

Sorting algorithm



Now explain me all sorting algorithm with its working using example in very simple way so that i can understand exact concept clearly and also provide java optimized and simple code without enhanced for loop and also provide time and space complexity

- 1] Bubble sort
 - 2] Selection sort
 - 3] Insertion sort
 - 4] Merge sort
 - 5] Quick sort
- if any remains add it also



Memory updated

Let's break down each sorting algorithm with examples and Java code.

1. Bubble Sort

Concept: Bubble Sort repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the list is sorted.

Example: Consider the list `[5, 3, 8, 4, 2]`.

- **Pass 1:** `[3, 5, 4, 2, 8]` (Swaps `5` and `3`, `5` and `4`, `5` and `2`)
- **Pass 2:** `[3, 4, 2, 5, 8]` (Swaps `4` and `2`)
- **Pass 3:** `[3, 2, 4, 5, 8]` (Swaps `3` and `2`)
- **Pass 4:** `[2, 3, 4, 5, 8]` (No swaps needed)

Java Code:

java

```
public class BubbleSort {
    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 main(String[] args) {
        int[] arr = {5, 3, 8, 4, 2};
        bubbleSort(arr);
        for (int num : arr) {
            System.out.print(num + " ");
        }
    }
}
```

```
}
}
```

Time Complexity:

- Worst-case: $O(n^2)$
- Best-case: $O(n)$ (when the array is already sorted)

Space Complexity: $O(1)$ (in-place sort)

2. Selection Sort

Concept: Selection Sort repeatedly selects the minimum element from the unsorted part and moves it to the sorted part.

Example: Consider the list `[5, 3, 8, 4, 2]`.

- **Pass 1:** `[2, 3, 8, 4, 5]` (Selects `2` as the minimum and swaps with the first element)
- **Pass 2:** `[2, 3, 8, 4, 5]` (Selects `3` as the minimum, no swap needed)
- **Pass 3:** `[2, 3, 4, 8, 5]` (Selects `4` as the minimum and swaps with `8`)
- **Pass 4:** `[2, 3, 4, 5, 8]` (Selects `5` as the minimum, no swap needed)

Java Code:

```
java

public class SelectionSort {
    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;
                }
            }
            int temp = arr[minIndex];
            arr[minIndex] = arr[i];
            arr[i] = temp;
        }
    }

    public static void main(String[] args) {
        int[] arr = {5, 3, 8, 4, 2};
        selectionSort(arr);
        for (int num : arr) {
            System.out.print(num + " ");
        }
    }
}
```

Time Complexity:

- Worst-case: $O(n^2)$
- Best-case: $O(n^2)$ (always performs the same number of comparisons)

Space Complexity: $O(1)$ (in-place sort)

3. Insertion Sort

Concept: Insertion Sort builds the final sorted array one item at a time by comparing and inserting each element into its correct position.

Example: Consider the list `[5, 3, 8, 4, 2]`.

- **Pass 1:** `[3, 5, 8, 4, 2]` (Inserts `3` into the sorted part)
- **Pass 2:** `[3, 5, 8, 4, 2]` (Inserts `8` into the sorted part)
- **Pass 3:** `[3, 5, 4, 8, 2]` (Inserts `4` into the sorted part)
- **Pass 4:** `[2, 3, 4, 5, 8]` (Inserts `2` into the sorted part)

Java Code:

```
java

public class InsertionSort {
    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 = {5, 3, 8, 4, 2};
        insertionSort(arr);
        for (int num : arr) {
            System.out.print(num + " ");
        }
    }
}
```

Time Complexity:

- Worst-case: $O(n^2)$
- Best-case: $O(n)$ (when the array is already sorted)

Space Complexity: $O(1)$ (in-place sort)

4. Merge Sort

Concept: Merge Sort divides the array into two halves, recursively sorts them, and then merges the sorted halves to produce a sorted array.

Example: Consider the list `[5, 3, 8, 4, 2]`.

- **Split:** `[5, 3, 8]` and `[4, 2]`
- **Further Split:** `[5]`, `[3, 8]` and `[4]`, `[2]`
- **Sort and Merge:** `[3, 5, 8]` and `[2, 4]`
- **Final Merge:** `[2, 3, 4, 5, 8]`

Java Code:

java

```
public class MergeSort {
    public static void mergeSort(int[] arr) {
        if (arr.length < 2) return;
        int mid = arr.length / 2;
        int[] left = new int[mid];
        int[] right = new int[arr.length - mid];

        System.arraycopy(arr, 0, left, 0, mid);
        System.arraycopy(arr, mid, right, 0, arr.length - mid);

        mergeSort(left);
        mergeSort(right);

        merge(arr, left, right);
    }

    private static void merge(int[] arr, int[] left, int[] right) {
        int i = 0, j = 0, k = 0;
        while (i < left.length && j < right.length) {
            if (left[i] <= right[j]) {
                arr[k++] = left[i++];
            } else {
                arr[k++] = right[j++];
            }
        }
        while (i < left.length) arr[k++] = left[i++];
        while (j < right.length) arr[k++] = right[j++];
    }

    public static void main(String[] args) {
        int[] arr = {5, 3, 8, 4, 2};
        mergeSort(arr);
        for (int num : arr) {
            System.out.print(num + " ");
        }
    }
}
```

Time Complexity:

- Worst-case: $O(n \log n)$
- Best-case: $O(n \log n)$

Space Complexity: $O(n)$ (requires additional space for merging)

5. Quick Sort

Concept: Quick Sort selects a "pivot" element and partitions the array into two sub-arrays: elements less than the pivot and elements greater than the pivot. It then recursively sorts the sub-arrays.

Example: Consider the list `[5, 3, 8, 4, 2]`.

- **Choose Pivot:** `5`
- **Partition:** `[3, 4, 2]` and `[8]`
- **Sort Sub-arrays:** `[2, 3, 4]` and `[8]`
- **Combine:** `[2, 3, 4, 5, 8]`

Java Code:

java

```
public class QuickSort {
    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);
        }
    }

    private 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 main(String[] args) {
        int[] arr = {5, 3, 8, 4, 2};
        quickSort(arr, 0, arr.length - 1);
        for (int num : arr) {
            System.out.print(num + " ");
        }
    }
}
```

Time Complexity:

- Worst-case: $O(n^2)$ (when the pivot selection is poor)
- Best-case: $O(n \log n)$

Space Complexity: $O(\log n)$ (in-place sort with stack space for recursion)

Summary

- **Bubble Sort:** Simple but inefficient for large lists ($O(n^2)$ time complexity).
- **Selection Sort:** Simple with predictable performance ($O(n^2)$ time complexity).
- **Insertion Sort:** Efficient for small or nearly sorted lists ($O(n^2)$ time complexity).
- **Merge Sort:** Consistently good performance ($O(n \log n)$ time complexity) but requires extra space.
- **Quick Sort:** Fast and efficient on average ($O(n \log n)$ time complexity) but can degrade to $O(n^2)$.

Let me know if you need more details or examples!



any remaining sorting algorithm please explain



Certainly! There are several other important sorting algorithms. Here are explanations and Java code for some of the remaining commonly used sorting algorithms:

6. Heap Sort

Concept: Heap Sort builds a binary heap from the input data and repeatedly extracts the maximum element from the heap to build the sorted list.

Example: Consider the list `[5, 3, 8, 4, 2]`.

- **Build Max Heap:** `[8, 4, 5, 3, 2]`
- **Extract Max:** `[5, 4, 3, 2, 8]` (8 is placed in the correct position)
- **Repeat:** `[4, 3, 2, 5, 8]`
- **Final Sorted Array:** `[2, 3, 4, 5, 8]`

Java Code:

java

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

    private 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 main(String[] args) {
        int[] arr = {5, 3, 8, 4, 2};
        heapSort(arr);
        for (int num : arr) {
            System.out.print(num + " ");
        }
    }
}
```

Time Complexity:

- Worst-case: $O(n \log n)$
- Best-case: $O(n \log n)$

Space Complexity: $O(1)$ (in-place sort)

7. Counting Sort

Concept: Counting Sort counts the occurrences of each distinct element and then uses this count to place elements in the correct position.

Example: Consider the list `[5, 3, 8, 4, 2]`.

- **Count Occurrences:** `[2: 1, 3: 1, 4: 1, 5: 1, 8: 1]`
- **Place Elements:** `[2, 3, 4, 5, 8]`

Java Code:

```
java

public class CountingSort {
    public static void countingSort(int[] arr) {
        int max = findMax(arr);
        int[] count = new int[max + 1];

        for (int num : arr) {
            count[num]++;
        }

        int index = 0;
        for (int i = 0; i < count.length; i++) {
            while (count[i] > 0) {
                arr[index++] = i;
                count[i]--;
            }
        }
    }

    private static int findMax(int[] arr) {
        int max = arr[0];
        for (int i = 1; i < arr.length; i++) {
            if (arr[i] > max) {
                max = arr[i];
            }
        }
        return max;
    }

    public static void main(String[] args) {
        int[] arr = {5, 3, 8, 4, 2};
        countingSort(arr);
        for (int num : arr) {
            System.out.print(num + " ");
        }
    }
}
```

Time Complexity:

- Worst-case: $O(n + k)$ (where k is the range of the input)
- Best-case: $O(n + k)$

Space Complexity: $O(k)$ (requires additional space proportional to the range of the input)

8. Radix Sort

Concept: Radix Sort processes integers digit by digit starting from the least significant digit to the most significant digit using a stable subroutine (like Counting Sort) to sort the digits.

Example: Consider the list `[170, 45, 75, 90, 802, 24, 2, 66]`.

- **Least Significant Digit (LSD):** `[170, 90, 802, 45, 75, 24, 66, 2]`
- **Next Digit:** `[170, 802, 24, 45, 75, 66, 2, 90]`
- **Most Significant Digit (MSD):** `[2, 24, 45, 66, 75, 90, 170, 802]`

Java Code:

java

```
public class RadixSort {
    public static void radixSort(int[] arr) {
        int max = findMax(arr);
        for (int exp = 1; max / exp > 0; exp *= 10) {
            countingSort(arr, exp);
        }
    }

    private static void countingSort(int[] arr, int exp) {
        int n = arr.length;
        int[] output = new int[n];
        int[] count = new int[10];

        for (int i = 0; i < n; i++) {
            count[(arr[i] / exp) % 10]++;
        }

        for (int i = 1; i < 10; i++) {
            count[i] += count[i - 1];
        }

        for (int i = n - 1; i >= 0; i--) {
            output[count[(arr[i] / exp) % 10] - 1] = arr[i];
            count[(arr[i] / exp) % 10]--;
        }

        System.arraycopy(output, 0, arr, 0, n);
    }

    private static int findMax(int[] arr) {
        int max = arr[0];
        for (int i = 1; i < arr.length; i++) {
            if (arr[i] > max) {
                max = arr[i];
            }
        }
        return max;
    }

    public static void main(String[] args) {
        int[] arr = {170, 45, 75, 90, 802, 24, 2, 66};
        radixSort(arr);
        for (int num : arr) {
            System.out.print(num + " ");
        }
    }
}
```

Time Complexity:

- Worst-case: $O(nk)$ (where k is the number of digits in the largest number)
- Best-case: $O(nk)$

Space Complexity: $O(n + k)$ (requires additional space for the counting and output arrays)

9. Bucket Sort

Concept: Bucket Sort distributes elements into several buckets and then sorts each bucket individually, either using another sorting algorithm or recursively applying Bucket Sort.

Example: Consider the list `[0.78, 0.17, 0.39, 0.26, 0.72, 0.94, 0.68, 0.55, 0.47, 0.89]`.

- **Distribute into Buckets:** `[0.17, 0.26, 0.39, 0.47, 0.55, 0.68, 0.72, 0.78, 0.89, 0.94]`
- **Sort Buckets:** (using Insertion Sort or other method)
- **Combine:** `[0.17, 0.26, 0.39, 0.47, 0.55, 0.68, 0.72, 0.78, 0.89, 0.94]`

Java Code:

java

```
import java.util.ArrayList;
import java.util.Collections;

public class BucketSort {
    public static void bucketSort(float[] arr) {
        int n = arr.length;
        if (n <= 0) return;

        // Create empty buckets
        @SuppressWarnings("unchecked")
        ArrayList<Float>[] buckets = new ArrayList[n];
        for (int i = 0; i < n; i++) {
            buckets[i] = new ArrayList<>();
        }

        // Distribute elements into buckets
        for (float num : arr) {
            int bucketIndex = (int) (num * n);
            buckets[bucketIndex].add(num);
        }

        // Sort each bucket and concatenate results
        int index = 0;
        for (ArrayList<Float> bucket : buckets) {
            Collections.sort(bucket);
            for (float num : bucket) {
                arr[index++] = num;
            }
        }
    }

    public static void main(String[] args) {
        float[] arr = {0.78f, 0.17f, 0.39f, 0.26f, 0.72f, 0.94f, 0.68f, 0.55f, 0.47f,
0.89f};
        bucketSort(arr);
        for (float num : arr) {
            System.out.print(num + " ");
        }
    }
}
```

Time Complexity:

- Worst-case: $O(n^2)$ (when bucket sizes are not balanced)

- Best-case: $O(n + k)$ (when buckets are well-distributed)

Space Complexity: $O(n + k)$ (additional space for buckets)

These are the fundamental sorting algorithms you might encounter. Each has its use cases, advantages, and trade-offs. If you need any more details or have other questions, feel free to ask!

ChatGPT can make mistakes. Check important info.