Discussion-6: Understanding and Analyzing Sorting Algorithms

Discussion Topic:

In this module, we focus our attention on sorting algorithms in Java. Keep in mind that there are a number of sorting algorithms that can be used on a list of objects. Implement a sorting algorithm of your choice and attach your working program with screenshots. What are some factors to consider when determining which sorting algorithm would be best to utilize?

In your answer, specifically think of and give a real-life scenario where:

A given sorting algorithm is used
One algorithm outperforms the other

My Post:

Hello class,

In computing science, different types of sorting algorithms are used, see Table 1 and Table 2. They are a fundamental component for organizing data, which is often a prerequisite step before the data can be processed, searched, or optimized for algorithms in various applications uses.

Table 1

Comparative Sorting Algorithms

Algorithm	Technique	Best Used For	How It Works	Time Complexity	Space Complexity
Bubble Sort	Brute Force	Basic small datasets	Compares each element with every other and swaps if they are not in the correct order.	$O(n^2)$	O(1) - In-place
Merge Sort	Divide and Conquer	Merging two or more sorted input lists	Recursively divides the input data in half until each part is of size 1, then merges the sorted parts back together.	O(n log n)	O(n) - Extra space
Insertion Sort	Insertion-based	Small datasets, preserve insertion order	Removes an element from the input list and inserts it into the correct position in the already sorted part of the list.	$O(n^2)$	O(1) - In-place
Quick Sort	Divide and Conquer	General large datasets	Selects a pivot, splits the array into parts based on the pivot, and recursively sorts each part.	O(n log n)	O(1) - In-place
Selection Sort	Selection-based	Small datasets	Finds the minimum value in the list and swaps it with the current index, repeating until the list is sorted.	$O(n^2)$	O(1) - In-place

Heap Sort	Divide and Conquer	Priority queues, ensuring min/max element is at top	Inserts elements into a heap (minHeap or maxHeap) and removes elements until the heap is empty, ensuring min/max values are always at the root.	O(n log n)	O(n)

Note: In Java comparative sorting of both primitive data (via algorithms) and user-defined data is implemented through the Comparable or Comparator interfaces. From “Sorting Cheat Sheet” by evanescesn09 (2019). Modify.

Table 2

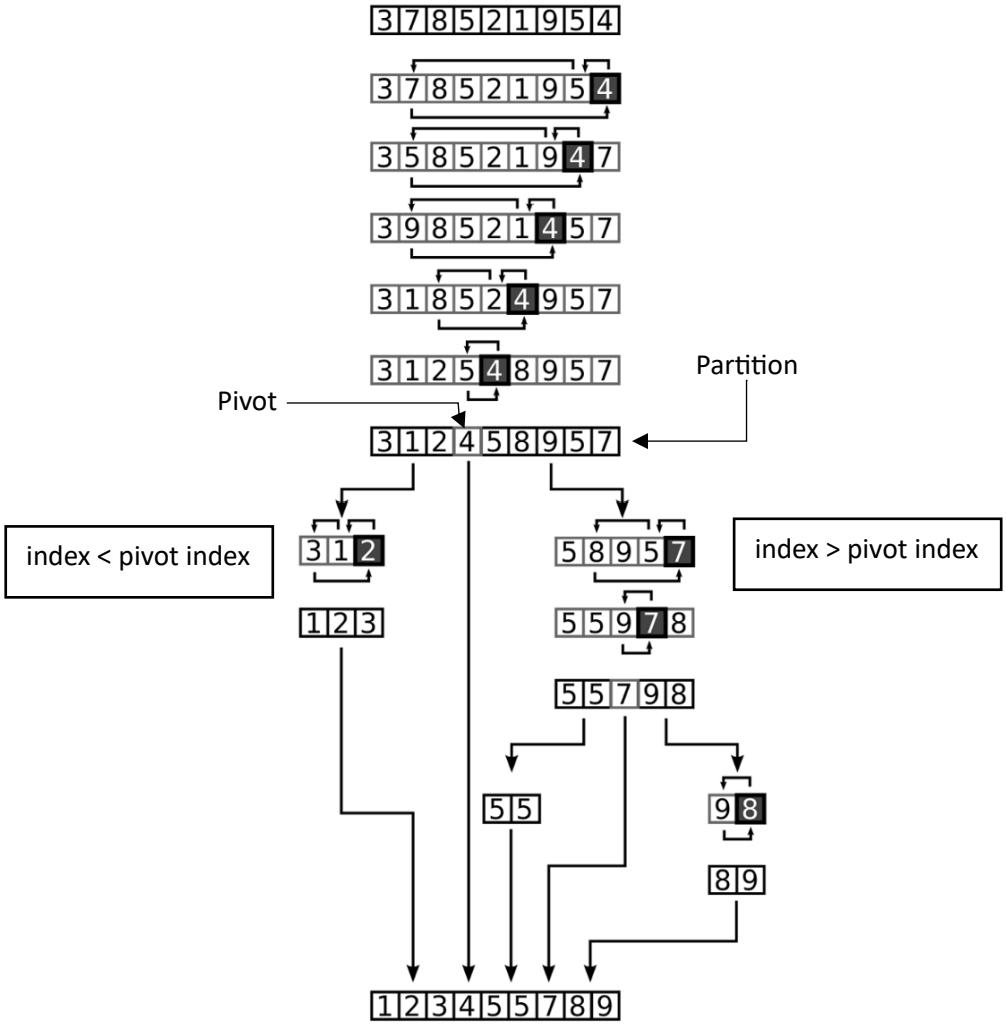
Non-Comparative Sorting Algorithms

Algorithm	Technique	Best Used For	How It Works	Time Complexity	Space Complexity
Radix Sort	Non-Comparison, Digit-Based	Large datasets of fixed-width integers or strings	Sorts data digit by digit, starting from the least significant digit to the most significant, using Counting Sort as a subroutine for each digit.	O(d*(n + k)) where d is number of digits, n is number of elements, and k is the range of digits	O(n + k) - Extra space
Counting Sort	Non-Comparison, Counting	Small range of integer data (e.g., grades, ages)	Counts the occurrences of each element, then uses this information to place elements in their correct positions in the output array.	O(n + k), where n is the number of elements and k is the range of input values	O(n + k) - Extra space
Bucket Sort	Non-Comparison, Bucket-Based	Uniformly distributed floating-point numbers or integers	Divides data into a fixed number of "buckets", sorts the individual buckets (usually using another sorting algorithm like Insertion Sort), and then concatenates the sorted buckets.	O(n + k), where n is the number of elements and k is the number of buckets	O(n + k) - Extra space

Below is an example of implementing a Quick Sort algorithm. This algorithm is often implemented by E-commerce businesses to filter products by price, viewing items from the lowest to highest cost or vice versa. Quick Sort is well suited for this kind of scenario, especially when dealing with a large number of products.

Figure 1

Quick Sort



Note: From "Data Structures and Algorithms Cheat Sheet" by Elhannat (n.d.) Modify.

The code below is an implementation in Java of a Quick Sort algorithm sorting prices in ascending order.

QuickSortPrices.java

```java
/**
 * Implements a QuickSort algorithm for sorting an array of prices (doubles)
 */
public class QuickSortPrices {

	// ------------------------------------------------------------------------------------
	/*----------------------
	 |  Partition Method  |
	 ----------------------*/

	/**
	 * Divides the array into two parts: one part with elements smaller than or
	 * equal to the pivot and another with elements larger than the pivot elements
	 * smaller than or equal to the pivot are moved to the left of elements larger
	 * than the pivot are moved to the right
	 *
	 * @param prices The array of doubles to be partitioned
	 * @param low    The starting index of the portion of the array to be
	 *               partitioned
	 * @param high   The ending index of the portion of the array to be partitioned.
	 * @return The index of the pivot element after partitioning
	 */
	public static int partition(double[] prices, int low, int high) {
		// The last element in the array becomes the pivot
		double pivot = prices[high];
		// i keeps track of the index where smaller elements than the pivot should go
		// set to low - 1, which means no smaller element has been found yet
		int i = low - 1;

		// Iterate from the 'low' index to 'high - 1' (excluding the pivot)
		for (int j = low; j < high; j++) {
			// If smaller than or equal to the pivot, it is moved
			// to the part of the array that holds smaller elements
			if (prices[j] <= pivot) {
				i++;

				// Swap prices[i] and prices[j]
				double temp = prices[i];
				prices[i] = prices[j];
				prices[j] = temp;
			}
		}

		// Swap the pivot element (prices[high]) with the element at i+1
		double temp = prices[i + 1];
		prices[i + 1] = prices[high];
		prices[high] = temp;

		// Return the index of the pivot element. This index is used to divide the array
		// into two parts to implement further recursive sort
		return i + 1;
	}

	// ------------------------------------------------------------------------------------
	/*------------------------
	 |  QuickSort algorithm  |
	 ------------------------*/
```

```
	/**
	 * QuickSort algorithm is a recursive method that sorts an array by dividing
	 * using a pivot
	 *
	 * @param prices The array of doubles to be sorted
	 * @param low    The starting index of the portion of the array to be sorted
	 * @param high   The ending index of the portion of the array to be sorted
	 */
	public static void quickSort(double[] prices, int low, int high) {

		int pivotIndex; // Stores the pivot index

		// Check if the portion if low is less than high
		if (low < high) { // ----- Base case low is not < than high ------

			// ----- Recursive Base -----

			// Divides the array around a pivot
			// Returns the index of the pivot element
			pivotIndex = partition(prices, low, high);

			// --- Recursion call to < than the pivot
			// Recursive call to a subarray of elements that are smaller than th
			// pivot
			quickSort(prices, low, pivotIndex - 1); // Once low is not < than high,
													// it exists and the next
													// recursion call to > than the
													// pivot

			// --- Recursion call to > than the pivot
			// Recursively apply quicksort on the subarray of elements that are
			// larger than the pivot
			quickSort(prices, pivotIndex + 1, high);
		}
	}

	// ---------------------------------------------------------------------------
}
```

Main.java

```
import java.util.Arrays;

public class Main {
	public static void main(String[] args) {
		// Example of product prices
		double[] productPrices = { 199.99, 49.99, 349.95, 5.99, 20.00, 89.99, 150.00 };

		System.out.println("\nProduct prices before sorting: " +
				Arrays.toString(productPrices));

		QuickSortPrices.quickSort(productPrices, 0, productPrices.length - 1);

		System.out.println("Product prices after sorting: " +
				Arrays.toString(productPrices));
	}

}
```

Output

```
Product prices before sorting: [199.99, 49.99, 349.95, 5.99, 20.0, 89.99, 150.0]
Product prices after sorting: [5.99, 20.0, 49.99, 89.99, 150.0, 199.99, 349.95]
```

Quick Sort is efficient with large datasets, its time complexity is *O(n log n)* which is better than algorithms such as Bubble Sort or Selection Sort which have a time complexity of $O(n^2)$. Additionally, it operates in-place, meaning that it uses minimal memory *O(1),* which is crucial when handling large datasets.

No one algorithm fits all applications. For example, in the shipping industry where linked lists are used, even though having the same time complexity as Quick Sort, *O(n log n)*, Merge Sort outperformed it. This is because Merge Sort is more suited for linked lists, as it doesn't require random access to elements, unlike Quick Sort, which relies on indexes to access elements.

Another example is small and nearly sorted data. In this senior insertion is better suited as its time complexity is in a best-case situation *O(n)* when the data is already sorted. However, in unsorted large data sets its worst-case time complexity is O(n^2).

To summarize, in computing science, different types of sorting algorithms are used. No one algorithm fits all applications and the choice of which sorting algorithm to use depends on factors such as dataset size, data structure, and whether the data is already partially sorted.

-Alex

References:

Elhannat, K. (n.d.). Data structures and slgorithms cheat sheet. Zero to Mastery. https://zerotomastery.io/cheatsheets/data-structures-and-algorithms-cheat-sheet/#contents

evanescesn09 (2019, August 17). Sorting cheat sheet [PDF]. Cheatography. https://cheatography.com/evanescesn09/cheat-sheets/sorting/pdf/?last=1566081837