an idea of how the Insertion Sort algorithm works. We consider a particular book and then insert it into the correct sorted position by moving books that come after it to the right by one position. The sorted portion will grow as we consider each remaining book in the unsorted portion. Finally, the unsorted section of the bookcase will be empty, and all the books will be sorted properly. This example also illustrates that there are other types of ordering. Numerical ordering and alphabetical ordering are probably the most common orderings you will encounter. Date and time ordering are also common, and sorting algorithms will work just as well with these as with other types of orderings.Insertion Sort ImplementationFor our implementation of Insertion Sort, we will only consider arrays of integers as with Selection Sort. Specifically, we will assume that the values of positions in the array are comparable and will lead to the correct ordering. The process will work equally well with alphabetical characters as with numbers provided the relational operators are defined for these and other orderable types. We will examine a way to make comparisons more flexible later in the chapter.This algorithm relies on careful manipulation of array indexes. Manipulation of array indexes often leads to errors as humans are rarely careful. As always, you are encouraged to test your algorithms using different types of data. Let us test our implementation using the array from before. We will do this by using a trace of an algorithm. A trace is just a way to write out or visualize the sequence of steps in an algorithm. Consider the following array with indexes:Figure 3.11The algorithm will start as follows with endSorted set to 1 and end set to 8. Entering the body of the while-loop, we set currentValue to the value 27, and index is set to endSorted, which is 1. The image below gives an illustration of this scenario:Figure 3.12We now consider the inner loop of the algorithm. The compound condition of index > 0 and currentValue < array[index 1] is under consideration. Index is greater than 0, check! This part is true. The value in array[index 1] is the value at array[1 1] or array[0]. This value is 43. Now we consider 27 < 43, which is true. This makes the compound condition of the while-loop true, so we enter the loop. This executes set array[index] to array[index 1], which copies 43 into position array[index] or array[1]. Next, the last command is executed to update index to index 1 or 0. This gives us the following state:Figure 3.13You may be thinking, What about 27? Is it lost? No, the value of 27 was saved in the currentValue variable. We now loop and check index > 0 and currentValue < array[index 1] again. This time index > 0 fails with index equal to 0. The algorithm then executes set array[index] to currentValue. This operation places 27 into position 0, providing the correct order. Finally, the endSorted variable is increased by 1 and now points to the next value to consider. At the start of the next loop, after currentValue and index are set, we have the state of execution presented below:Figure 3.14Next, the algorithm would check the condition of the inner loop. Here index is greater than 0, but currentValue is not less than array[index 1], as 45 is not less than 43. Moving over the loop, currentValue is placed back into its original position (line 12), and the indexes are updated at the end of the loop as usual. Moving to the next value to consider, before line 7 we now have the scenarios presented below:Figure 3.15From this image, we can imagine what would happen next. The value 24 is smaller than all these values. Let us trace how the algorithm would proceed. As 24 is less than 45, the body will execute copying 45 to the right and updating the index.Figure 3.16In the above image, we notice that 24 is less than 43. Therefore, we copy and update our indexes. This gives the following successive states of execution:Figure 3.17ThenFigure 3.18Now with index 0, the inner loops body will not execute. We will copy the currentValue into the index position and begin the outer loops execution again. At the start of the next inner loop, we have the scene below:Figure 3.19These illustrations give you an idea of the execution of the Insertion Sort. These drawings are sometimes called traces. Creating algorithm tracing will give a good idea of how your algorithm is working and will also help you understand if your algorithm is correct or not. From these diagrams, we can infer that the endSorted value will grow to reach the end of the array and all the values will eventually be properly sorted. You should attempt to complete the rest of this trace for practice.Insertion Sort ComplexityFrom this example execution, you may have noticed that sometimes Insertion Sort does a lot of work, but other times it seems that very little needs to be done. This observation allows us to consider a different way to analyze algorithmsnamely, the best-case time complexity. Before we address this question, let us analyze the worst-case space complexity and the worst-case time complexity of Insertion Sort.The space complexity of Insertion Sort should be easy to determine. We only need space for the array itself and a few other index- and value-storage variables. This means that our memory usage is O(n) for the array and a small constant number of other values where c numbersToSort) { int i; int range = upperBound - lowerBound; std::vector < int > counts(range + 1); std::fill(counts.begin(), counts.end(), 0); std::vector < int > storedNumbers(numbersToSort.size()); std::fill(storedNumbers.begin(), storedNumbers.end(), 0); for (i = 0; i < numbersToSort.size(); i++) { int index = numbersToSort[i] - lowerBound; counts[index] += 1; } for (i = 1; i < counts.size(); i++) { counts[i] += counts[i - 1]; } for (i = numbersToSort.size() - 1; i >= 0; i--) { storedNumbers[--counts[numbersToSort[i] - lowerBound]] = numbersToSort[i]; } for (i = 0; i < storedNumbers.size(); i++) { std::cout 1 => [3 5 1 8 4 2] 5 > 1 => [3 1 5 8 4 2] 3 > 1 => [1 3 5 8 4 2] Result: [1 3 5 8 4 2]Step 4 : key = 4 8 > 4 => [1 3 5 4 8 2] 5 > 4 => [1 3 4 5 8 2] 3 > 4 > stop Result: [1 3 4 5 8 2]Step 5 : key = 2 8 > 2 => [1 3 4 5 2 8] 5 > 2 => [1 3 4 2 5 8] 4 > 2 => [1 3 2 4 5 8] 3 > 2 => [1 2 3 4 5 8] 1 > 2 > stop Result: [1 2 3 4 5 8]The algorithm shown below is a slightly optimized version to avoid swapping the key element in every iteration. Here, the key element will be swapped at the end of the iteration (step). InsertionSort(arr[]) for j = 1 to arr.length key = arr[j] i = j - 1 while i > 0 and arr[i] > key arr[i+1] = arr[i] i = i - 1 arr[i+1] = keyHere is a detailed implementation in JavaScript:function insertion_sort(A) { var len = array_length(A); var i = 1; while (i < len) { var x = A[i]; var j = i - 1; while (j >= 0 && A[j] > x) { A[j + 1] = A[j]; j = j - 1; } A[j+1] = x; i = i + 1; }}A quick implementation in Swift is shown below: var array = [8, 3, 5, 1, 4, 2] func insertionSort(array:inout Array) -> Array{ for j in 0.. 0 && array[i] > key){ array[i+1] = array[i] i = i-1 } array[i+1] = key } return array }The Java example is shown below:public int[] insertionSort(int[] arr) for (j = 1; j < arr.length; j++) { int key = arr[j] int i = j - 1 while (i > 0 and arr[i] > key) { arr[i+1] = arr[i] i -= 1 } arr[i+1] = key } return arr;And in c....void insertionSort(int arr[], int n) { int i, key, j; for (i = 1; i < n; i++) { key = arr[i]; j = i-1; while (j >= 0 && arr[j] > key) { arr[j+1] = arr[j]; j = j-1; } arr[j+1] = key; } }Space Complexity: O(1)Time Complexity: O(n), O(n n), O(n n) for Best, Average, Worst cases respectively.Best Case: array is already sortedAverage Case: array is randomly sortedWorst Case: array is reversely sorted.Sorting In Place: YesStable: Yes Heapsort is an efficient sorting algorithm based on the use of max/min heaps. A heap is a tree-based data structure that satisfies the heap property that is for a max heap, the key of any node is less than or equal to the key of its parent (if it has a parent). This property can be leveraged to access the maximum element in the heap in O(logn) time using the maxHeapify method. We perform this operation n times, each time moving the maximum element in the heap to the top of the heap and extracting it from the heap and into a sorted array. Thus, after n iterations we will have a sorted version of the input array. The algorithm is not an in-place algorithm and would require a heap data structure to be constructed first. The algorithm is also unstable, which means when comparing objects with same key, the original ordering would not be preserved. This algorithm runs in O(nlogn) time and O(1) additional space [O(n) including the space required to store the input data] since all operations are performed entirely in-place.The best, worst and average case time complexity of Heapsort is O(nlogn). Although heapsort has a better worse-case complexity than quicksort, a well-implemented quicksort runs faster in practice. This is a comparison-based algorithm so it can be used for non-numerical data sets insofar as some relation (heap property) can be defined over the elements.An implementation in Java is as shown below :import java.util.Arrays;public class Heapsort { public static void main(String[] args) { Integer[] arr = {1, 4, 3, 2, 64, 3, 2, 4, 5, 5, 2, 12, 14, 5, 3, 0, -1}; String[] strarr = {"hope you find this helpful!", "wef", "rg", "q2rq2r", "avs", "erhijer0g", "ewofij", "gwe", "q", "random"}; arr = heapsort(arr); strarr = heapsort(strarr); System.out.println(Arrays.toString(arr)); System.out.println(Arrays.toString(strarr)); } public static E[] heapsort(E[] arr){ int heaplength = arr.length; for(int i = arr.length/2; i>0;i--){ arr = maxheapify(arr, i, heaplength); } for(int i=arr.length-1;i>=0;i--){ E max = arr[0]; arr[0] = arr[i]; arr[i] = max; heaplength--; arr = maxheapify(arr, 1, heaplength); } return arr; } public static E[] maxheapify(E[] arr, Integer node, Integer heaplength){ Integer left = node*2; Integer right = node*2+1; Integer largest = node; if(left.compareTo(heaplength) = 0){ largest = left; } if(right.compareTo(heaplength) = 0){ largest = right; } if(largest != node){ E temp = arr[node-1]; arr[node-1] = arr[largest-1]; arr[largest-1] = temp; maxheapify(arr, largest, heaplength); } return arr; }}Implementation in C++using namespace std;void heapify(int arr[], int n, int i) { int largest = i; int l = 2*i + 1; int r = 2*i + 2; if (l < n && arr[l] > arr[largest]) largest = l; if (r < n && arr[r] > arr[largest]) largest = r; if (largest != i) { swap(arr[i], arr[largest]); heapify(arr, n, largest); } } void heapSort(int arr[], int n) { for (int i = n / 2 - 1; i >= 0; i--) heapify(arr, n, i); for (int i=n-1; i>=0; i--) { swap(arr[0], arr[i]); heapify(arr, i, 0); } } void printArray(int arr[], int n) { for (int i=0; i item: return start else: return start + 1 if start > end: return start mid = round((start + end)/ 2) if the_array[mid] < item: return binary_search(the_array, item, mid + 1, end) elif the_array[mid] > item: return binary_search(the_array, item, start, mid - 1) else: return mid """Insertion sort that timsort uses if the array size is small or ifthe size of the "run" is small"""def insertion_sort(the_array): l = len(the_array) for index in range(1, l): value = the_array[index] pos = binary_search(the_array, value, 0, index - 1) the_array = the_array[:pos] + [value] + the_array[pos:index] + the_array[index+1:] return the_array def merge(left, right): """Takes two sorted lists and returns a single sorted list by comparing the elements one at a time. [1, 2, 3, 4, 5, 6] """ if not left: return right if not right: return left if left[0] < right[0]: return [left[0]] + merge(left[1:], right) return [right[0]] + merge(left, right[1:]) def timsort(the_array): runs, sorted_runs = [], [] length = len(the_array) new_run = [the_array[0]] for i in range(1, length): if i == length - 1: new_run.append(the_array[i]) runs.append(new_run) break if the_array[i] < the_array[i-1]: if not new_run: runs.append([the_array[i]]) new_run.append(the_array[i]) else: runs.append(new_run) new_run = [] else: new_run.append(the_array[i]) for item in runs: sorted_runs.append(insertion_sort(item)) sorted_array = [] for run in sorted_runs: sorted_array = merge(sorted_array, run) print(sorted_array) timsort([2, 3, 1, 5, 6, 7]) Tim sort has a stable Complexity of O(N log(N)) and compares really well with Quicksort. A comparison of complexities can be found on this chart. Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. The major portion of the algorithm is given two sorted arrays, and we have to merge them into a single sorted array. The whole process of sorting an array of N integers can be summarized into three steps-Divide the array into two halves.Sort the left half and the right half using the same recurring algorithm.Merge the sorted halves.There is something known as the Two Finger Algorithm that helps us merge two sorted arrays together. Using this subroutine and calling the merge sort function on the array halves recursively will give us the final sorted array we are looking for. Since this is a recursion based algorithm, we have a recurrence relation for it. A recurrence relation is simply a way of representing a problem in terms of its subproblems.T(n) = 2 * T(n / 2) + O(n)Putting it in plain English, we break down the subproblem into two parts at every step and we have some linear amount of work that we have to do for merging the two sorted halves together at each step. The biggest advantage of using Merge sort is that the time complexity is only n*log(n) to sort an entire Array. It is a lot better than n^2 running time of bubble sort or insertion sort.Before we write code, let us understand how merge sort works with the help of a diagram.Initially we have an array of 6 unsorted integers Arr(5, 8, 3, 9, 1, 2)We split the array into two halves Arr1 = (5, 8, 3) and Arr2 = (9, 1, 2).Again, we divide them into two halves: Arr3 = (5, 8) and Arr4 = (3) and Arr5 = (9, 1) and Arr6 = (2)Again, we divide them into two halves: Arr7 = (5), Arr8 = (8), Arr9 = (9), Arr10 = (1) and Arr6 = (2)We will now compare the elements in these sub arrays in order to merge them.Space Complexity: O(n)Time Complexity: O(n*log(n)). The time complexity for the Merge Sort might not be obvious from the first glance. The log(n) factor that comes in is because of the recurrence relation we have mentioned before.Sorting In Place: No in a typical implementationStable: YesParallelizable :yes (Several parallel variants are discussed in the third edition of Cormen, Leiserson, Rivest, and Stein's Introduction to Algorithms.) void merge(int array[], int left, int mid, int right){ int i, j, k; int size_left = mid - left + 1; int size_right = right - mid; int Left[size_left], Right[size_right]; for(i = 0; i < size_left; i++){ Left[i] = array[left+i]; } for(j = 0; j < size_right; j++){ Right[j] = array[mid+1+j]; } i = 0; j = 0; k = left; while (i < size_left && j < size_right){ if (Left[i] 0 && b.length >0) result.push(a[0] < b[0]? a.shift() : b.shift()); return result.concat(a.length? a : b);}When we merge the two halfs, we store the result in an auxilliary array. We will compare the starting element of left array to the starting element of right array. Whichever is lesser will be pushed into the results array and we will remove it from there respective arrays using [shift() operator. If we still end up with values in either of left or right array, we would simply concatenate it in the end of the result. Here is the sorted result:var test = [5,6,7,3,1,3,15];console.log(mergeSort(test)); >> [1, 3, 3, 5, 6, 7, 15] Here's a good YouTube video that walks through the topic in detail. const list = [23, 4, 42, 15, 16, 8, 3] const mergeSort = (list) =>{ if(list.length { var result = []; while(left.length || right.length) { if(left.length && right.length) { if(left[0] < right[0]) { result.push(left.shift()) } else { result.push(right.shift()) } } else if(left.length) { result.push(left.shift()) } else { result.push(right.shift()) } } return result;} console.log(mergeSort(list)) void merge(int arr[], int l, int m, int r) { int i, j, k; int n1 = m - l + 1; int n2 = r - m; int L[n1], R[n2]; for (i = 0; i < n1; i++) L[i] = arr[l + i]; for (j = 0; j < n2; j++) R[j] = arr[m + 1+ j]; i = 0; j = 0; k = l; while (i < n1 && j < n2) { if (L[i]

- nanopeta
- dna fingerprinting worksheet key
- http://www.css-jp.com/upfile/files/2025/07/12/19041959448.pdf