Report on Sorting Algorithms

Bhavya Piyush

22 August 2023

1 Introduction

Sorting is a fundamental operation in computer science, involving arranging a collection of items in a specific order. Sorting is crucial for various applications, such as data organization, searching, and optimizing algorithms. Different sorting algorithms use various techniques to achieve this task with varying levels of efficiency.

Key concepts related to sorting include:

Comparison-based Sorting:

Many sorting algorithms are comparison-based, meaning they compare elements using a specific comparison operation to determine their order. Examples of comparison operations include "greater than," "less than," and "equal to." Stability:

A sorting algorithm is stable if it maintains the relative order of equal elements in the sorted output as they were in the original input. Stability is important in scenarios where multiple sort passes are needed. In-Place Sorting:

An in-place sorting algorithm doesn't require additional memory for sorting. It rearranges elements within the original data structure. In-place sorting is memory-efficient but might require more complex operations. Adaptive Sorting:

An adaptive sorting algorithm takes advantage of existing order in the input to improve performance. For example, if a small portion of the input is already sorted, an adaptive algorithm might be faster. Time Complexity:

Sorting algorithms are analyzed based on their time complexity, which indicates the number of operations required as a function of the input size. Common time complexities include O(n^2)$, O(nlogn), andlineartime(O(n)).SpaceComplexity :$

Space complexity measures the additional memory required by an algorithm. Some algorithms require additional space for temporary storage, while others can sort in-place. Best, Average, and Worst-case Scenarios:

Sorting algorithms are evaluated for their performance in best-case (ideal input), average-case (random input), and worst-case (worst possible input) scenarios. Optimizations:

Many sorting algorithms have variations or optimizations that improve their efficiency for specific scenarios. These optimizations might focus on reducing comparisons, minimizing swaps, or adapting to different data distributions

2 Bubble Sort

Bubble Sort is a simple comparison-based sorting algorithm. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The process "bubbles up" the largest (or smallest) element to its correct position in each iteration.

2.1 Complexity Analysis:

Time Complexity:

Worst-case: $O(n^2)$ comparisons and swaps.

Best-case (when the list is already sorted): $O(n)$ comparisons and $O(1)$ swaps.

Space Complexity: $O(1)$ as it requires only a constant amount of extra space for temporary variables.

2.2 Advantages and Disadvantages:

Advantages: Simple to understand and implement. Works well for small input sizes or nearly sorted data.

Disadvantages: Inefficient for large input sizes due to its quadratic time complexity. Lacks adaptability to already sorted or partially sorted inputs.

2.3 Bubble Sort Algorithm:

- Start from the first element of the list.
- Compare the current element with the next element.
- If the current element is greater (for ascending order) than the next element, swap them.
- Move to the next pair of elements and repeat steps 2 and 3 until the end of the list.
- Repeat steps 1 to 4 for a total of $n - 1$ passes, where n is the number of elements in the list.

2.4 Sample Code (C Language)

```
for (int i = 0; i < n - 1; i++) {
    for (int j = 0; j < n - i - 1; j++) {
        if (arr[j] > arr[j + 1]) {
            // Swap arr[j] and arr[j+1]
            int temp = arr[j];
            arr[j] = arr[j + 1];
            arr[j + 1] = temp;
        }
    }
}
```

2.5 Basic Flowchart:

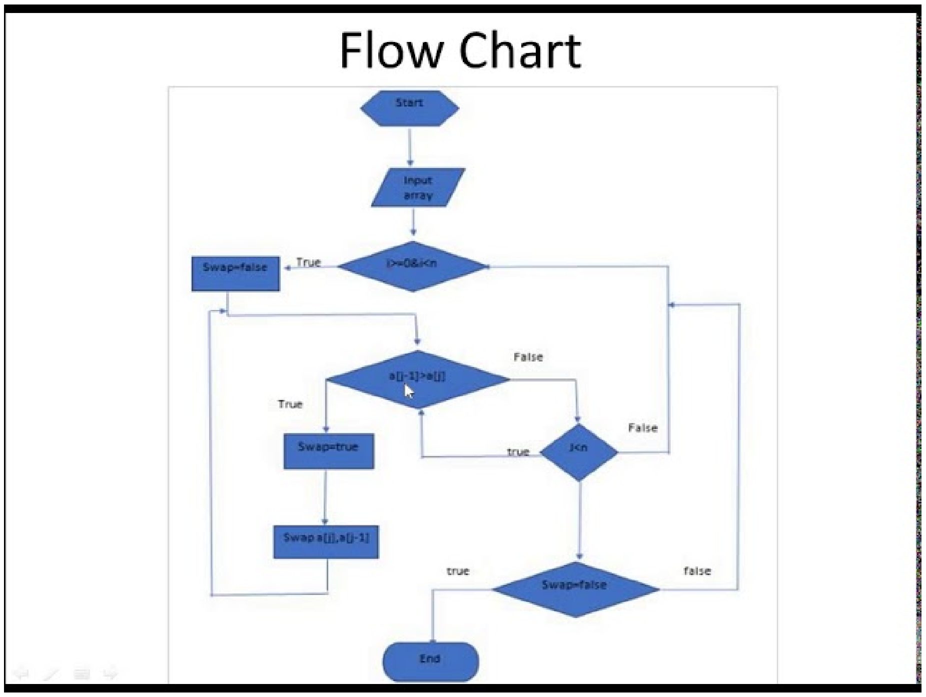


Figure 1: Bubble Sort Flowchart

Flowchart for Bubble Sort Algorithm:

Start: Begin the algorithm.

Initialize array: Initialize the array containing the elements to be sorted.

Loop i through 0 to n-1: Iterate over the entire array, from the first element to the second-to-last element.

Loop j through 0 to n-i-1: Within the outer loop, iterate over adjacent elements in the array.

Compare arr[j] and arr[j+1]: Compare the current element (arr[j]) with the next element (arr[j+1]).

If arr[j] ¿ arr[j+1], swap: If the current element is greater than the next element, swap them to ”bubble up” the larger element.

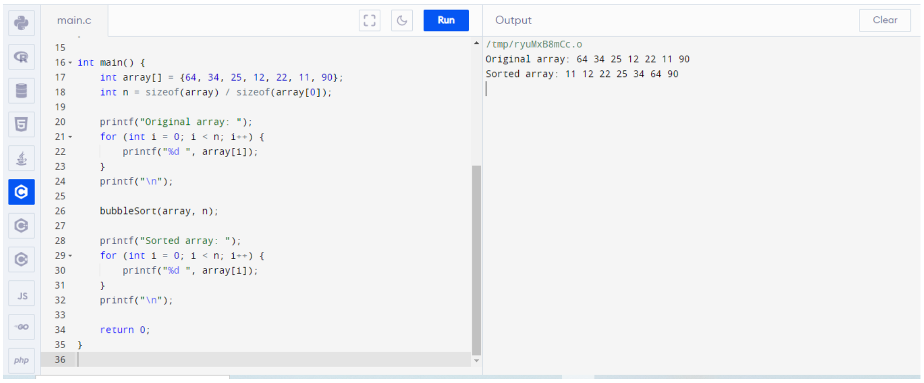
Inner loop end: Complete the inner loop, moving to the next iteration or exiting if all adjacent pairs have been checked.

Inner loop ends: After the inner loop completes, the largest unsorted element is in its correct position at the end of the array.

Outer loop end: Complete the outer loop, moving to the next iteration and reducing the length of the unsorted portion.

End: Finish the algorithm, with the entire array sorted in ascending order.

2.6 Output (C Language)



```c
int main() {
    int array[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(array) / sizeof(array[0]);

    printf("Original array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");

    bubbleSort(array, n);

    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");

    return 0;
}
```

```
/tmp/ryuMxB8mCc.o
Original array: 64 34 25 12 22 11 90
Sorted array: 11 12 22 25 34 64 90
```

Figure 2: Bubble Sort Output

2.7 Conclusion:

Bubble Sort is a straightforward sorting algorithm that compares and swaps adjacent elements until the entire list is sorted. While easy to understand, it's inefficient for larger datasets due to its quadratic time complexity. Bubble Sort is best suited for educational purposes or small datasets where simplicity outweighs efficiency.

3 Insertion Sort

3.1 Theory of Insertion Sort

Insertion Sort is a simple sorting algorithm that builds the final sorted array one item at a time. It takes an element from the unsorted part and inserts it into its correct position in the sorted part. The algorithm is likened to the way we sort playing cards in hand.

3.2 Complexity Analysis

Time Complexity:

- Worst-case: $O(n^2)$ comparisons and swaps.

- Best-case (when the list is already sorted): $O(n)$ comparisons and $O(1)$ swaps.

Space Complexity: $O(1)$ as it requires only a constant amount of extra space for temporary variables.

3.3 Advantages and Disadvantages

Advantages:

- Simple and easy to understand.
- Efficient for small datasets or partially sorted data.
- In-place sorting algorithm.

Disadvantages:

- Inefficient for large datasets due to quadratic time complexity.
- Not suitable for datasets with a random or reverse order.

3.4 Insertion Sort Algorithm

The algorithm involves iterating over the unsorted part of the list and moving elements to their correct position in the sorted part.

Insertion Sort [1] InsertionSortarr, n $i \leftarrow 1$ to $n-1$ $key \leftarrow arr[i]$ $j \leftarrow i-1$ $j \geq 0$ and $arr[j] > key$ $arr[j+1] \leftarrow arr[j]$ $j \leftarrow j-1$ $arr[j+1] \leftarrow key$

3.5 Sample Code(C language)

```
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;
}
```

3.6 Basic Flowchart

Flowchart for Insertion Sort Algorithm:

Start: Begin the algorithm.

Initialize array: Initialize the array containing the elements to be sorted.

Loop i through 0 to n-1: Iterate over the unsorted portion of the array, from the first element to the second-to-last element.

Loop j through 0 to n-i-1: Within the outer loop, iterate over the sorted portion of the array, from the first element to the last unsorted element.

Compare arr[j] and arr[j+1]: Compare the current element (arr[j]) with the next element (arr[j+1]).

If arr[j] ¿ arr[j+1], swap: If the current element is greater than the next element, swap them to maintain ascending order.

Inner loop end: Complete the inner loop, moving to the next iteration or exiting if the sorted portion is exhausted.

Inner loop ends: After the inner loop completes, move to the next iteration of the outer loop.

Outer loop end: Complete the outer loop, moving to the next unsorted element.

End: Finish the algorithm, with the entire array sorted in ascending order.

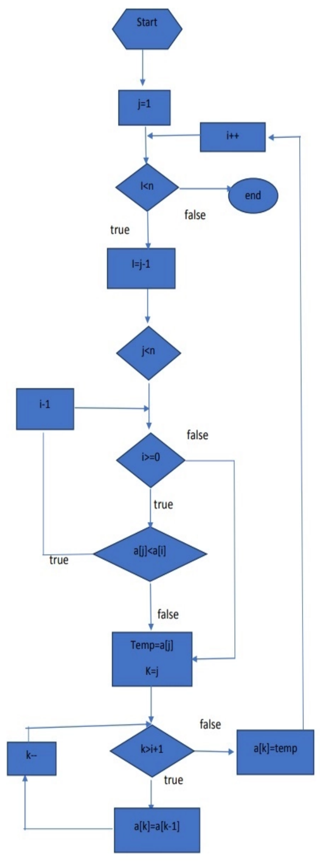
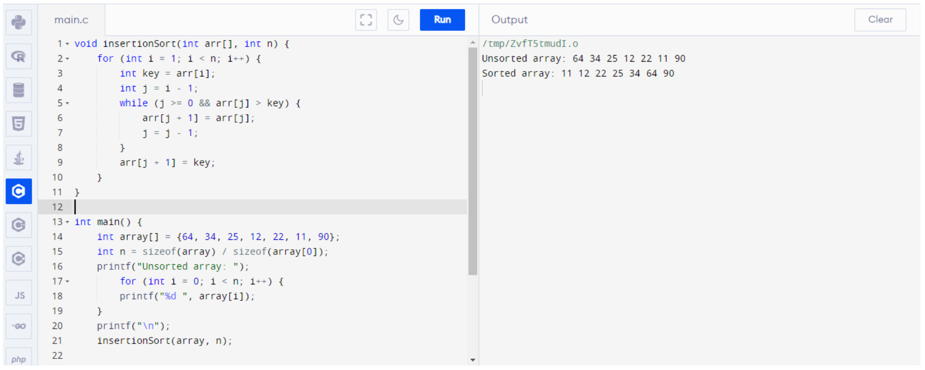


Figure 3: Flowchart of Insertion Sort

3.7 Output (C Language)



```c
void insertionSort(int arr[], int n) {
    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;
    }
}

int main() {
    int array[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(array) / sizeof(array[0]);
    printf("Unsorted array: ");
        for (int i = 0; i < n; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");
    insertionSort(array, n);

```

```
/tmp/ZvfT5tmudI.o
Unsorted array: 64 34 25 12 22 11 90
Sorted array: 11 12 22 25 34 64 90
```

Figure 6: Insertion Sort Output

3.8 Conclusion

Insertion Sort is a simple yet effective sorting algorithm, particularly useful for small datasets or when elements are nearly sorted. Its ease of implementation and in-place sorting nature make it a valuable tool for certain scenarios. However, its time complexity renders it inefficient for large datasets, and it may not be suitable for random or reverse-ordered data.

4 Selection Sort

4.1 Theory of Selection Sort

Selection Sort is a simple sorting algorithm that sorts an array by repeatedly selecting the minimum (or maximum) element from the unsorted part and placing it at the beginning (or end) of the sorted part. The algorithm divides the array into a sorted and an unsorted region.

4.2 Complexity Analysis

Time Complexity:

- Worst-case: $O(n^2)$ comparisons and $O(n)$ swaps.
- Best-case: $O(n^2)$ comparisons and $O(n)$ swaps.

Space Complexity: $O(1)$ as it requires only a constant amount of extra space for temporary variables.

4.3 Advantages and Disadvantages

Advantages:

- Simple to understand and implement.
- In-place sorting algorithm.
- Reduces the number of swaps compared to Bubble Sort.

Disadvantages:

- Inefficient for large datasets due to its quadratic time complexity.
- Not suitable for datasets with a random or reverse order.

4.4 Selection Sort Algorithm

The algorithm involves dividing the array into a sorted and an unsorted region. It repeatedly selects the minimum element from the unsorted region and swaps it with the first element of the unsorted region.

Selection Sort [1] SelectionSortarr, n $i \leftarrow 0$ to $n-1$ $minIndex \leftarrow i$ $j \leftarrow i+1$ to n $arr[j] < arr[minIndex]$ $minIndex \leftarrow j$ Swap $arr[i]$ and $arr[minIndex]$

4.5 Basic flowchart

1. Start the algorithm.
2. Initialize the array.
3. Loop through the array indices using i.
4. Set minIndex to the current index i.
5. Loop through the remaining array elements using j.
6. Compare the element at arr[j] with the element at arr[minIndex].
7. If arr[j] is smaller, update minIndex to j.
8. Swap the element at arr[i] with the element at arr[minIndex].
9. Complete the inner loop.
10. Complete the outer loop, moving to the next index.
11. End the algorithm.

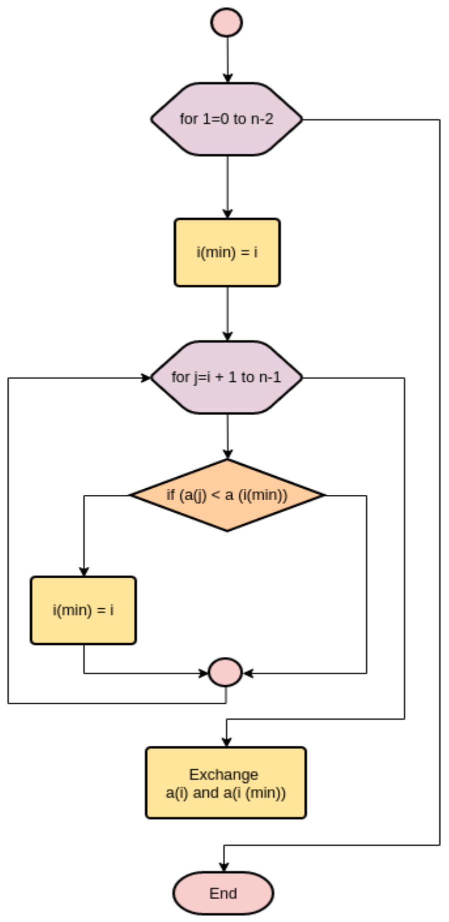
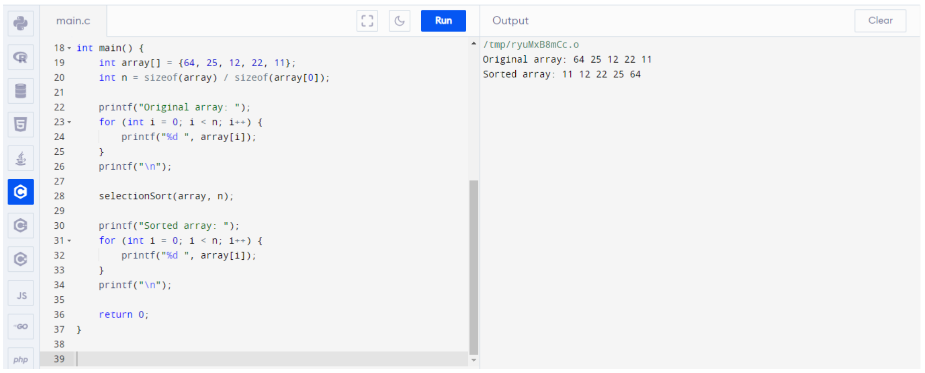


Figure 7: FlowChart of Selection Sort

4.6 Sample Code(C language)

```
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;
        }
    }
    // Swap arr[i] and arr[minIndex]
    int temp = arr[i];
    arr[i] = arr[minIndex];
    arr[minIndex] = temp;
}
```

4.7 Output (C Language)



```c
int main() {
    int array[] = {64, 25, 12, 22, 11};
    int n = sizeof(array) / sizeof(array[0]);

    printf("Original array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");

    selectionSort(array, n);

    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");

    return 0;
}
```

```
/tmp/ryuMxB8mCc.o
Original array: 64 25 12 22 11
Sorted array: 11 12 22 25 64
```

Figure 8: Selection Sort Output

4.8 Conclusion

Selection Sort is a straightforward sorting algorithm that efficiently places the smallest (or largest) element in its correct position with each pass. Its simplicity makes it easy to understand and implement, making it suitable for educational purposes and small datasets. However, due to its quadratic time complexity, Selection Sort becomes inefficient for larger datasets. While it's an improvement over algorithms like Bubble Sort, it still has limitations in terms of scalability and adaptability to varying input conditions. For scenarios where performance is critical, other sorting algorithms with better time complexities might be more appropriate choices.

5 Merge Sort

5.1 Theory of Merge Sort

Merge Sort is a divide-and-conquer algorithm that divides an array into two halves, sorts them separately, and then merges the sorted halves to produce the final sorted array. The algorithm uses a recursive approach to achieve its sorting.

5.2 Complexity Analysis

Time Complexity:

- Worst-case: $O(n \log n)$ comparisons and swaps.
- Best-case: $O(n \log n)$ comparisons and swaps.

Space Complexity: $O(n)$ for the additional space required for merging.

5.3 Advantages and Disadvantages

Advantages:

- Efficient for large datasets due to its better time complexity.
- Stable sorting algorithm.
- Well-suited for external sorting.

Disadvantages:

- Requires additional space for merging.
- More complex to implement compared to Bubble or Selection Sort.
- May have slightly slower performance for small datasets due to overhead.

5.4 Merge Sort Flowchart

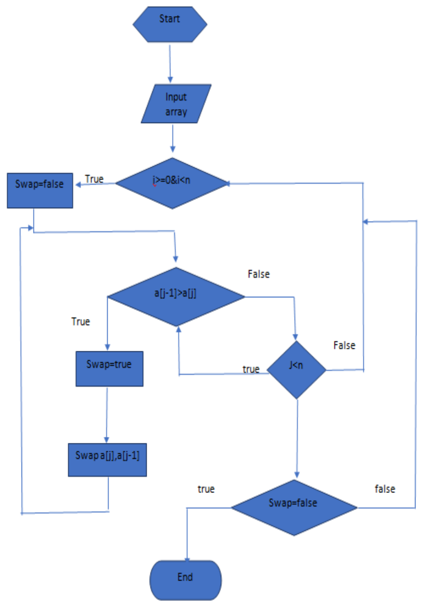


Figure 9: FlowChart of Merge Sort

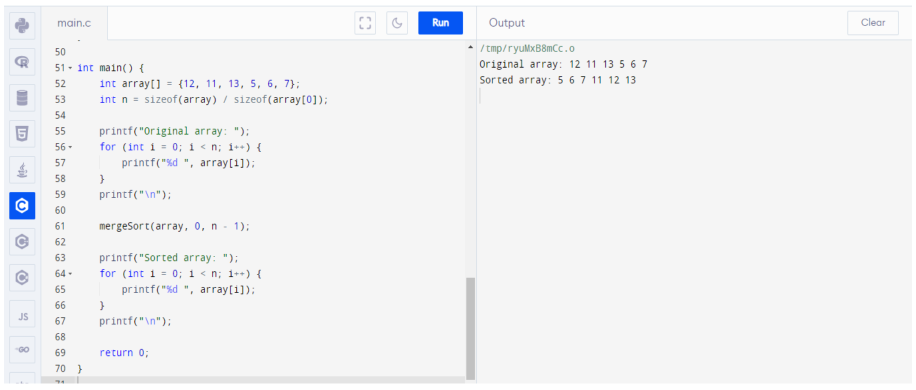
Merge Sort Algorithm: Step-by-Step Explanation Start: Begin the algorithm. Divide: Divide the array into two halves: a left half and a right half. Sort left half (Recursively): Apply the Merge Sort algorithm to the left half of the array. This involves dividing the left half further into sub-halves and sorting them using recursion. Sort right half (Recursively): Similarly, apply the Merge Sort algorithm to the right half of the array.

Merge sorted left and right halves: Once both the left and right halves are sorted, merge them to create a single sorted array. This is the core operation of the Merge Sort algorithm. Compare the first element of the left half with the first element of the right half. Place the smaller element in the merged array. Move to the next element in the sub-array from which the element was taken. Repeat the comparison and placement process until both sub-arrays are exhausted. If one sub-array is exhausted before the other, simply place the remaining elements from the other sub-array into the merged array. End: Finish the algorithm, with the entire array now sorted in ascending order.

5.5 Sample Code(C language)

```c
void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;

    int leftArr[n1], rightArr[n2];

    for (int i = 0; i < n1; i++) {
        leftArr[i] = arr[left + i];
    }
    for (int j = 0; j < n2; j++) {
        rightArr[j] = arr[mid + 1 + j];
    }

    int i = 0, j = 0, k = left;

    while (i < n1 && j < n2) {
        if (leftArr[i] <= rightArr[j]) {
            arr[k] = leftArr[i];
            i++;
        } else {
            arr[k] = rightArr[j];
            j++;
        }
        k++;
    }

    while (i < n1) {
        arr[k] = leftArr[i];
        i++;
        k++;
    }

    while (j < n2) {
            arr[k] = rightArr[j];
            j++;
            k++;
        }
}

void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
```

5.6 Output (C Language)



```
int main() {
    int array[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(array) / sizeof(array[0]);

    printf("Original array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");

    mergeSort(array, 0, n - 1);

    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");

    return 0;
}
```

```
/tmp/ryuMxB8mCc.o
Original array: 12 11 13 5 6 7
Sorted array: 5 6 7 11 12 13
```

Figure 10: Merge Sort Output

5.7 Conclusion

Merge Sort is a powerful sorting algorithm that leverages the divide-and-conquer strategy to achieve efficient and stable sorting. Its time complexity makes it suitable for sorting large datasets. While it requires additional space for merging and can be more complex to implement, its benefits in terms of performance and stability can outweigh these drawbacks. Merge Sort's suitability for external sorting also makes it a valuable choice in scenarios where sorting larger-than-memory datasets is required.

6 QuickSort

6.1 Introduction

QuickSort is a widely used sorting algorithm known for its efficiency in sorting large datasets. It employs a divide-and-conquer strategy and is particularly effective for average-case scenarios.

6.2 Complexities

Time Complexity:

- Average-case: $O(n \log n)$ comparisons and swaps.
- Worst-case (rare): $O(n^2)$ comparisons and swaps.

Space Complexity: $O(\log n)$ due to the recursion stack.

6.3 Advantages and Disadvantages

Advantages:

- Efficient for large datasets.
- In-place sorting algorithm.
- Better average-case performance compared to many other sorting algorithms.

Disadvantages:

- Worst-case time complexity can be poor.
- Not a stable sorting algorithm.

6.4 QuickSort Algorithm

The QuickSort algorithm involves the following steps:

[H] QuickSort [1] QuickSort$arr, low, high$ $low < high$ $pivotIndex \leftarrow$ Partition$arr, low, high$ QuickSort$arr, low, pivotIndex - 1$ QuickSort$arr, pivotIndex + 1, high$

6.5 Flowchart

Start: Begin the QuickSort algorithm.

Partitioning: - Choose a pivot element (typically the last element in the array). - Divide the array into two sub-arrays: elements less than the pivot and elements greater than the pivot.

Recursively Sort Sub-Arrays: - Apply QuickSort to the sub-array with elements less than the pivot. - Apply QuickSort to the sub-array with elements greater than the pivot.

Combining Sorted Sub-Arrays: - As the sub-arrays are sorted, no further action is needed to combine them.

End: Finish the QuickSort algorithm.

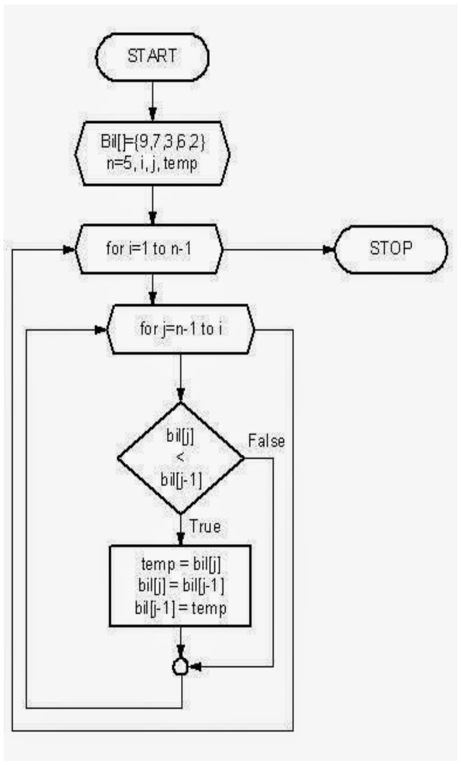
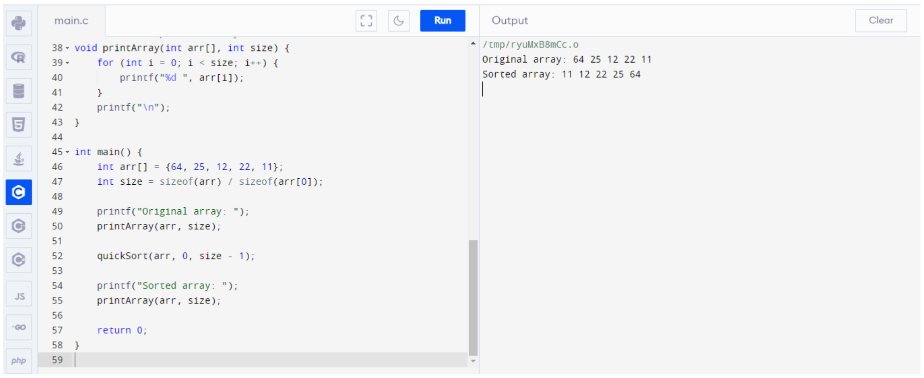


Figure 11: FlowChart of QuickSort

6.6 Sample Code (C language)

```
void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

// Function to partition the array and return the pivot index
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Choose the last element as the pivot
    int i = (low - 1); // Index of smaller element

    for (int j = low; j <= high - 1; j++) {
        // If the current element is smaller than or equal to the pivot
        if (arr[j] <= pivot) {
            i++; // Increment the index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

// Function to implement QuickSort algorithm
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // Find pivot such that element at pivot is in correct position
        int pivotIndex = partition(arr, low, high);

        // Recursive call on the left and right sub-arrays
        quickSort(arr, low, pivotIndex - 1);
        quickSort(arr, pivotIndex + 1, high);
    }
}
```

6.7 Output (C Language)



```
void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int size = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, size);

    quickSort(arr, 0, size - 1);

    printf("Sorted array: ");
    printArray(arr, size);

    return 0;
}
```

```
/tmp/ryuMxB8mCc.o
Original array: 64 25 12 22 11
Sorted array: 11 12 22 25 64
```

Figure 12: QuickSort Output

6.8 Conclusion

QuickSort is a powerful sorting algorithm that offers efficient performance for large datasets. Its divide-and-conquer approach and clever pivot selection make it a popular choice in practice. However, its worst-case time complexity in rare scenarios and lack of stability should be considered when choosing the algorithm for specific use cases.

7 HeapSort

7.1 Introduction

HeapSort is a comparison-based sorting algorithm that uses a binary heap data structure to build a partially ordered tree. It achieves sorting by repeatedly extracting the maximum element from the heap and placing it in the sorted portion of the array.

7.2 Complexities

Time Complexity:

- Average and worst-case: $O(n \log n)$ comparisons and swaps.
- Best-case: $O(n \log n)$ comparisons and swaps.

Space Complexity: $O(1)$ as it requires only a constant amount of extra space for temporary variables.

7.3 Advantages and Disadvantages

Advantages:

- Guaranteed $O(n \log n)$ performance.
- In-place sorting algorithm.
- Suitable for large datasets.

Disadvantages:

- Not a stable sorting algorithm.
- Requires additional memory for the heap data structure.

7.4 HeapSort Algorithm

The HeapSort algorithm involves the following steps:

[H] HeapSort [1] HeapSortarr, n $i \leftarrow n/2 - 1$ down to 0 Heapifyarr, n, i $i \leftarrow n - 1$ down to 1 Swap $arr[0]$ and $arr[i]$ Heapify$arr, i, 0$

7.5 Flowchart

Start: Begin the HeapSort algorithm.

Build Max Heap: - Convert the given array into a max heap, using the heapify procedure. - Starting from the last non-leaf node and moving upwards, ensure that each subtree satisfies the max-heap property.

Heapify Procedure: - Given an array and an index i: - Find the left and right child indices (2 * i + 1 and 2 * i + 2). - Identify the largest element among the current node, left child, and right child. - If the largest is not the current node, swap the largest element with the current node and call heapify on the affected subtree.

Sort the Heap: - Repeatedly extract the maximum element from the heap (root) and place it at the end of the sorted array. - After each extraction, adjust the heap structure using the heapify procedure to maintain the max heap property.

End: Finish the HeapSort algorithm.

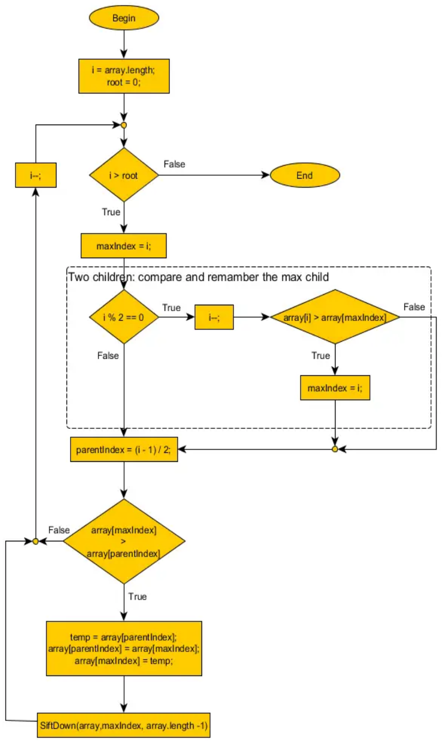
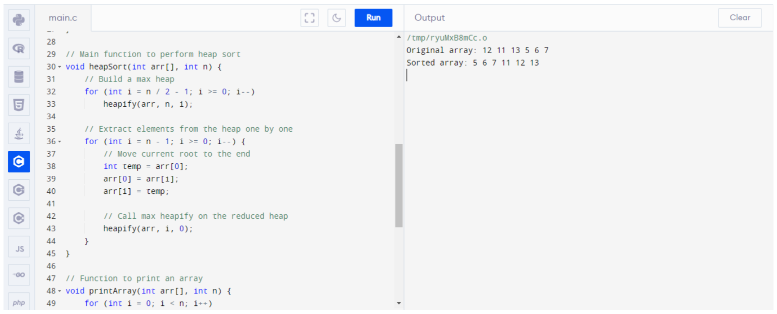

Begin
i = array.length;
root = 0;
i > root
False
End
True
i--;
maxIndex = i;
Two children: compare and remamber the max child
i % 2 == 0
True
i--;
array[i] > array[maxIndex]
False
True
maxIndex = i;
False
parentIndex = (i - 1) / 2;
array[maxIndex]
>
array[parentIndex]
False
True
temp = array[parentIndex];
array[parentIndex] = array[maxIndex];
array[maxIndex] = temp;
SiftDown(array,maxIndex, array.length -1)

Figure 13: FlowChart of HeapSort

7.6 Sample Code (C language)

```c
void heapify(int arr[], int n, int i) {
    int largest = i; // Initialize largest as root
    int left = 2 * i + 1; // Left child index
    int right = 2 * i + 2; // Right child index

    // If left child is larger than root
    if (left < n && arr[left] > arr[largest])
        largest = left;

    // If right child is larger than largest so far
    if (right < n && arr[right] > arr[largest])
        largest = right;

    // If largest is not root
    if (largest != i) {
        // Swap arr[i] and arr[largest]
        int temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}

// Main function to perform heap sort
void heapSort(int arr[], int n) {
    // Build a max heap
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // Extract elements from the heap one by one
    for (int i = n - 1; i >= 0; i--) {
        // Move current root to the end
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

        // Call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}
```

7.7 Output (C Language)



```c
// Main function to perform heap sort
void heapSort(int arr[], int n) {
    // Build a max heap
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // Extract elements from the heap one by one
    for (int i = n - 1; i >= 0; i--) {
        // Move current root to the end
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

        // Call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}

// Function to print an array
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++)
```

```
/tmp/ryuMxB8mCc.o
Original array: 12 11 13 5 6 7
Sorted array: 5 6 7 11 12 13
```

Figure 14: Heap Sort Output

7.8 Conclusion

HeapSort is an efficient sorting algorithm that achieves guaranteed $O(n \log n)$ performance. Its ability to handle large datasets and in-place sorting nature make it a valuable tool. However, its lack of stability and additional memory requirements for the heap data structure should be considered when choosing the algorithm for specific scenarios.