Java Sorting Algorithms

Title: Java Sorting Algorithms

Name : Dinidu Sachintha

Institution: IJSE GDSE- 71 BATCH

Date: 2024 - 05 - 15

Abstract

This document provides a detailed study of key sorting algorithms in Java, covering Selection Sort, Bubble Sort, Insertion Sort, Merge Sort, Heap Sort, Tree Sort, and Counting Sort. It includes comprehensive algorithm descriptions, pseudocode, Java implementations, and analyses of their advantages and disadvantages. The comparison of these algorithms is based on their performance metrics and use cases, offering insights into both comparison-based and non-comparison-based sorting techniques.

Table of Contents

1. Introduction

1.1 Background: Importance of Sorting in Computer Science

Sorting is a fundamental operation in computer science, serving as a critical process in various applications such as searching, data analysis, and optimization. Efficient sorting algorithms improve the performance of other algorithms that require sorted data, such as binary search algorithms and merge operations. Sorting is not only essential for simplifying data processing but also for enhancing data presentation and storage. Given its pivotal role, understanding and implementing sorting algorithms is a core component of computer science education and practice.

1.2 Objectives: Purpose of the Document

This document aims to provide a comprehensive study of several key sorting algorithms implemented in Java. By examining both comparison-based and non-comparison-based sorting algorithms, this document seeks to:

- **Offer detailed explanations of each algorithm, including their theoretical foundations and practical implementations in Java.**
- **Analyze the time and space complexities of these algorithms to understand their efficiency.**
- **Discuss the advantages and disadvantages of each algorithm to highlight their appropriate use cases.**

2. Sorting Algorithms Overview

Sorting algorithms are a fundamental concept in computer science that arrange elements of a list or an array in a specific order (ascending or descending). Sorting is a crucial operation in many applications, such as database management, data processing, and scientific computing. It is often a prerequisite for other algorithms, such as searching and merging, which require data to be in a sorted order for efficient operation.

- **Data management: Sorting data facilitates efficient searching, retrieval, and analysis.**
- **Database optimization: Sorted data in databases enhances query performance.**
- **Machine learning: Sorting is often a preprocessing step for training machine learning models.**
- **Data visualization: Sorting data can reveal patterns, trends, and outliers in datasets.**

2.1 Classification

Sorting algorithms can be broadly classified into two categories: **comparison-based sorting algorithms** and **non-comparison-based sorting algorithms.**

2.2 Comparison-Based Sorting Algorithms

Comparison-based sorting algorithms work by comparing elements to determine their relative order. These algorithms have a lower bound time complexity of O(n log n) in the average case, where n is the number of elements to be sorted. Some examples of comparison-based sorting algorithms include:

- **Selection Sort**: This algorithm divides the input list into two parts: a sorted sublist and an unsorted sublist. It repeatedly finds the minimum element from the unsorted sublist and swaps it with the leftmost unsorted element.

- **Bubble Sort**: This algorithm repeatedly swaps adjacent elements if they are in the wrong order, effectively "bubbling" the largest elements to the end of the list.

- **Insertion Sort**: This algorithm builds the final sorted array one item at a time by taking an element from the input and inserting it into its correct position in the already sorted sublist.

- **Merge Sort**: This is a divide-and-conquer algorithm that recursively divides the input array into two halves, sorts them, and then merges the sorted halves.

- **Heap Sort**: This algorithm builds a binary heap data structure from the input array and then repeatedly extracts the maximum element from the heap to construct the sorted array.

- **Quick Sort**: This is another divide-and-conquer algorithm that selects a "pivot" element from the array and partitions the other elements into two sub-arrays, according to whether they are less than or greater than the pivot.

2.3 Non-Comparison-Based Sorting Algorithms

Non-comparison-based sorting algorithms do not rely on comparisons between elements. Instead,use the internal character of the values to be sorted. It can only be applied to some particular cases, and requires particular values. And the best complexity is probably better depending on cases, such as `O(n)`. Examples of non-comparison-based sorting algorithms include:

- **Counting Sort**: This algorithm counts the number of occurrences of each distinct element and uses this information to construct the sorted array.

- **Radix Sort**: This algorithm sorts elements by processing individual digits or groups of digits, starting from the least significant digit or group.

3. Comparison-Based Sorting Algorithms

3.1 Bubble Sort

Bubble Sort is a simple sorting algorithm that repeatedly swaps adjacent elements if they are in the wrong order. It is an in-place algorithm with a time complexity of O(n^2), making it inefficient for large datasets.

3.1.1 How does Bubble Sort Work without programming?

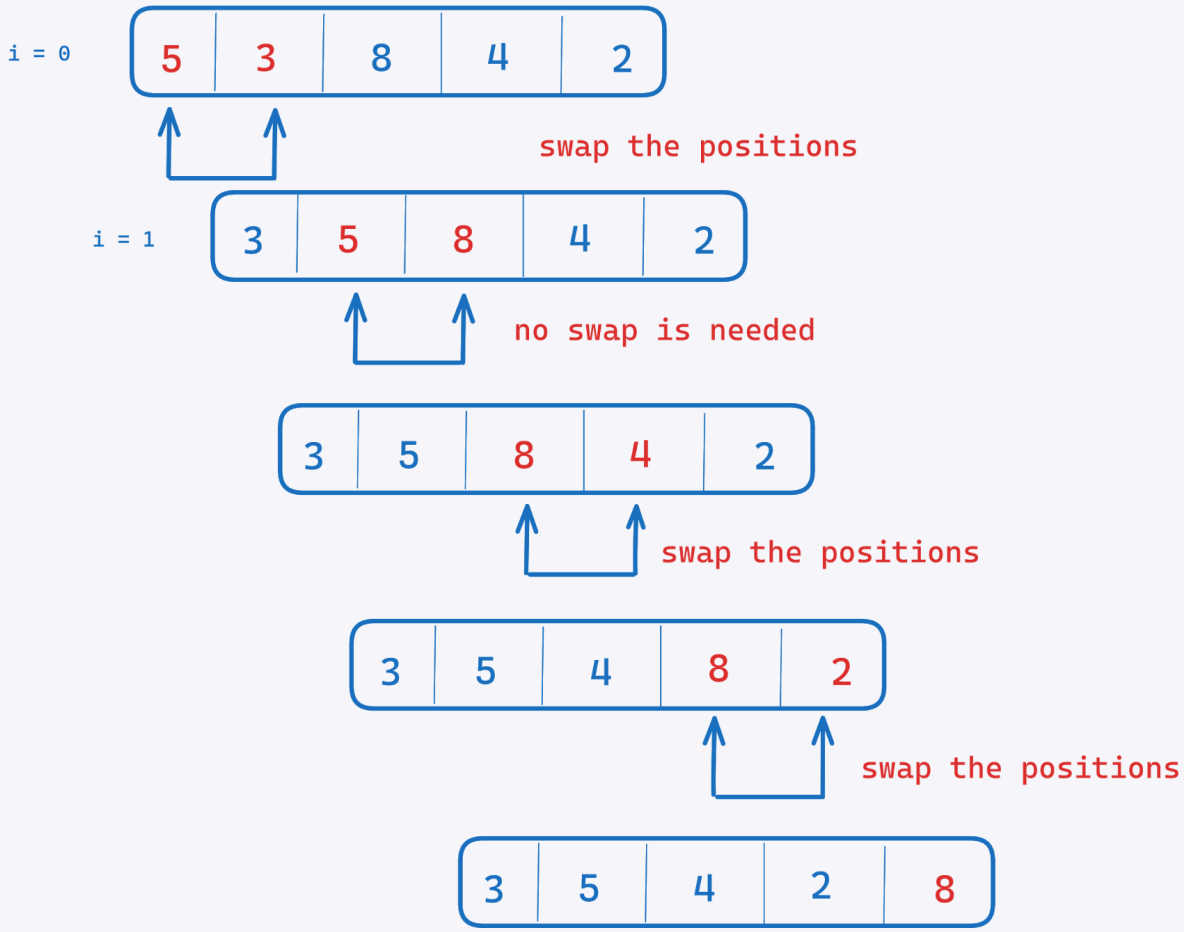
Let us understand the working of bubble sort with the help of the following illustration .

Imagine you have a list of numbers that you want to sort, for example: [5, 3, 8, 4, 2]

Bubble sort.png

Created by :
Dinidu Sachintha

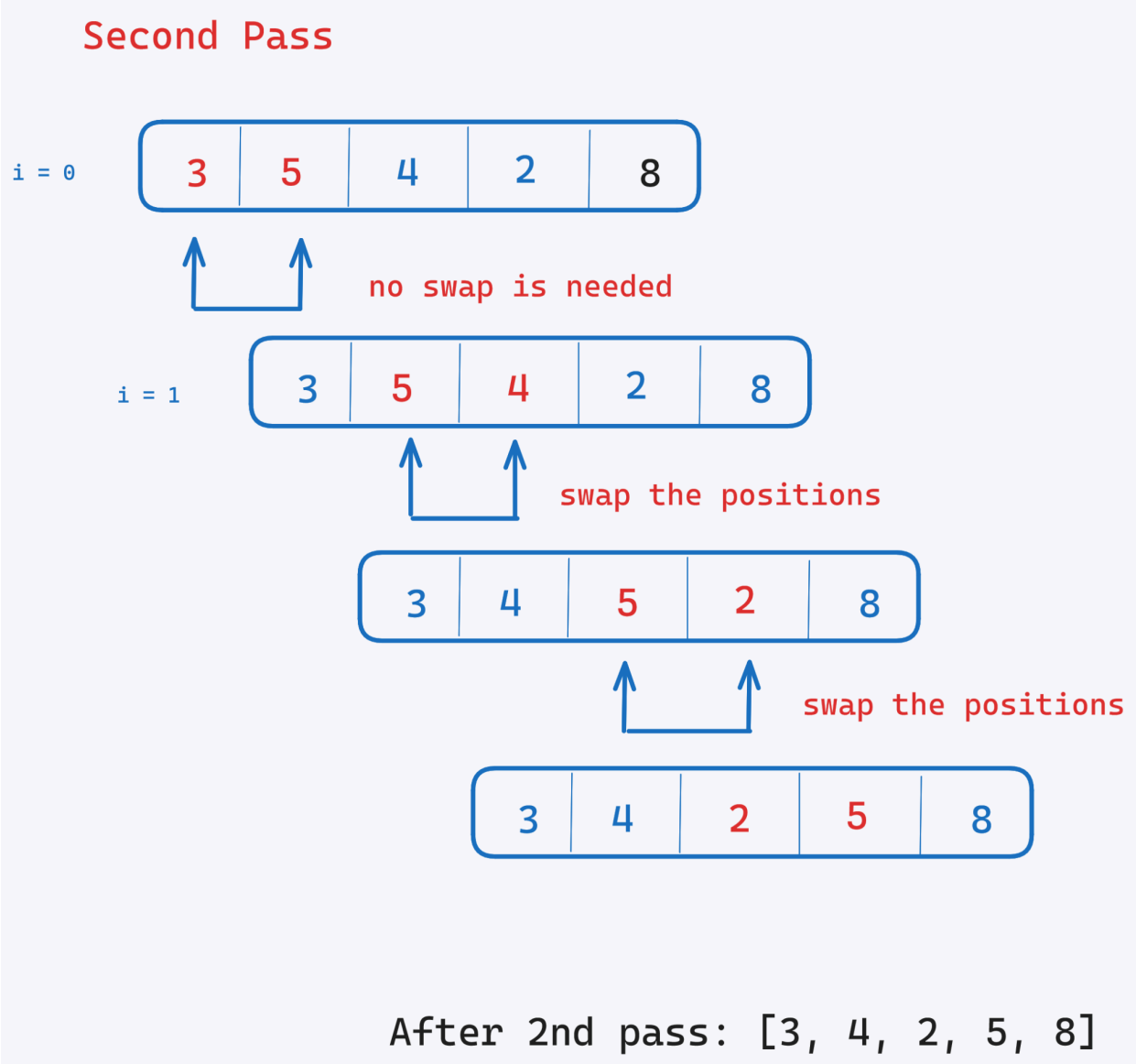
First Pass



After 1st pass: [3, 5, 4, 2, 8]

First Pass:

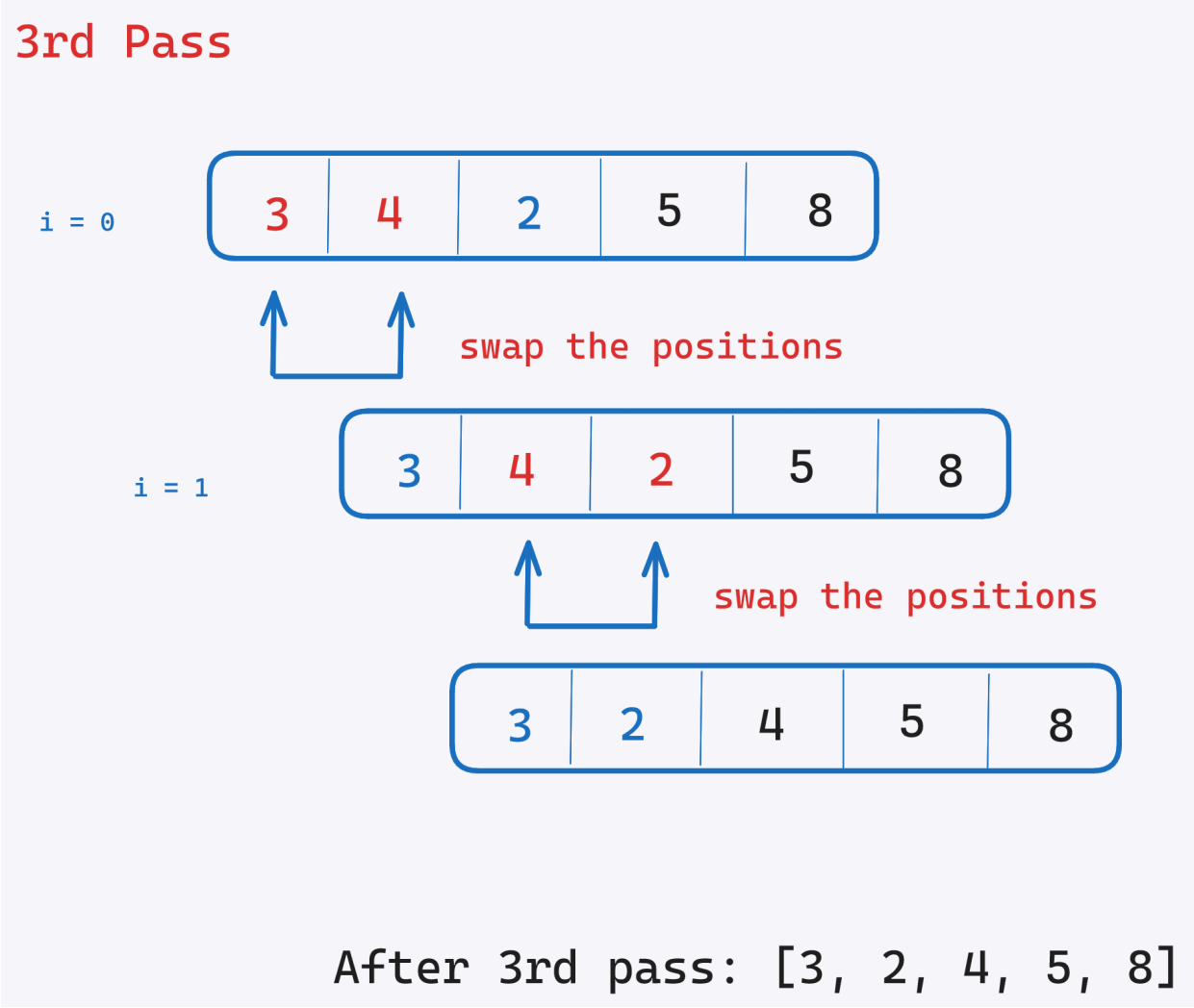
- Compare the first two numbers (5 and 3). Since 5 is greater than 3, swap them. The list now looks like: [3, 5, 8, 4, 2].
- Move to the next pair (5 and 8). Since 5 is less than 8, no swap is needed.
- Move to the next pair (8 and 4). Since 8 is greater than 4, swap them. The list now looks like: [3, 5, 4, 8, 2]
- Move to the next pair (8 and 2). Since 8 is greater than 2, swap them. The list now looks like: [3, 5, 4, 2, 8].
- At the end of the first pass, the largest number (8) has "bubbled up" to its correct position at the end of the list.



Second Pass:

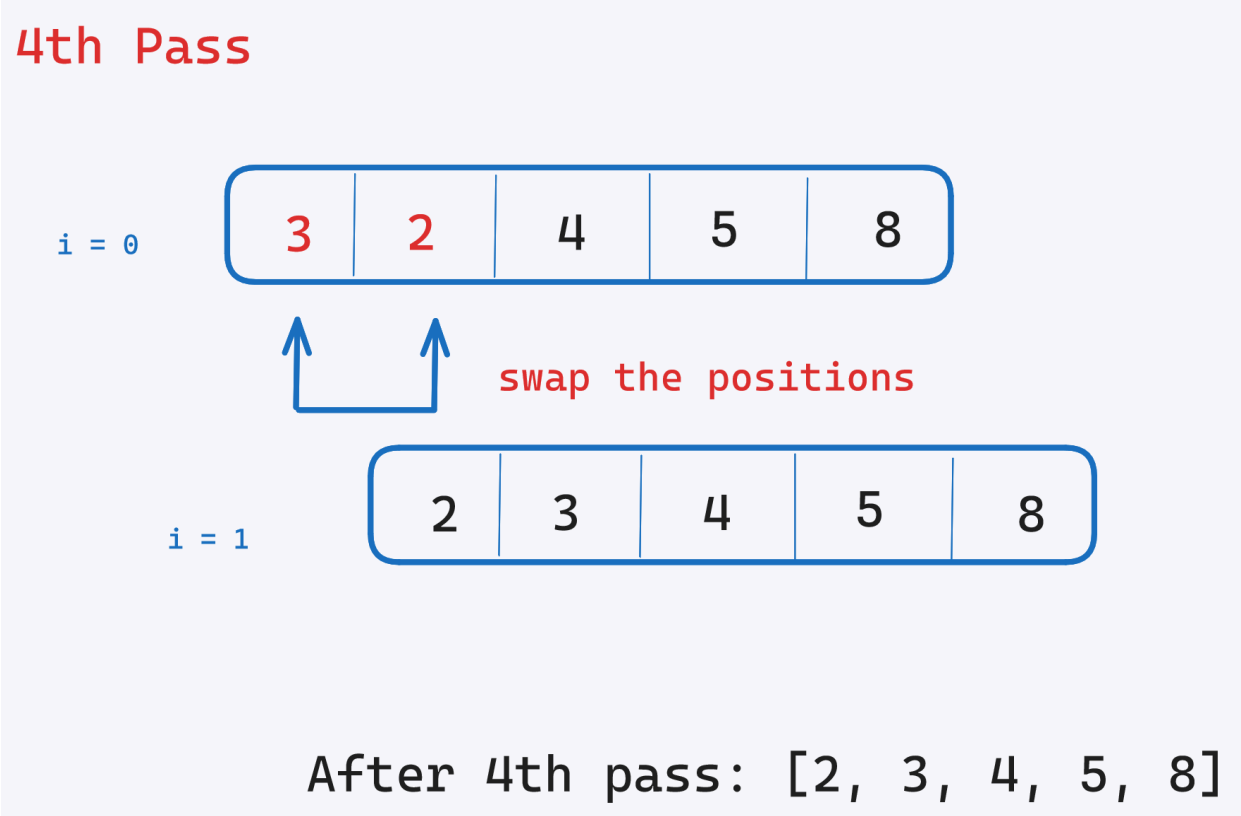
- Start again at the beginning of the list.
- Compare the first two numbers (3 and 5). Since 3 is less than 5, no swap is needed.
- Move to the next pair (5 and 4). Since 5 is greater than 4, swap them. The list now looks like: [3, 4, 5, 2, 8].
- Move to the next pair (5 and 2). Since 5 is greater than 2, swap them. The list now looks like: [3, 4, 2, 5, 8].

- The second largest number (5) is now in its correct position, and the list is partially sorted.



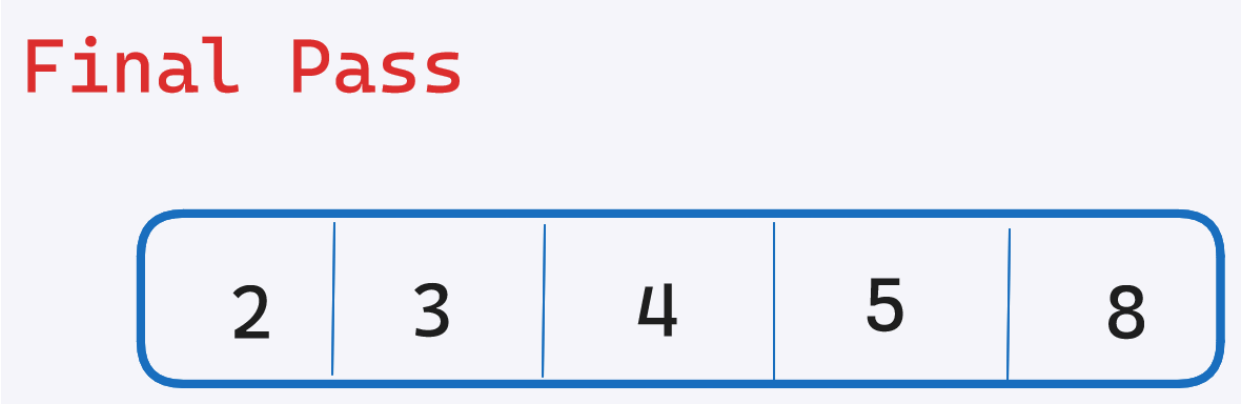
Third Pass:

- Start again at the beginning.
- Compare the first two numbers (3 and 4). Since 3 is less than 4, no swap is needed.
- Move to the next pair (4 and 2). Since 4 is greater than 2, swap them. The list now looks like: [3, 2, 4, 5, 8].
- The third largest number (4) is now in its correct position.



Fourth Pass:

- Start again at the beginning.
- Compare the first two numbers (3 and 2). Since 3 is greater than 2, swap them. The list now looks like: [2, 3, 4, 5, 8].
- Now, all elements are in their correct positions.



Final Check:

- The algorithm performs one more pass to check that no swaps are needed, confirming that the list is sorted.

Visual Example:
Initial list: [5, 3, 8, 4, 2]
After 1st pass: [3, 5, 4, 2, 8]
After 2nd pass: [3, 4, 2, 5, 8]
After 3rd pass: [3, 2, 4, 5, 8]
After 4th pass: [2, 3, 4, 5, 8]
Final list (after check pass): [2, 3, 4, 5, 8]

YOU CAN CAN SEE MY WHOLE DIAGRAM CLICKING THIS Bubble sort.png

3.1.2 Practical Scenario :

Imagine you have a queue of people waiting for a movie and want to sort them alphabetically by their last name. You can use Bubble Sort like this:

1. **Compare the last names of the first two people in line. If the second person's name comes alphabetically after the first, swap their places in line.**

2. **Repeat step 1, moving down the line one position at a time, comparing and swapping as needed**

3. **Once you reach the end of the line (one complete pass), the person with the "latest" last name alphabetically will be at the very back.**

4. **Repeat steps 1-3, starting from the beginning again. With each pass, you'll need to compare and swap fewer people as those with "later" last names move towards the back.**

5. **Continue making passes until you reach a point where no one needs to swap places during a complete pass through the entire line. This means the line is now sorted alphabetically by last name.**

3.1.3 Implementation of Bubble Sort :

Main.java

```java
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]) {
                // Swap arr[j] and arr[j+1]
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

    public static void main(String args[]) {
        int[] arr = {45, 50, 98, 41};
        System.out.println(Arrays.toString(arr));    //[45, 50, 98, 41]

        bubbleSort(arr);
        System.out.println(Arrays.toString(arr));    //[41, 45, 50, 98]
    }
}
```

3.1.4 Advantages of Bubble Sort:

- Bubble sort is easy to understand and implement.
- It does not require any additional memory space.
- It is a stable sorting algorithm, meaning that elements with the same key value maintain their relative order in the sorted output.

3.1.5 Disadvantages of Bubble Sort:

- Bubble sort has a time complexity of $O(N^2)$ which makes it very slow for large data sets.
- Bubble sort is a comparison-based sorting algorithm, which means that it requires a comparison operator to determine the relative order of elements in the input data set. It can limit the efficiency of the algorithm in certain cases.

3.1.6 Where is the Bubble sort algorithm used?

Due to its simplicity, bubble sort is often used to introduce the concept of a sorting algorithm. In computer graphics, it is popular for its capability to detect a tiny error (like a swap of just two elements) in almost-sorted arrays and fix it with just linear complexity (2n).

Example: It is used in a polygon filling algorithm, where bounding lines are sorted by their x coordinate at a specific scan line (a line parallel to the x-axis), and with incrementing by their order changes (two elements are swapped) only at intersections of two lines.

3.2 Selection Sort

Selection sort is a simple and efficient sorting algorithm that works by repeatedly selecting the smallest (or largest) element from the unsorted portion of the list and moving it to the sorted portion of the list.

- **The algorithm repeatedly selects the smallest (or largest) element from the unsorted portion of the list and swaps it with the first element of the unsorted part. This process is repeated for the remaining unsorted portion until the entire list is sorted.**

3.2.1 How does Selection Sort Work without programming?

Let us understand the working of Selection sort with the help of the following illustration.

Imagine you have a list of numbers that you want to sort in ascending order. For example,

let's use the list: [29, 10, 14, 37, 13]

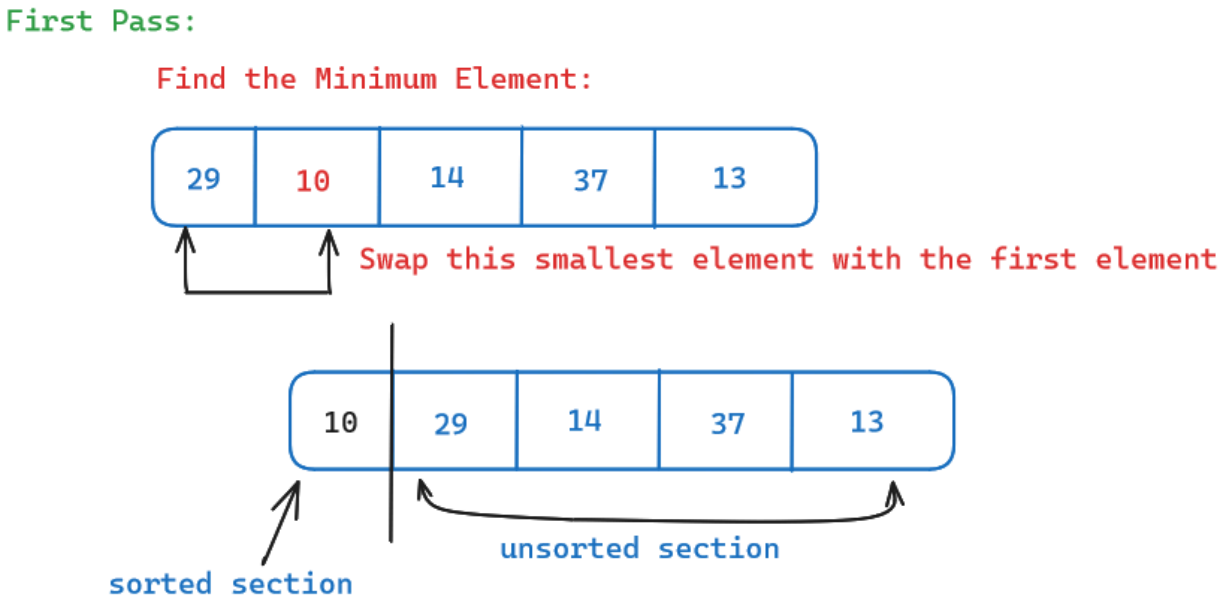
Selection Sort.png

1. Start with an Unsorted List:

Imagine you have a list of numbers that you want to sort in ascending order. For example, let's use the list: [29, 10, 14, 37, 13].

2. Divide the List into Two Parts:

The list is conceptually divided into two parts: a sorted part (initially empty) and an unsorted part (initially containing all the elements).




```
- Find minimum in [29, 10, 14, 37, 13]: 10
- Swap 10 with 29
- List: [10, 29, 14, 37, 13]
```

```
CREATED BY : DINIDU SACHINTHA
```

3. Find the Minimum Element:

In each iteration, find the smallest (minimum) element from the unsorted part of the list.

- In the first pass, look at the entire list: [29, 10, 14, 37, 13].

- The smallest element is 10.

4. Swap the Minimum Element:

Swap this smallest element with the first element of the unsorted part of the list.

- Swap 10 with 29.

- The list now looks like this: [10, 29, 14, 37, 13].

- Now, 10 is part of the sorted section, and the rest is unsorted.

Second Pass:

Repeat the Process: (Find the Minimum Element:)

Swap this smallest element with the second element

- Find minimum in [29, 14, 37, 13]: 13
- Swap 13 with 29
- List: [10, 13, 14, 37, 29]

CREATED BY : DINIDU SACHINTHA

5. Repeat the Process:

Repeat the process for the remaining unsorted part of the list.

- Look at the unsorted part: [29, 14, 37, 13].

- The smallest element is 13.

- Swap 13 with 29.

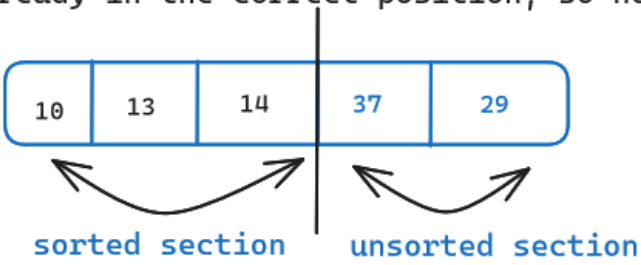
- The list now looks like this: [10, 13, 14, 37, 29].

- Now, 10 and 13 are sorted, and the rest is unsorted.

Third Pass:

Repeat the Process: (Find the Minimum Element:)

14 is already in the correct position, so no swap is needed.



- Find minimum in [14, 37, 29]: 14
- 14 is already in place, no swap
- List: [10, 13, 14, 37, 29]

CREATED BY : DINIDU SACHINTHA

6. Continue Until the List is Sorted:

Continue the process until the entire list is sorted.

- Next, look at [14, 37, 29].

- The smallest element is 14.

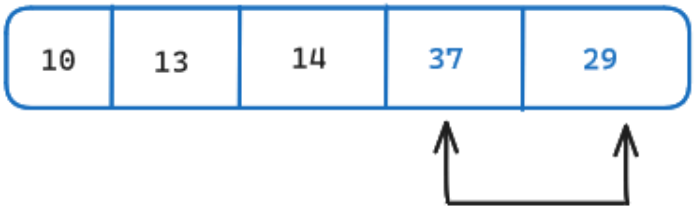
- 14 is already in the correct position, so no swap is needed.

- The list remains: [10, 13, 14, 37, 29].

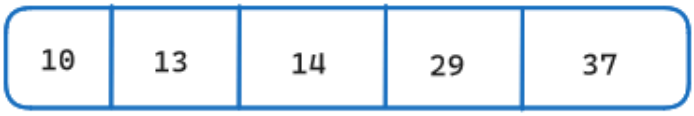
- Now, 10, 13, and 14 are sorted.

Fourth Pass:

Repeat the Process:(Find the Minimum Element:)



Swap this smallest element with the fourth element



- Find minimum in [37, 29]: 29
- Swap 29 with 37
- List: [10, 13, 14, 29, 37]

CREATED BY : DINIDU SACHINTHA

- Next, look at [37, 29].

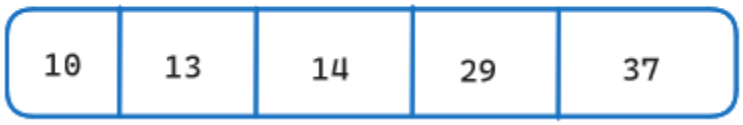
- The smallest element is 29.

- Swap 29 with 37.

- The list now looks like this: [10, 13, 14, 29, 37].

- Now, 10, 13, 14, and 29 are sorted, leaving only one element.

Final List:



- The final element is already in place: [10, 13, 14, 29, 37].

Visualizing Selection Sort

1. Initial List:

[29, 10, 14, 37, 13]

2. First Pass:

- Find minimum in [29, 10, 14, 37, 13]: 10

- Swap 10 with 29

- List: [10, 29, 14, 37, 13]

3. Second Pass:

- Find minimum in [29, 14, 37, 13]: 13

- Swap 13 with 29

- List: [10, 13, 14, 37, 29]

4. Third Pass:

- Find minimum in [14, 37, 29]: 14

- 14 is already in place, no swap

- List: [10, 13, 14, 37, 29]

5. Fourth Pass:

- Find minimum in [37, 29]: 29

- Swap 29 with 37

- List: [10, 13, 14, 29, 37]

6. Final List:

[10, 13, 14, 29, 37]

YOU CAN CAN SEE MY WHOLE DIAGRAM CLICKING THIS Selection Sort.png

3.2.2 Practical Scenario :

Imagine you're sorting a bookshelf alphabetically by title.

1. **Start with the first book.**
2. **Compare it with all the remaining books.**
3. **If it finds a book with a title alphabetically before the first book, that book becomes the new "assumed minimum."**
4. **Swap the positions of the first book and the actual minimum.**
5. **Repeat steps 2-4 for the second book, then the third, and so on.**

By the end, each book will be in its correct alphabetical order on the shelf.

3.2.3 Implementation of Selection Sort :

3.2.4 Advantages of Selection Sort Algorithm

- Simple and easy to understand.
- Works well with small datasets.

3.2.5 Disadvantages of the Selection Sort Algorithm

- **Selection sort has a time complexity of O(n^2) in the worst and average case.**
- **Does not work well on large datasets.**
- **Does not preserve the relative order of items with equal keys which means it is not stable.**

3.2.6 Where is the Bubble sort algorithm used?

Selection sort is a simple sorting algorithm that can be useful in certain situations, despite its relatively inefficient time complexity of O(n^2). Here are some scenarios where selection sort may be used:

- **Small datasets: When the size of the dataset is small, the quadratic time complexity of selection sort is not a major concern. It can be an appropriate choice for sorting small arrays or lists.**

- **Limited memory: Selection sort is an in-place sorting algorithm, meaning it does not require additional memory proportional to the size of the input array. This makes it suitable for situations where memory is limited or when the cost of memory allocation is high.**

- **Minimal data movement: Selection sort minimizes the number of swaps required to sort the data. It performs at most n swaps, where n is the size of the input array. This can be advantageous when the cost of swapping elements is high, such as in certain embedded systems or when dealing with large data structures.**

- **Partially sorted data: If the input data is already partially sorted, selection sort can be more efficient than other quadratic sorting algorithms, such as bubble sort or insertion sort.**

- **Sorting small subarrays: Selection sort can be used as a part of more complex sorting algorithms, such as the hybrid sorting algorithms, where it is used to sort small subarrays efficiently.**

- **Specific applications: In certain applications where the dataset is small and the performance requirements are not stringent, selection sort may be a suitable choice due to its simplicity and ease of implementation.**

3.3 Insertion Sort

Insertion Sort is an in-place comparison-based algorithm that builds the final sorted array one item at a time. It has a time complexity of O(n^2) in the average and worst cases, but it is efficient for small datasets or partially sorted lists.

Insertion sort is a simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list. It is a **stable sorting** algorithm, meaning that elements with equal values maintain their relative order in the sorted output.

Insertion sort is like sorting playing cards in your hands. You split the cards into two groups: the sorted cards and the unsorted cards. Then, you pick a card from the unsorted group and put it in the right place in the sorted group.

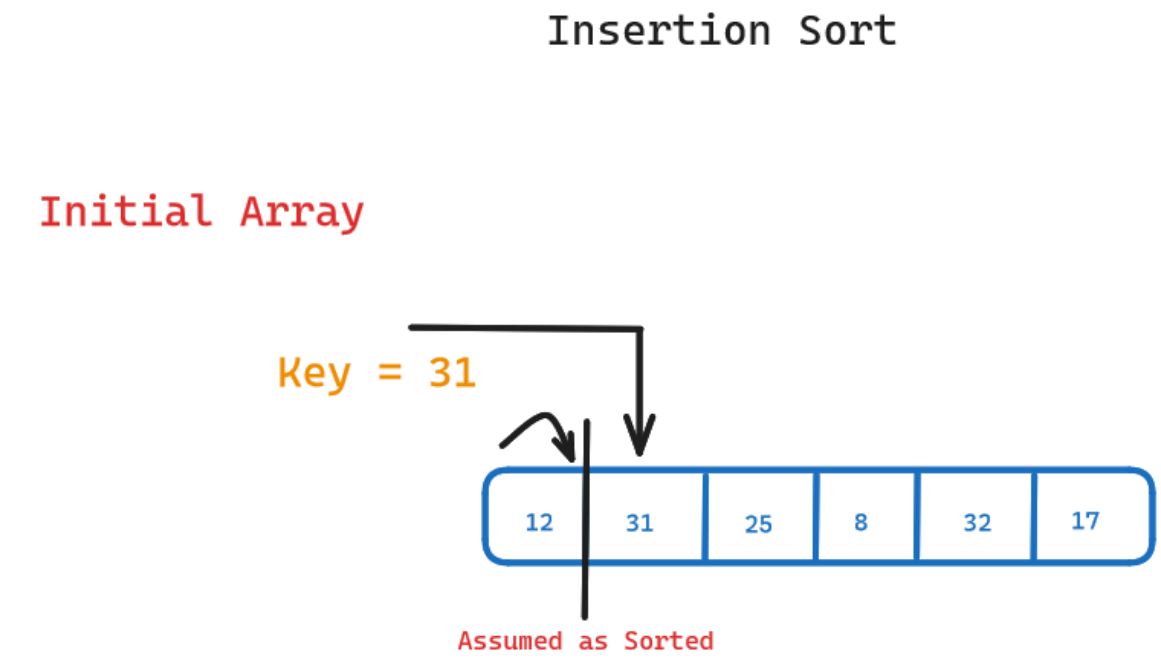
3.3.1 How does Insertion Sort Work without programming?

Let us understand the working of Insertion sort with the help of the following illustration .

Insertion Sort.png

Step-by-Step Process

1. **Initial Array:**




```
[12, 31, 25, 8, 32, 17]
```

First Iteration (index 1):

Compare 31 with 12. Since 31 is greater than 12, no shifting is needed

Created by Dinidu Sachintha

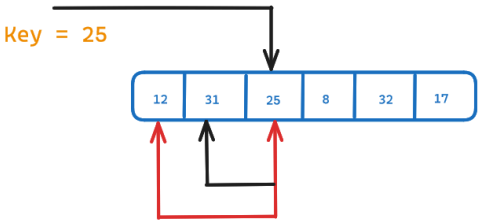
First Iteration (index 1):

- **Key = 31**
- **Compare 31 with 12. Since 31 is greater than 12, no shifting is needed.**
- **Array remains:**

[12, 31, 25, 8, 32, 17]

Second Iteration (index 2):

Created by Dinidu Sachintha



Compare 25 with 31. Since 25 is less than 31, shift 31 to the right.
Compare 25 with 12. Since 25 is greater than 12, place 25 in the position after 12.

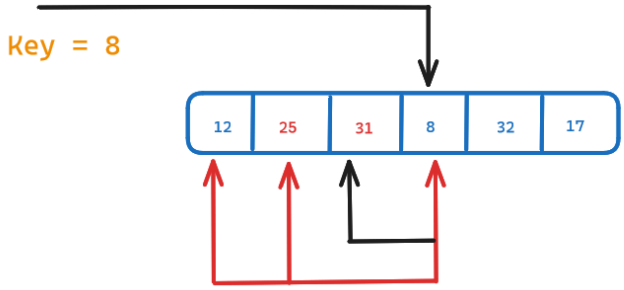
Second Iteration (index 2):

- **Key = 25**
- **Compare 25 with 31. Since 25 is less than 31, shift 31 to the right.**
- **Compare 25 with 12. Since 25 is greater than 12, place 25 in the position after 12.**

- **Array becomes:**

```
[12, 25, 31, 8, 32, 17]
```

```
Third Iteration (index 3):
```



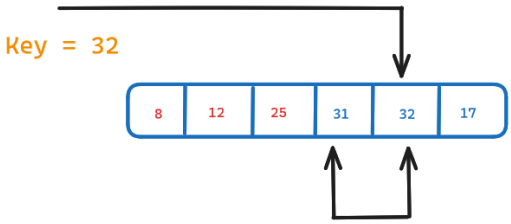

```
Compare 8 with 31. Since 8 is less than 31, shift 31 to the right.
Compare 8 with 25. Since 8 is less than 25, shift 25 to the right.
Compare 8 with 12. Since 8 is less than 12, shift 12 to the right.
Place 8 in the position of the first element.
```

Third Iteration (index 3):

- **Key = 8**
- **Compare 8 with 31. Since 8 is less than 31, shift 31 to the right.**
- **Compare 8 with 25. Since 8 is less than 25, shift 25 to the right.**
- **Compare 8 with 12. Since 8 is less than 12, shift 12 to the right.**
- **Place 8 in the position of the first element.**
- **Array becomes:**

```
[8, 12, 25, 31, 32, 17]
```

Fourth Iteration (index 4):



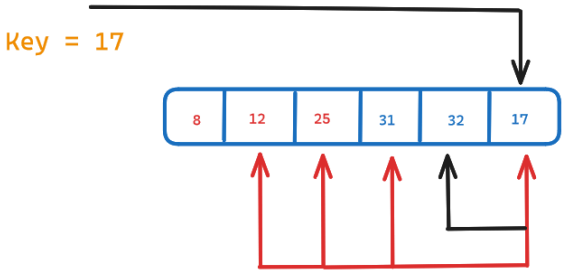
Compare 32 with 31. Since 32 is greater than 31, no shifting is needed.

Fourth Iteration (index 4):

- **Key = 32**
- **Compare 32 with 31. Since 32 is greater than 31, no shifting is needed.**
- **Array remains:**

```
[8, 12, 25, 31, 32, 17]
```

Fifth Iteration (index 5):



Compare 17 with 32. Since 17 is less than 32, shift 32 to the right.
Compare 17 with 31. Since 17 is less than 31, shift 31 to the right.
Compare 17 with 25. Since 17 is less than 25, shift 25 to the right.
Compare 17 with 12. Since 17 is greater than 12, place 17 after 12.

Fifth Iteration (index 5):

- **Key = 17**
- **Compare 17 with 32. Since 17 is less than 32, shift 32 to the right.**
- **Compare 17 with 31. Since 17 is less than 31, shift 31 to the right.**
- **Compare 17 with 25. Since 17 is less than 25, shift 25 to the right.**
- **Compare 17 with 12. Since 17 is greater than 12, place 17 after 12.**
- **Array becomes:**

```
[8, 12, 17, 25, 31, 32]
```

```
Final Sorted Array:
```

Final Sorted Array:

[8, 12, 17, 25, 31, 32]

YOU CAN CAN SEE MY WHOLE DIAGRAM CLICKING THIS Insertion Sort.png

3.3.2 Practical Scenario :

Imagine a pile of shirts of various sizes.

- **You start by laying out the first shirt (consider it small). Then, you pick up another shirt.**
- **If it's smaller than the first one, you move the first shirt aside to create space and place the smaller one in front.**
- **You keep doing this, comparing each shirt to the already sorted pile and inserting it in the correct position (based on size) until all shirts are sorted from smallest to biggest.**

3.3.3 Implementation of Insertion Sort :

Main.java

```java
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--;
        }
        arr[j + 1] = key;
    }
}

    public static void main(String args[]) {
        int[] arr = {45, 50, 98, 41};
        System.out.println(Arrays.toString(arr));    //[45, 50, 98, 41]

        insertionSort(arr);
        System.out.println(Arrays.toString(arr));    //[41, 45, 50, 98]
    }
}
```

3.3.4 Advantages of Insertion Sort:

- **Simple and easy to implement.**
- **Stable sorting algorithm.**
- **Efficient for small lists and nearly sorted lists.**
- **Space-efficient.**

3.3.5 Disadvantages of Insertion Sort:

- **Inefficient for large lists.**
- **Not as efficient as other sorting algorithms (e.g., merge sort, quick sort) for most cases.**

3.3.6 When is the Insertion Sort algorithm used?

Insertion sort is used when the number of elements is small. It can also be useful when the input array is almost sorted, and only a few elements are misplaced in a complete big array.

Time Complexity of Insertion Sort

- **Best case: O(n), If the list is already sorted, where n is the number of elements in the list.**
- **Average case: $O(n^2)$, If the list is randomly ordered**
- **Worst case: $O(n^2)$, If the list is in reverse order**

3.4 Merge Sort

Merge sort is a sorting algorithm that follows the divide-and-conquer approach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array. It has a time complexity of O(n log n) in all cases, making it efficient for large datasets.

In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge

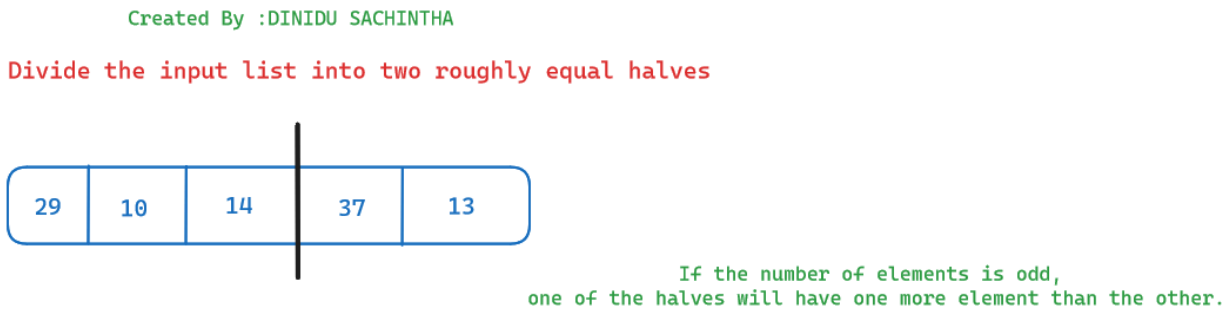
the sorted halves back together. This process is repeated until the entire array is sorted.

3.4.1 How does Merge Sort Work without programming?

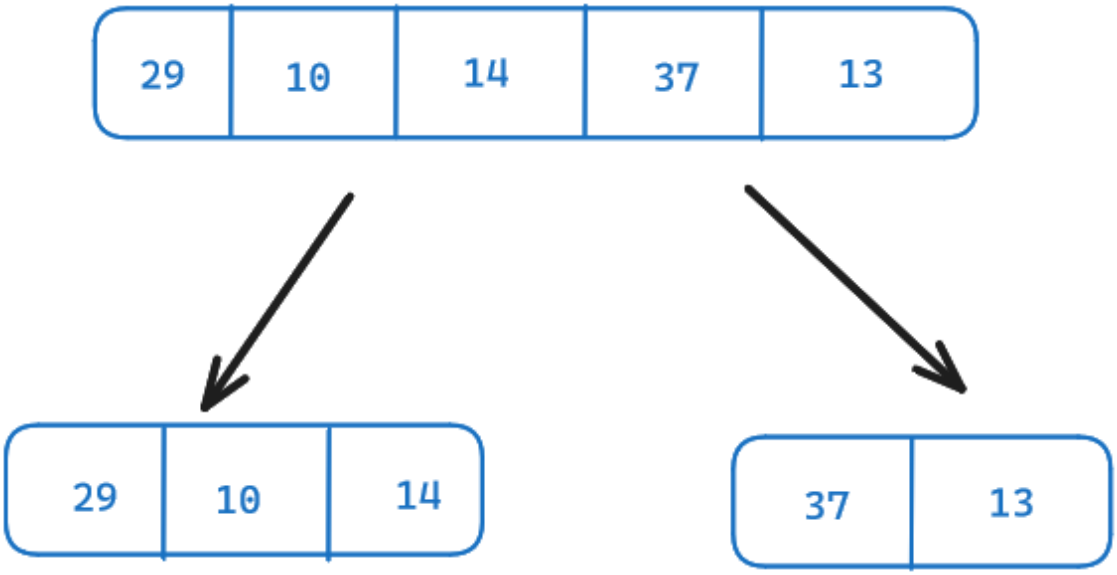
Here's a step-by-step explanation of how merge sort works:

1. **Divide: Divide the list or array recursively into two halves until it can no more be divided.**
2. **Conquer: Each subarray is sorted individually using the merge sort algorithm.**
3. **Merge: The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.**

Merge Sort.png

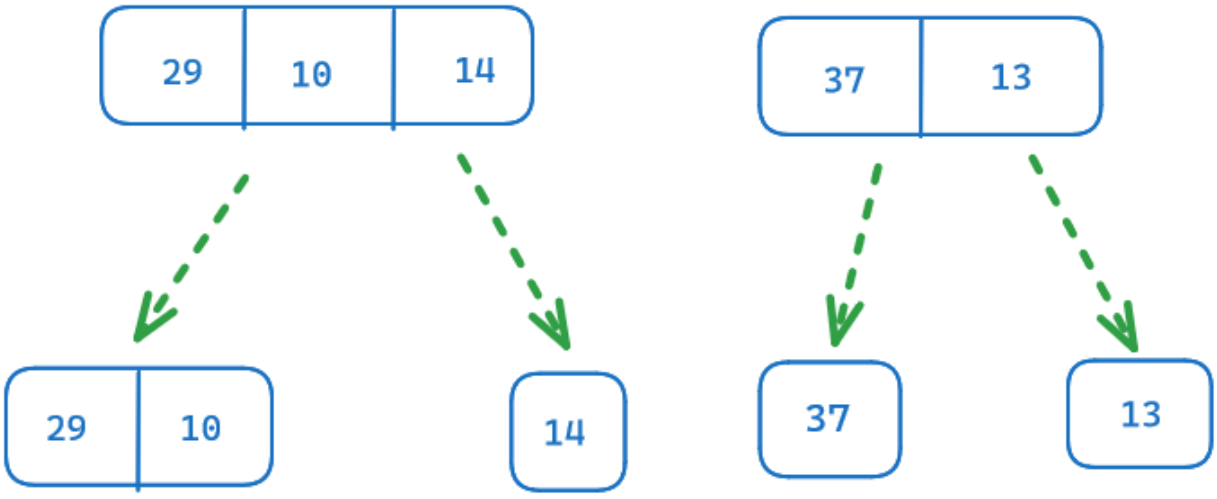


1. Start with an Unsorted List: Imagine you have a list of numbers that you want to sort in ascending order. For example, let's use the list: [29, 10, 14, 37, 13].



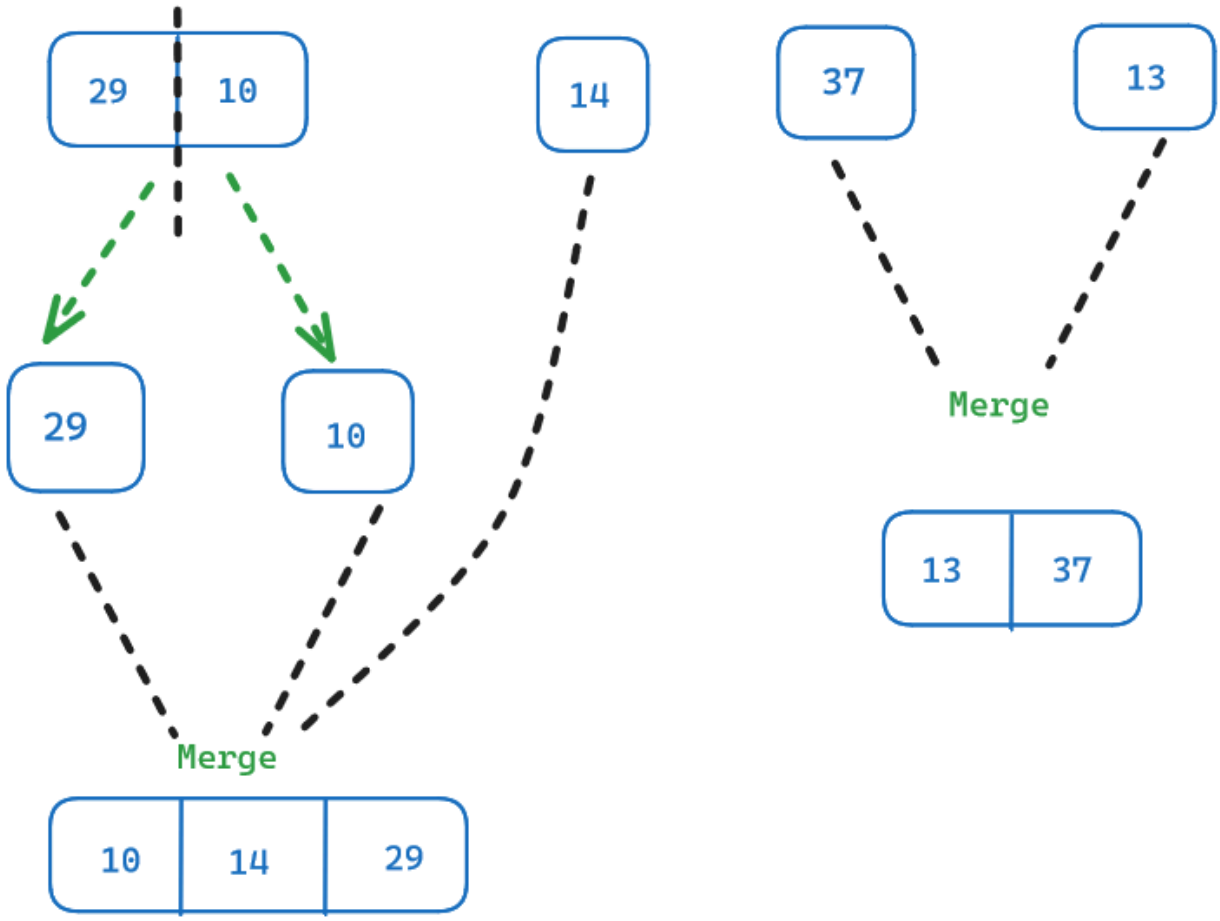
2. Divide the List into Two Halves: Divide the input list into two roughly equal halves. If the number of elements is odd, one of the halves will have one more element than the other.

- **First half: [29, 10, 14]**
- **Second half: [37, 13]**



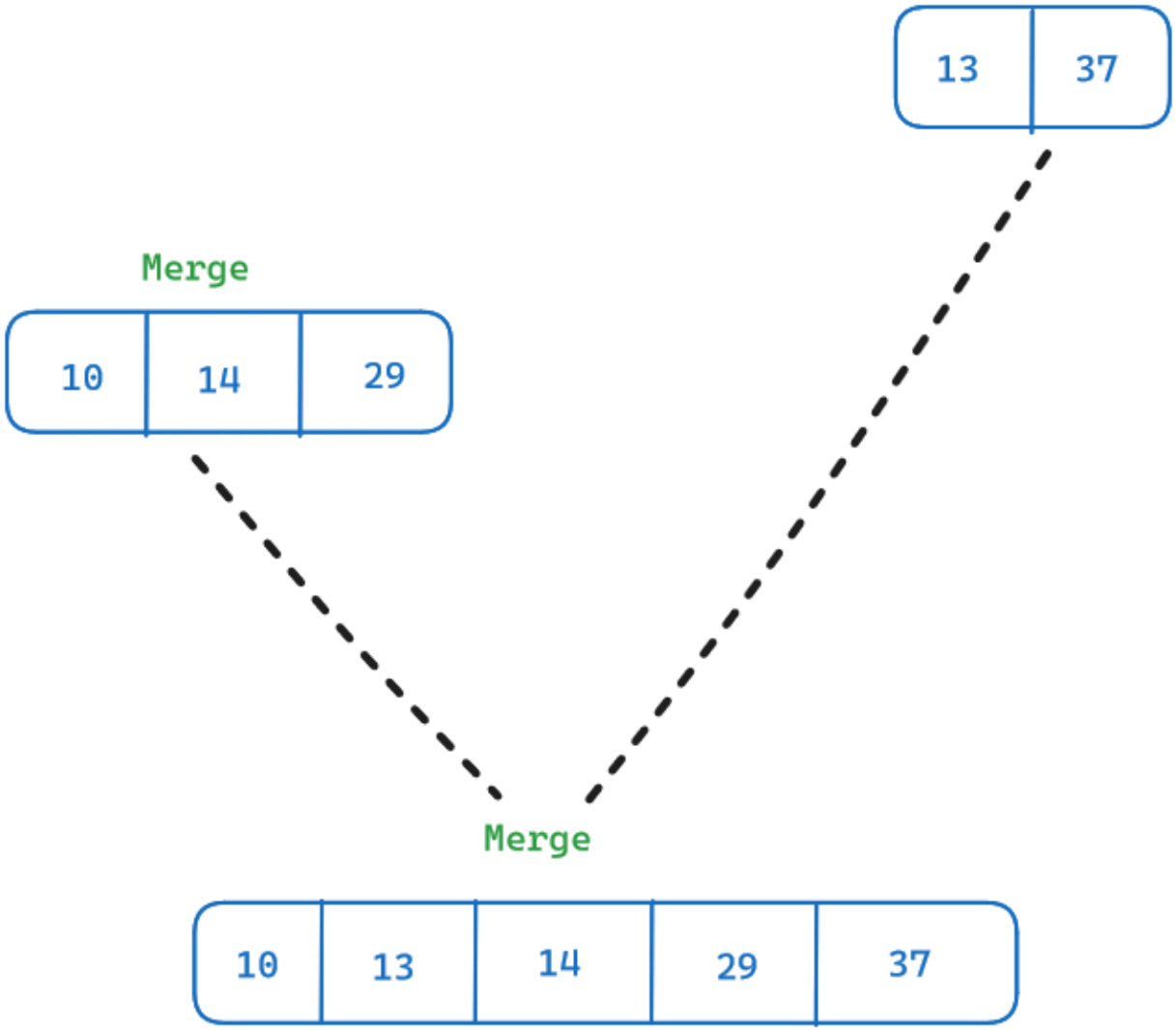
3. Recursively Sort the Halves: Apply the merge sort algorithm recursively to sort each half of the list.

- **Sort the first half: [10, 14, 29]**
- **Sort the second half: [13, 37]**



4. Merge the Sorted Halves: Merge the sorted halves to produce a single sorted array.

- **Compare the first elements of both halves (10 and 13).**
- **Since 10 is smaller, it goes first in the merged list.**
- **Continue comparing and merging elements until both halves are exhausted.**



Repeat Until Fully Sorted: Repeat steps 2 to 4 until each half contains only one element. At this point, each individual element is considered sorted by itself.

Visualizing Merge Sort

1. **Initial List: [29, 10, 14, 37, 13]**
2. **Divide the List:**
    - **First half: [29, 10, 14]**
    - **Second half: [37, 13]**
3. **Recursively Sort the Halves:**
    - **Sort the first half: [10, 14, 29]**
    - **Sort the second half: [13, 37]**
4. **Merge the Sorted Halves:**
    - **Compare 10 and 13. Place 10 first.**
    - **Compare 14 and 13. Place 13 next.**
    - **Compare 14 and 37. Place 14 next.**
    - **Compare 29 and 37. Place 29 next.**
    - **The merged list: [10, 13, 14, 29, 37]**
5. **Final Sorted List: [10, 13, 14, 29, 37]**

YOU CAN CAN SEE MY WHOLE DIAGRAM CLICKING THIS Merge Sort.png

3.4.2 Practical Scenario :

Imagine sorting a stack of books by genre (fiction, non-fiction). Merge Sort works like this:

- **Divide the stack into two roughly equal halves.**
- **Ask two people to independently sort their assigned halves by genre (like separating fiction and non-fiction).**

- **Once both halves are sorted, take turns picking the book with the alphabetically 'earlier' genre (e.g., fiction comes before non-fiction) from each sorted half and placing it on a new stack (the final sorted stack).**
- **Continue this process until one half is empty, then add the remaining books from the other half to the new stack.**

3.4.3 Implementation of Merge Sort

Main.java

```java
public class MergeSort {
    public static 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);
        }
    }

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

        int[] leftArr = new int[n1];
        int[] rightArr = new int[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++];
            } else {
                arr[k++] = rightArr[j++];
            }
        }

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

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

    public static void main(String[] args) {
        int[] arr = {12, 11, 13, 5, 6, 7};
        mergeSort(arr, 0, arr.length - 1);

        for (int num : arr) {
            System.out.print(num + " "); // 5 6 7 11 12 13
        }
    }
}
```

3.4.4 Advantages of Merge Sort:

- **Stability:** **Merge sort is a stable sorting algorithm, which means it maintains the relative order of equal elements in the input array.**
- **Guaranteed worst-case performance:** **Merge sort has a worst-case time complexity of O(N logN), which means it performs well even on large datasets.**
- **Simple to implement:** **The divide-and-conquer approach is straightforward.**

3.4.5 Disadvantage of Merge Sort:

- **Space complexity:** **Merge sort requires additional memory to store the merged sub-arrays during the sorting process.**
- **Not in-place:** **Merge sort is not an in-place sorting algorithm, which means it requires additional memory to store the sorted data. This can be a disadvantage in applications where memory usage is a concern.**

3.4.6 Applications of Merge Sort:

Merge Sort is a versatile sorting algorithm used in various computing applications due to its efficiency and unique properties. Here are some prominent use cases:

- **Large Datasets:** **Merge Sort shines when dealing with massive datasets because its time complexity is O(n log n), which is generally faster than algorithms like Bubble Sort (O(n^2)) for larger data sizes.**

- **Linked Lists:** **Merge Sort is particularly well-suited for sorting linked lists. Unlike some other sorting algorithms that require random access and frequent element movement (which can be slow for linked lists), Merge Sort can be implemented efficiently without modifying the underlying structure of the linked list, maintaining its space complexity at O(1).**

- **External Sorting:** **When dealing with datasets that are too large to fit in main memory, Merge Sort is a popular choice for external sorting algorithms. This involves breaking the data into smaller chunks, sorting them on disk, and then merging the sorted chunks back together in a step-by-step manner.**

- **Stable Sorting:** **Merge Sort is a stable sorting algorithm, meaning it preserves the original order of equal elements. This can be crucial in situations where the order within groups of identical values matters. For instance, sorting a list of customer records by name while maintaining the order of customers with the same name might be a requirement, and Merge Sort would be a suitable choice for such a scenario.**

3.5 Heap Sort

Heap sort is a comparison-based sorting technique based on the Binary Heap data structure. It is similar to the selection sort where we first find the maximum element and place it at the end of the array. Repeat the same process for the remaining elements.

The key process in Heap Sort is to build a heap from the input data and then repeatedly remove the maximum element from the heap and reconstruct the heap until all elements are sorted.

3.5.1 How does Heap Work without programming?

1. **Build a Max-Heap: Rearrange the array into a binary heap.**
2. **Swap the root with the last item of the heap: Move the current largest value to the end of the array.**
3. **Reduce the size of the heap by one: Exclude the last item from the heap and heapify the root again to get the next largest value.**
4. **Repeat until the heap is reduced to one item.**

Building the Heap

The first step in the Heapsort algorithm is to build a max-heap from the input array. This is done using the heapify procedure.

The heapify procedure works as follows:

1. **Start from the first non-leaf node in the array, which is at index (n/2) - 1 for an array of size n. This is because leaf nodes (nodes without children) are already heaps by themselves.**

2. **For each non-leaf node at index i, perform the following steps:**
   a. Compare the value at i with its children at indices 2*i + 1 and 2*i + 2 (if they exist).
   b. If the value at i is smaller than either of its children, swap it with the larger child.
   c. After swapping, recursively heapify the subtree rooted at the child index where the swap occurred.

3. **Repeat step 2 for all non-leaf nodes, starting from the last non-leaf node and moving upwards towards the root.**

This process essentially converts the array into a max-heap by pushing the largest elements towards the root, while maintaining the max-heap property at each level. The time complexity of building the max-heap is O(n), as each node is visited at most once during the heapify process.

Extracting Elements from the Heap

After building the max-heap, the root node contains the maximum value in the heap. To sort the array, we repeatedly extract the maximum element from the heap and place it at the end of the array. The steps are as follows:

1. **Swap the root node (maximum value) with the last element of the heap.**
2. **Reduce the heap size by one, effectively removing the last element (which is now the maximum value).**
3. **Heapify the root node to maintain the max-heap property for the reduced heap.**
4. **Repeat steps 1-3 until all elements are extracted from the heap.**

The time complexity of extracting all n elements from the heap is O(n log n), as each extraction takes O(log n) time due to the heapify operation.

Sorted Array

After extracting all elements from the heap, the array will be sorted in descending order. To get the sorted array in ascending order, we can simply reverse the extracted elements.

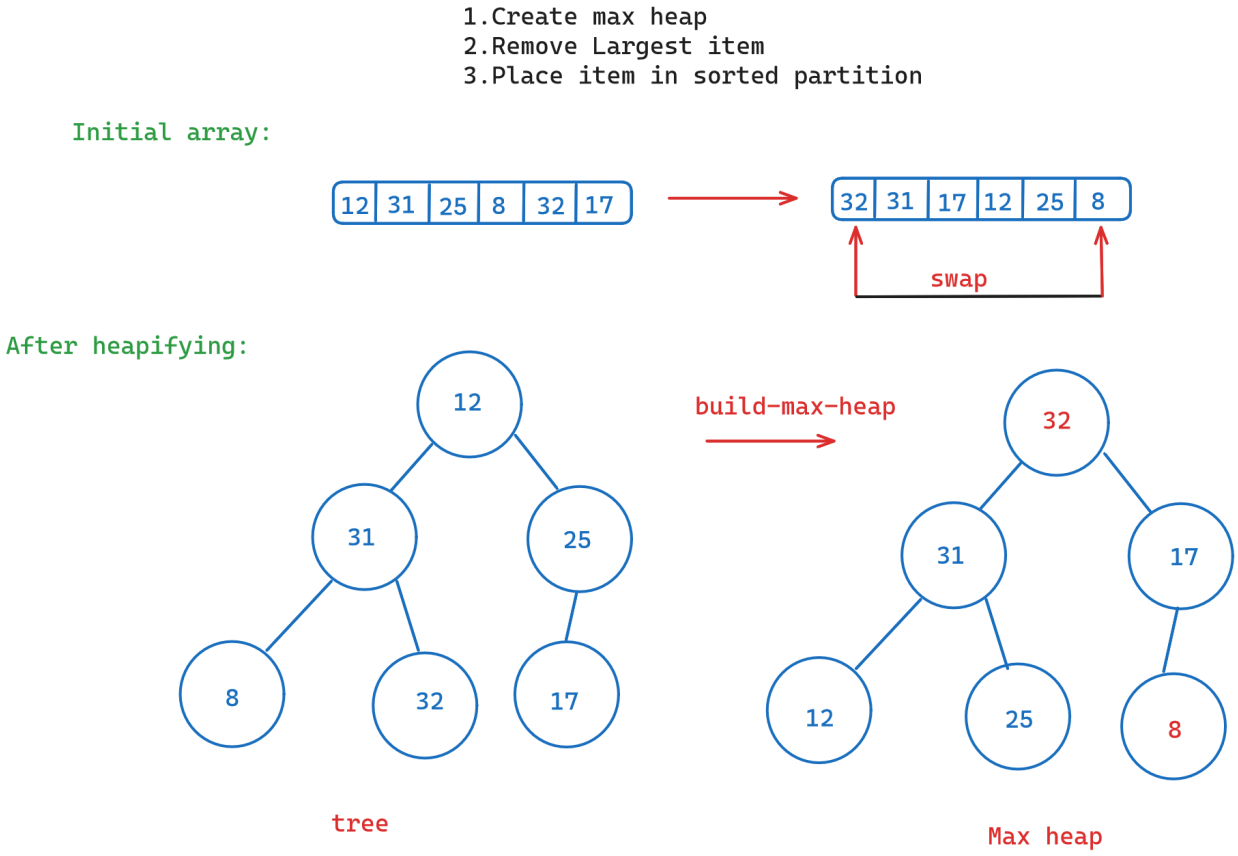
Here's the step-by-step process of Heap Sort on the array :

Heap Sort.png

Step 1: Build a Max Heap

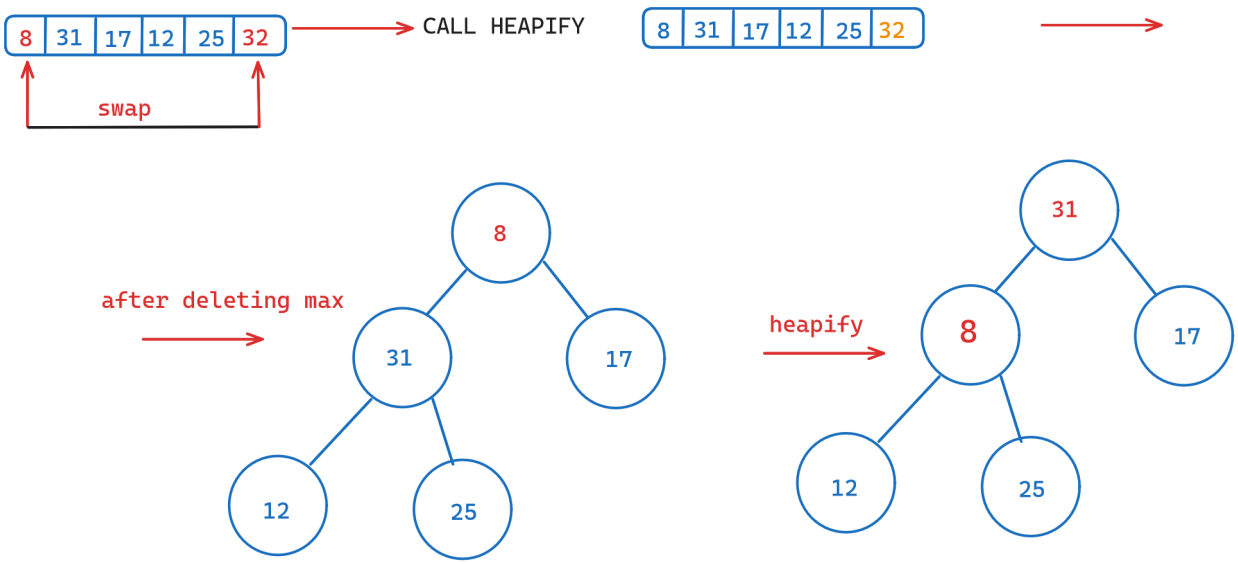
We start by building a max heap from the input array. A max heap is a binary tree where the value of each node is greater than or equal to the values of its children. To build the max heap, we start from the middle of the array and work our way backward, heapifying each sub-tree.

`Initial array: [12, 31, 25, 8, 32, 17]`



Step 2: Extract Maximum Element and Swap with Last Element

After building the max heap, we repeatedly extract the maximum element (root node) from the heap and swap it with the last element in the array. Then, we heapify the remaining elements to maintain the max heap property.

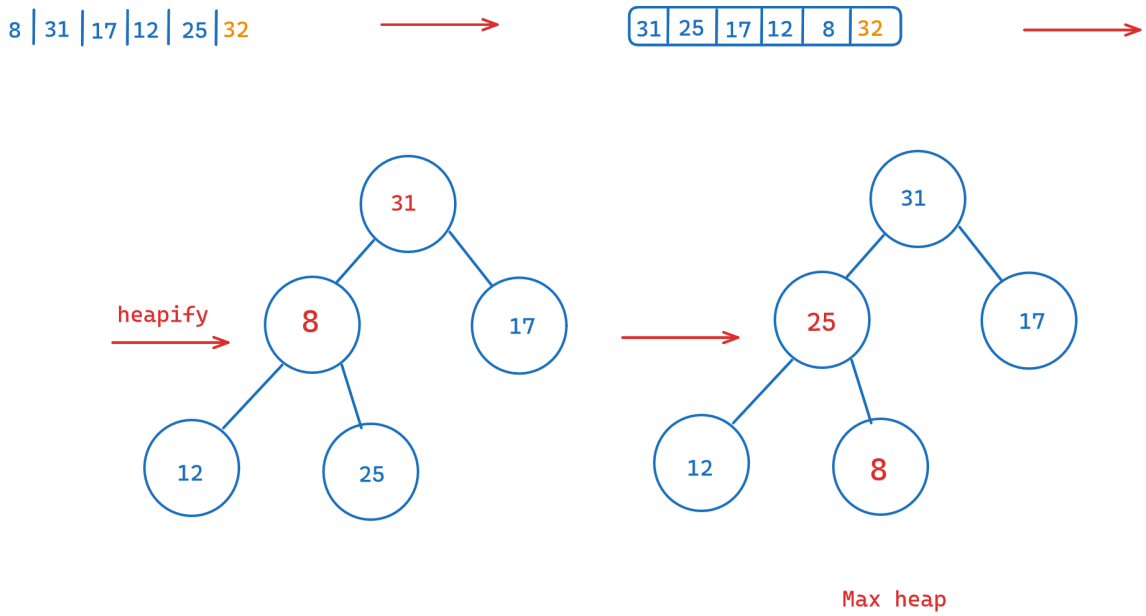


3. Extract Maximum and Heapify:

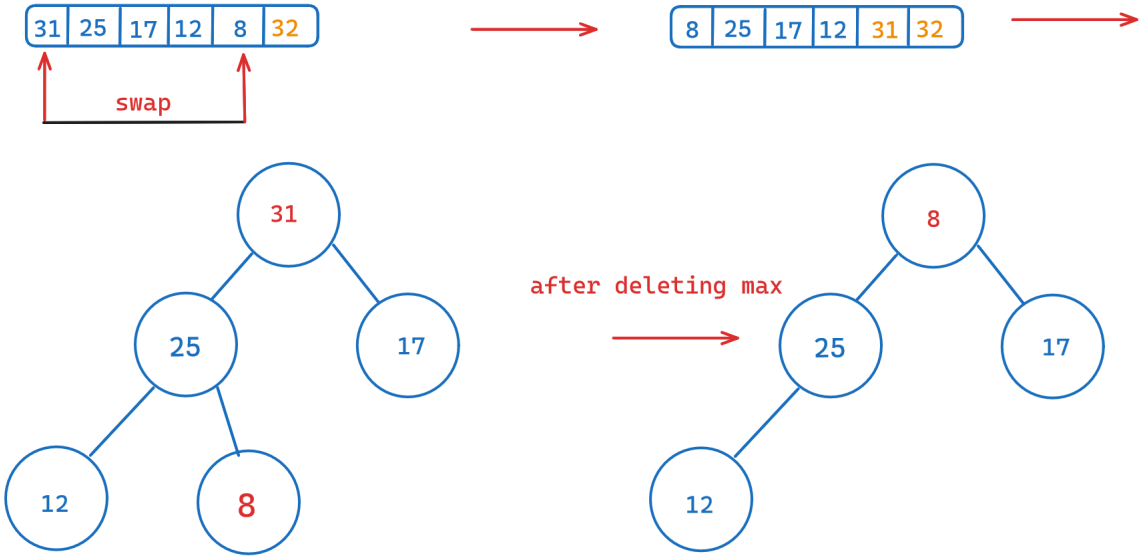
Now that we have a Max Heap, we can start extracting the largest element (at the root) and placing it in its sorted position.

1. **Swap the root element (which is the maximum in the Max Heap) with the last element in the array. In this**

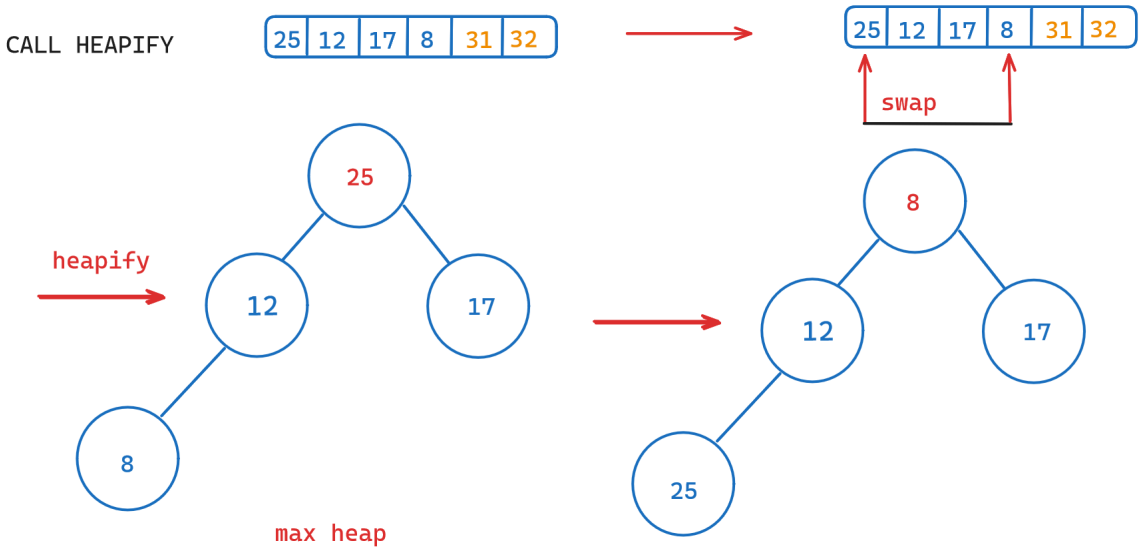
example, swap `32` with `8`, resulting in `[8, 31, 25, 12, 17, 32]`.



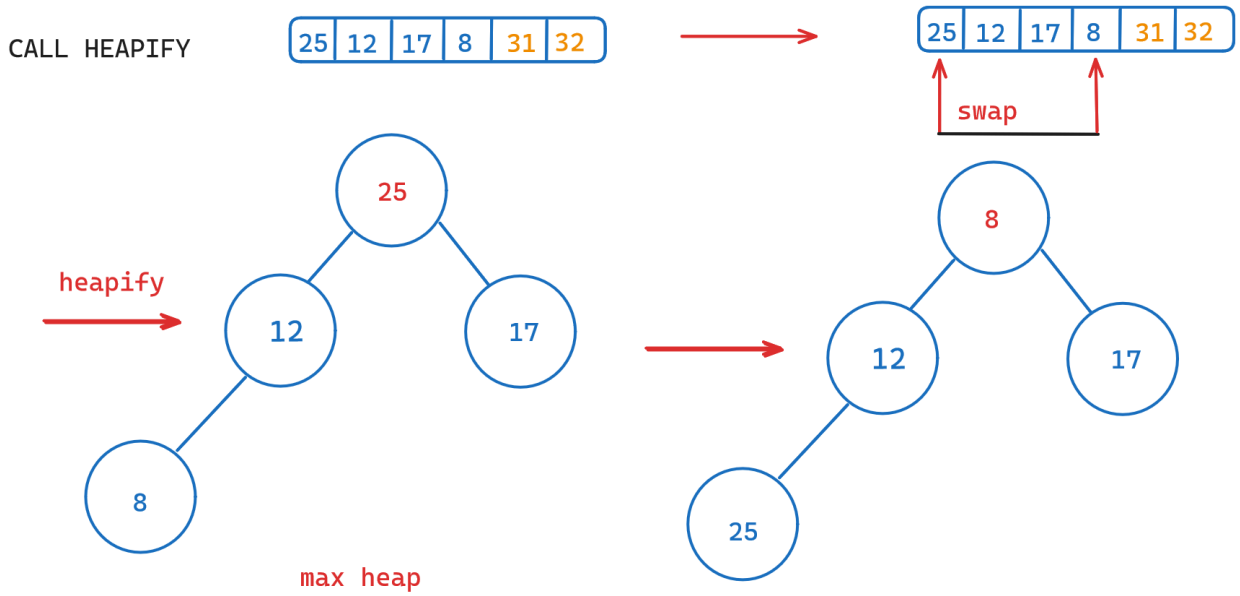
2. We've effectively placed the largest element (`32`) in its final position (last element in the sorted array). Now, we need to fix the remaining elements (sub-array `[8, 31, 25, 12, 17]`) to maintain the Max Heap property.



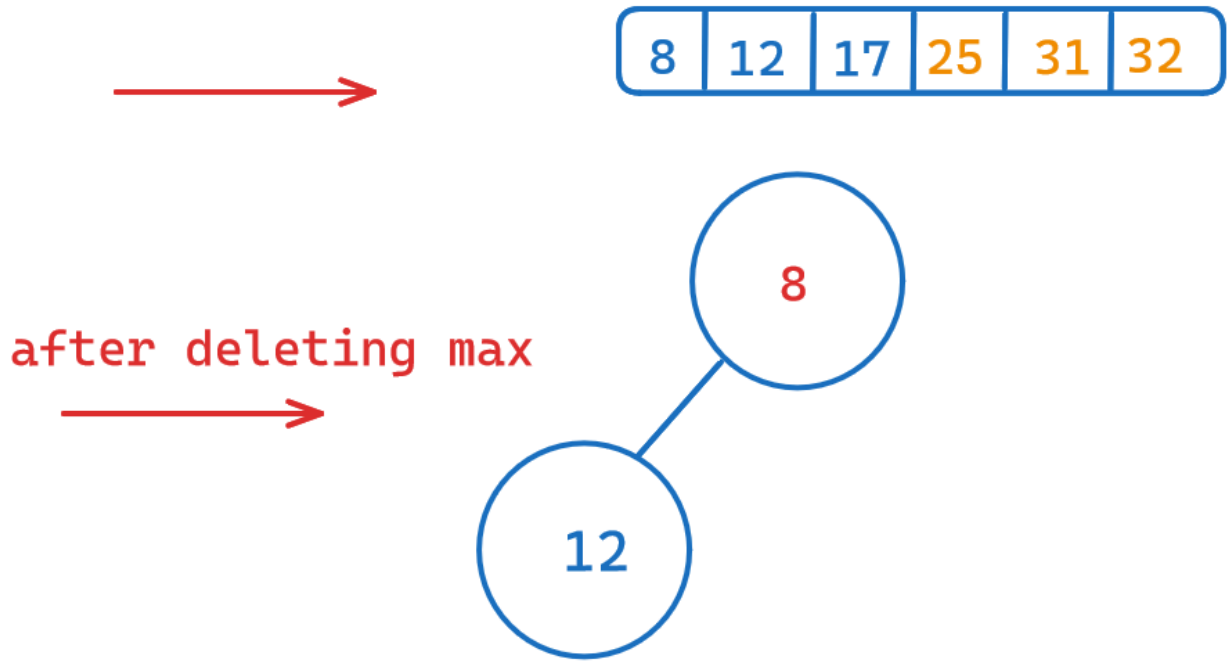
3. **Apply `heapify` on this sub-array. `Heapify` ensures the largest element in the sub-array bubbles up to the root, making it a valid Max Heap again. In this case, `heapify` might rearrange the sub-array to `[31, 25, 17, 12, 8]`.**



4. Repeat Extraction and Heapify:

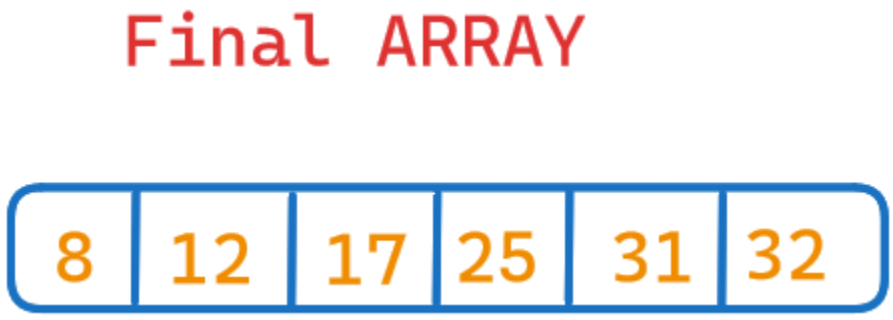


Repeat steps 3.1 (swap root with last element) and 3.3 (heapify on remaining sub-array) until there's only one element left in the array (which is already sorted).



5. Sorted Array:

After repeating the extraction and heapify process, you'll end up with the sorted array: `[8, 12, 17, 25, 31, 32]`.



Note:

The specific order of elements in the Max Heap during the build phase (step 2) might vary depending on the implementation. However, the core concept of rearranging elements to follow the Max Heap property and then repeatedly extracting the maximum and heapifying the remaining sub-array remains the same throughout the sorting process.

YOU CAN CAN SEE MY WHOLE DIAGRAM BY CLICKING THIS Heap Sort.png

3.5.2 Practical Scenario Analogy:

Imagine sorting a pile of books by thickness using Heap Sort:

1. **Build a Max-Heap: Think of piling up books such that the thickest book is on top (root of the heap).**

2. **Move the thickest book to the sorted section: Swap the thickest book with the book at the end of the unsorted section.**

3. **Heapify the remaining books: Reconstruct the heap with the remaining unsorted books to bring the next thickest book to the root.**

4. **Repeat the process: Continue moving the thickest book from the heap to the sorted section until all books are sorted by thickness.**

3.5.3 Implementation of Heap Sort :

Main.java

```java
public class HeapSort {
    public static void sort(int[] arr) {
        int n = arr.length;

        // Build max heap
        for (int i = n / 2 - 1; i >= 0; i--)
            heapify(arr, n, i);

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

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

    // To heapify a subtree rooted at index i
    private static void heapify(int[] arr, int n, int i) {
        int largest = i; // Initialize largest as root
        int l = 2 * i + 1; // Left child
        int r = 2 * i + 2; // Right child

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

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

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

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

3.5.4 Advantages of Heap Sort :

- **Efficient Time Complexity: Heap Sort has a time complexity of O(n log n) in all cases. This makes it efficient for sorting large datasets. The log n factor comes from the height of the binary heap, and it ensures that the algorithm maintains good performance even with a large number of elements.**

- **Memory Usage – Memory usage can be minimal (by writing an iterative heapify() instead of a recursive one). So apart from what is necessary to hold the initial list of items to be sorted, it needs no additional memory space to work**

- **Simplicity – It is simpler to understand than other equally efficient sorting algorithms because it does not use advanced computer science concepts such as recursion.**

3.5.5 Disadvantages of Heap Sort :

- **Costly:** **Heap sort is costly as the constants are higher compared to merge sort even if the time complexity is O(n Log n) for both.**

- **Unstable:** **Heap sort is unstable. It might rearrange the relative order.**

- **Efficient:** **Heap Sort is not very efficient when working with highly complex data.**

3.3.6 When is the Heap Sort algorithm used?

Heap Sort is an efficient and versatile sorting algorithm that is used in various scenarios due to its time complexity of O(n log n). Here are some key practical applications and scenarios where Heap Sort is commonly used:

1. **Sorting large datasets**

   Heap Sort's O(n log n) time complexity makes it suitable for sorting large datasets efficiently, such as databases with millions of records or large data analytics tasks. It outperforms simple sorting algorithms like Bubble Sort (O(n^2)) for large inputs.

2. **Priority Queue implementation**

   The binary heap data structure used in Heap Sort can be leveraged to implement priority queues, which are useful in operating systems, network traffic control, and job scheduling systems where tasks need to be processed based on priorities.

3. **Order statistics**

   Heap Sort can be used to efficiently find the kth smallest or largest element in a dataset, which has applications in areas like finding top-k elements, outlier detection, and selection algorithms.

4. **External sorting**

   When dealing with datasets that are too large to fit in main memory, Heap Sort can be used as part of external sorting algorithms that involve merging sorted chunks of data from disk.

5. **Real-time systems**

   In real-time systems with limited memory, such as embedded systems or routers, Heap Sort can be preferred over other efficient sorting algorithms like Merge Sort, which requires additional memory for merging.

6. **Graphics and gaming**

   Heap Sort is used in computer graphics applications for operations like building priority render queues and in gaming for sorting sprite ordering based on depth for visibility determination.

While Heap Sort may not be the fastest sorting algorithm in practice for small datasets, its efficiency, memory usage characteristics, and ability to handle large datasets make it a valuable algorithm in various domains where sorting large amounts of data is required.

4. Conclusion

In conclusion, this document has provided a comprehensive overview of several key sorting algorithms implemented in Java, focusing on Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, and Heap Sort. Each algorithm has been thoroughly explained, including their theoretical foundations, practical implementations in Java, time and space complexities, as well as their advantages and disadvantages. By understanding these sorting algorithms, developers can make informed decisions regarding their use in various applications, optimizing performance and enhancing efficiency.

5. Appendix

Appendix: Java Code Implementations

Below are the Java implementations for the Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, and Heap Sort algorithms discussed in this document:

1. **Bubble Sort Implementation**
2. **Selection Sort Implementation**
3. **Insertion Sort Implementation**
4. **Merge Sort Implementation**
5. **Heap Sort Implementation**

These implementations can serve as practical references for developers looking to integrate these sorting algorithms into their Java projects.

6. Resources

1. **"Sorting Algorithms" - GeeksforGeeks (Online Resource)**
2. **“Java Sorting Algorithms" - Tutorialspoint (Online Resource)**
3. **“Heap Sort Algorithm Explained" - Programiz (Online Resource)**

https://www.geeksforgeeks.org/sorting-algorithms/
https://www.javatpoint.com/sorting-algorithms