Sorting Algorithms And Their Implementation

Chapter 1 - Introduction

Defined as an ordered set of unambiguous, executable steps, algorithms provide a systematic approach to solving problems. They must be clear and well-defined, leaving no room for ambiguity or interpretation. Additionally, algorithms must guarantee termination, meaning they will eventually reach an endpoint or solution, even if it may take some time. Reasoning behind this paper is to deliver information about specifically sorting algorithms. What makes sorting algorithms unique are their approaches to organizing data and **their efficiency in terms of time complexity and space complexity**. Different sorting algorithms may perform differently depending on factors such as the size of the dataset, the distribution of data, and the specific requirements of the application.

Chapter 2 - Algorithm Efficiency

Algorithm efficiency refers to how well an algorithm solves a problem concerning its use of resources like time, memory, and processing power. It's crucial to consider efficiency when designing algorithms, especially for larger or more complex problems, as it impacts performance and scalability.

Importance of Efficiency

Efficient algorithms are vital for various reasons:

- **Speed:** Faster algorithms mean quicker processing, crucial in time-sensitive applications.

- **Scalability:** As input size grows, efficient algorithms maintain reasonable performance.
- **Resource Utilization:** Efficient algorithms consume fewer resources, minimizing costs and energy usage.
- **User Experience:** Faster algorithms result in a better user experience, especially in software applications.

Factors Impacting Efficiency are:

1. **Algorithm Design:** Different algorithms can solve the same problem but with different efficiencies. For instance, sorting algorithms like Bubble Sort vs. Merge Sort have different time complexities (O(n^2) vs. O(n log n)).
2. **Input Size:** Larger inputs may impact an algorithm's performance differently. Some algorithms might handle small inputs efficiently but struggle with larger ones.
3. **Data Structures:** Efficient data structures (arrays, trees, hash tables) can significantly impact algorithm performance. Choosing the right structure for the problem is crucial.
4. **Optimizations:** Techniques like memoization, dynamic programming, or pruning can optimize algorithms and improve efficiency.
5. **Hardware and Environment:** The efficiency might also depend on the hardware and environment where the algorithm runs. Some algorithms perform better on specific architectures.

Time Complexity:

This measures how the algorithm's execution time grows with the input size. Common notations like O(n), O(n^2), O(log n), etc., describe how the algorithm scales concerning the input size `n`.

- **O(1) - Constant Time:** Operations take the same time regardless of input size.

- **O(n) - Linear Time:** Time increases linearly with input size.
- **O(n^2) - Quadratic Time:** Time grows with the square of the input size.
- **O(log n) - Logarithmic Time:** Time increases logarithmically relative to the input size.

Space Complexity:

This measures the amount of memory an algorithm uses concerning the input size. It's important to manage memory usage, especially in constrained environments.

CHAPTER 3 - THE ALGORITHMS

Now that the method and reasoning behind the paper is clear, let's start with an introduction and explanation of each of the nine sorting algorithms. The first sorting algorithm is affectionately named the "bubble" sort. This sort is perhaps one of the simplest sorts in terms of complexity. It makes use of a sorting method known as the exchange method. This algorithm compares pairs of adjacent elements and makes exchanges if necessary. The name comes from the fact that each element "bubbles" up to it's own proper position. Here is how bubble sort would sort the integer array {4 3 1 2}:

```
- pass 1: {1 4 3 2}
- pass 2: {1 2 4 3}
- pass 3: {1 2 3 4}
```

The code for this algorithm is located under the algorithm codes heading.

The second algorithm uses a different method of sorting known as sorting by selection. This 'selection sort algorithm picks the

smallest element from the array and switches it with the first element. It then picks the smallest element from the rest of the array and switches it with the second element. This process is repeated up to the last pair of elements. Here is how
it would sort {2 4 1 3}:

- pass 1: {1 4 2 3}
- pass 2: {1 2 4 3}
- pass 3: {1 2 3 4}

The code for this algorithm is located under the algorithm codes heading.

The third algorithm uses the insertion method of sorting. This algorithm first sorts the first two elements of the array. It then inserts the third element in it's proper place in relation to the first 2 sorted elements. This process continues until all of the remaining elements are inserted in their proper
position. Here is how it would sort {4 3 1 2}:

- pass 1: {3 4 1 2}
- pass 2: {1 3 4 2}
- pass 3: {1 2 3 4}

The code for this algorithm is located under the algorithm codes heading.

It has been suggested that the next algorithm is the best sorting algorithm available today. It is named quick sort due to it's speedy sort time. Quick sort is based on the exchange method of sorting, as is bubble sort, but is also uses the idea of partitioning. Quick sort chooses a median value from the array and uses it to partition the array into two subarrays. The left subarray contains all of the elements

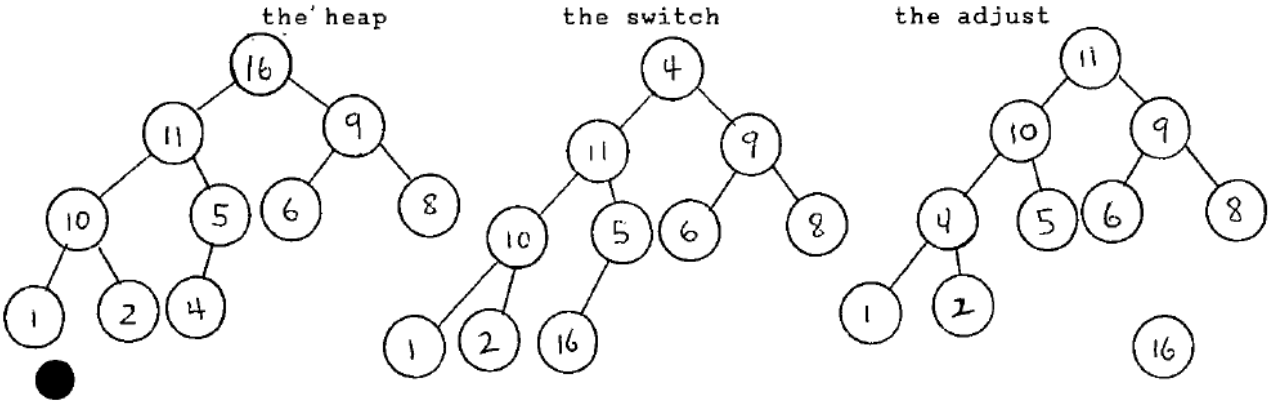
that are less than the median value and the right subarray contains all of the elements that are greater than the median value. This process is recursively repeated for each subarray until the array is sorted. The median value can be chosen randomly and this brings up one nasty aspect of quick sort. If the median value chosen is always the smallest or largest element, the algorithm slows down drastically. This usually will not happen however, since most input data is in a random order and the chance of always picking an extreme value is small. Note that this algorithm is naturally recursive and I have coded it as such. Here is how quick sort would sort {6 5 4 1 3 2}:

```
- pass 1: {2 3 1 4 5 6}
- pass 2: {1 2 3 4 5 6}
```

Note that after pass 1 , the array is partitioned into 2 3 1 and 4 5 6. The process is then repeated for each of these. Also the code for this algorithm is located under the algorithm codes heading.

The fifth algorithm is known as heap sort. It makes
use of a data structure known as a heap. A heap is a complete binary tree organized such that the value of the parent nodes are greater than their children's. With this type of structure, the largest element happens to be the root node. This property of a
heap makes it ideal for a sorting algorithm.
The heap sort algorithm first builds a heap with all of the elements. After heap creation, the largest element is located at the root. This root element is switched with the last element at
the end of the array. Since the root, which is the largest element, is placed at the end of the array, this largest element is now in it's correct position. This position in the array is now off-limits to the algorithm and the rest of the heap is adjusted, starting at the new root, to ensure that all of the

parent's values are greater that the values of their children. This switching and readjusting is performed repeatedly until all of the elements are in their proper position. Here is one pass of heap sort on 16 11 9 10 5 6 8 1 2 4:



Also the code for heap sort is located under the algorithm codes section.

The next algorithm is an example of divide and conquer. It is called merge sort. Merge sort splits the array into two subarrays, each of almost equal size, and recursively sorts each.
The two sorted subarrays are then merged together. The recursive version is very simple and takes full advantage of the power of recursion. Here is an example of how merge sort would sort 5 2 3 1 7, (the [ ] bars indicate a subarray):

```
[5 2 3][1 7]
[5 2][3]
[5][2]
[2 5]
[2 3 5]
[1][7]
[1 7]
[2 3 5 7]
```

Notice how the subarrays are broken down until there is only one element left. Then, two subarrays with only one element each are merged. The merging continues until all left side sorted arrays are merged together. The right side is then broken down and merged. This is a very good example of how recursion can be used to simplify programs. The code for this sort is located under the algorithms code section.

Algorithm Codes

When discussing algorithms like these, it's important to note that these algorithms are not tied to any specific programming language. Algorithms are abstract concepts that can be implemented in various programming languages, each with its syntax and features. In this example, we've demonstrated them using Java, a widely-used programming language known for its simplicity, portability, and extensive libraries.

Implementations for the bubble sort:

```
public class BubbleSortDemo {
    public static void main(String[] args) {
        int[] array = {4, 3, 1, 2};
        System.out.println("Original Array: ");
        printArray(array);
        bubbleSort(array);
        System.out.println("Sorted Array: ");
        printArray(array);
    }

    public static void bubbleSort(int[] arr) {
        int n = arr.length;
        for (int i = 0; i < n-1; i++) {
            for (int j = 0; j < n-i-1; j++) {
```

```
                if (arr[j] > arr[j+1]) {
                    int temp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = temp;
                }
            }
        }
    }

    public static void printArray(int[] arr) {
        for (int num : arr) {
            System.out.print(num + " ");
        }
        System.out.println();
    }
}
```

Implementations for the selection sort:

```
public class SelectionSortDemo {
    public static void main(String[] args) {
        int[] array = {2, 4, 1, 3};
        System.out.println("Original Array:");
        printArray(array);
        selectionSort(array);
        System.out.println("Sorted Array:");
        printArray(array);
    }

    public static void selectionSort(int[] arr) {
        int n = arr.length;
        for (int i = 0; i < n - 1; i++) {
```

```
            int minIndex = i;
            for (int j = i + 1; j < n; j++) {
                if (arr[j] < arr[minIndex]) {
                    minIndex = j;
                }
            }
            if (minIndex != i) {
                int temp = arr[i];
                arr[i] = arr[minIndex];
                arr[minIndex] = temp;
            }
        }
    }

    public static void printArray(int[] arr) {
        for (int num : arr) {
            System.out.print(num + " ");
        }
        System.out.println();
    }
}
```

Implementations for the insertion sort:

```
public class InsertionSortDemo {
    public static void main(String[] args) {
        int[] array = {4, 3, 1, 2};
        System.out.println("Original Array:");
        printArray(array);
        insertionSort(array);
        System.out.println("Sorted Array:");
        printArray(array);
```

```
    }

    public static void insertionSort(int[] arr) {
        int n = arr.length;
        for (int i = 1; i < n; i++) {
            int key = arr[i];
            int j = i - 1;
            while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                j = j - 1;
            }
            arr[j + 1] = key;
        }
    }

    public static void printArray(int[] arr) {
        for (int num : arr) {
            System.out.print(num + " ");
        }
        System.out.println();
    }
}
```

Implementations for the insertion sort:

```
public class QuickSortDemo {
    public static void main(String[] args) {
        int[] array = {6, 5, 4, 1, 3, 2};
        System.out.println("Original Array:");
        printArray(array);
        quickSort(array, 0, array.length - 1);
        System.out.println("Sorted Array:");
```

```
        printArray(array);
    }

    public static void quickSort(int[] arr, int low, int
high) {
        if (low < high) {
            int pi = partition(arr, low, high);

            quickSort(arr, low, pi - 1);
            quickSort(arr, pi + 1, high);
        }
    }

    public static int partition(int[] arr, int low, int high)
{
        int pivot = arr[high];
        int i = (low - 1);
        for (int j = low; j < high; j++) {
            if (arr[j] < pivot) {
                i++;

                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }

        int temp = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp;

        return i + 1;
    }
```

```
    public static void printArray(int[] arr) {
        for (int num : arr) {
            System.out.print(num + " ");
        }
        System.out.println();
    }
}
```

Implementations for the heap sort:

```
public class HeapSortDemo {
    public static void main(String[] args) {
        int[] array = {16, 11, 9, 10, 5, 6, 8, 1, 2, 4};
        System.out.println("Original Array:");
        printArray(array);
        heapSort(array);
        System.out.println("Sorted Array:");
        printArray(array);
    }

    public static void heapSort(int[] arr) {
        int n = arr.length;
        for (int i = n / 2 - 1; i >= 0; i--) {
            heapify(arr, n, i);
        }
        for (int i = n - 1; i >= 0; i--) {
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;
            heapify(arr, i, 0);
        }
```

```
    }

    public static void heapify(int[] arr, int n, int i) {
        int largest = i;
        int left = 2 * i + 1;
        int right = 2 * i + 2;
        if (left < n && arr[left] > arr[largest]) {
            largest = left;
        }
        if (right < n && arr[right] > arr[largest]) {
            largest = right;
        }
        if (largest != i) {
            int swap = arr[i];
            arr[i] = arr[largest];
            arr[largest] = swap;
            heapify(arr, n, largest);
        }
    }

    public static void printArray(int[] arr) {
        for (int num : arr) {
            System.out.print(num + " ");
        }
        System.out.println();
    }
}
```