**Sorting** **Techniques**

Java provides several sorting techniques for arrays, using both built-in methods and custom implementations. Here's a comprehensive list:

**1. Built-in Sorting Techniques**

Java's standard library includes highly optimized methods for sorting arrays:

* **Arrays.sort()**
  + Sorts the array in ascending order.
  + Can handle int[], double[], char[], String[], and arrays of custom objects (if they implement Comparable or use a Comparator).
  + Example: int[] arr = {5, 2, 8, 1};

Arrays.sort(arr);

* **Arrays.parallelSort()**
  + Uses the divide-and-conquer technique for parallel sorting.
  + Faster than Arrays.sort() for large datasets on multi-core processors.
  + Example: int[] arr = {5, 2, 8, 1};

Arrays.parallelSort(arr);

**2. Custom Sorting Techniques**

### ****1. Bubble Sort****

#### **Description**

Bubble Sort repeatedly compares adjacent elements and swaps them if they are in the wrong order. It is simple but inefficient for large datasets.

#### **Time Complexity**

* Best Case: O(n)O(n)O(n) (when the array is already sorted)
* Average Case: O(n2)O(n^2)O(n2)
* Worst Case: O(n2)O(n^2)O(n2)

#### **Space Complexity**

* Space: O(1)O(1)O(1) (in-place sorting)

#### **Steps**

1. Start from the first element and compare adjacent elements.
2. Swap them if they are out of order.
3. Continue for all elements in the array.
4. Repeat the process for n−1n-1n−1 iterations, where nnn is the array size.

#### **Example in Java**

|  |
| --- |
| public class BubbleSortExample {  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 = {64, 34, 25, 12, 22, 11, 90};  bubbleSort(arr);  System.out.println("Sorted array: ");  for (int num : arr) System.out.print(num + " ");  }  } |

#### **Execution Steps**

1. Compare 64 and 34. Swap → [34, 64, 25, 12, 22, 11, 90].
2. Compare 64 and 25. Swap → [34, 25, 64, 12, 22, 11, 90].
3. Continue till the largest element (90) is at the last position.
4. Repeat n−1n-1n−1 iterations.

### ****2. Selection Sort****

#### **Description**

Selection Sort repeatedly selects the smallest (or largest) element from the unsorted part of the array and places it in the correct position.

#### **Time Complexity**

* Best Case: O(n2)O(n^2)O(n2)
* Average Case: O(n2)O(n^2)O(n2)
* Worst Case: O(n2)O(n^2)O(n2)

#### **Space Complexity**

* Space: O(1)O(1)O(1) (in-place sorting)

#### **Steps**

1. Find the smallest element in the array.
2. Swap it with the first element.
3. Repeat for the remaining unsorted array.

#### **Example in Java**

|  |
| --- |
| public class SelectionSortExample {  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;  }  }  // Swap arr[i] and arr[minIndex]  int temp = arr[minIndex];  arr[minIndex] = arr[i];  arr[i] = temp;  }  }  public static void main(String[] args) {  int[] arr = {64, 25, 12, 22, 11};  selectionSort(arr);  System.out.println("Sorted array: ");  for (int num : arr) System.out.print(num + " ");  }  } |

#### **Execution Steps**

1. Find the smallest element (11) and swap it with 64.
2. Find the next smallest (12) and swap it with 25.
3. Repeat until the array is sorted.

### ****3. Insertion Sort****

#### **Description**

Insertion Sort builds the sorted array one element at a time by placing each element in its correct position.

#### **Time Complexity**

* Best Case: O(n)O(n)O(n) (when the array is already sorted)
* Average Case: O(n2)O(n^2)O(n2)
* Worst Case: O(n2)O(n^2)O(n2)

#### **Space Complexity**

* Space: O(1)O(1)O(1) (in-place sorting)

#### **Steps**

1. Start from the second element.
2. Compare it with elements to its left.
3. Shift larger elements to the right and insert the current element in its correct position.

#### **Example in Java**

|  |
| --- |
| public class InsertionSortExample {  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 = {12, 11, 13, 5, 6};  insertionSort(arr);  System.out.println("Sorted array: ");  for (int num : arr) System.out.print(num + " ");  }  } |

#### **Execution Steps**

1. Start with 11. Compare with 12 and insert it in the correct position.
2. Move to 13. No changes needed.
3. Compare 5 with all left elements and insert it correctly.

### ****4. Merge Sort****

#### **Description**

Merge Sort is a divide-and-conquer algorithm that divides the array into smaller subarrays, sorts them, and then merges the sorted subarrays into a single sorted array.

#### **Time Complexity**

* Best Case: O(nlog⁡n)O(n \log n)O(nlogn)
* Average Case: O(nlog⁡n)O(n \log n)O(nlogn)
* Worst Case: O(nlog⁡n)O(n \log n)O(nlogn)

#### **Space Complexity**

* Space: O(n)O(n)O(n) (requires additional space for merging)

#### **Steps**

1. Divide the array into two halves recursively until each subarray contains one element.
2. Merge the two sorted halves into a single sorted array.

#### **Example in Java**

|  |
| --- |
| public class MergeSortExample {  public static void mergeSort(int[] arr, int left, int right) {  if (left < right) {  int mid = left + (right - left) / 2;  // Recursively sort the two halves  mergeSort(arr, left, mid);  mergeSort(arr, mid + 1, right);  // Merge the sorted halves  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;  // Temporary arrays  int[] L = new int[n1];  int[] R = new int[n2];  // Copy data  for (int i = 0; i < n1; i++) L[i] = arr[left + i];  for (int i = 0; i < n2; i++) R[i] = arr[mid + 1 + i];  int i = 0, j = 0, k = left;  // Merge the temp arrays  while (i < n1 && j < n2) {  if (L[i] <= R[j]) arr[k++] = L[i++];  else arr[k++] = R[j++];  }  // Copy remaining elements  while (i < n1) arr[k++] = L[i++];  while (j < n2) arr[k++] = R[j++];  }  public static void main(String[] args) {  int[] arr = {38, 27, 43, 3, 9, 82, 10};  mergeSort(arr, 0, arr.length - 1);  System.out.println("Sorted array: ");  for (int num : arr) System.out.print(num + " ");  }  } |

#### **Execution Steps**

1. Divide the array into smaller parts: [38,27,43][38, 27, 43][38,27,43] and [3,9,82,10][3, 9, 82, 10][3,9,82,10].
2. Continue dividing until subarrays have one element each.
3. Merge sorted subarrays step-by-step.

### ****5. Quick Sort****

#### **Description**

Quick Sort is a divide-and-conquer algorithm that partitions the array around a pivot such that all elements smaller than the pivot are on the left, and all elements larger are on the right. The process is then repeated recursively for the subarrays.

#### **Time Complexity**

* Best Case: O(nlog⁡n)O(n \log n)O(nlogn)
* Average Case: O(nlog⁡n)O(n \log n)O(nlogn)
* Worst Case: O(n2)O(n^2)O(n2) (occurs when the pivot is poorly chosen)

#### **Space Complexity**

* Space: O(log⁡n)O(\log n)O(logn) (for recursive stack calls)

#### **Steps**

1. Select a pivot element.
2. Partition the array such that elements smaller than the pivot go to the left and larger ones to the right.
3. Recursively apply the above steps to subarrays.

#### **Example in Java**

|  |
| --- |
| public class QuickSortExample {  public static void quickSort(int[] arr, int low, int high) {  if (low < high) {  int pi = partition(arr, low, high);  // Recursively sort elements before and after partition  quickSort(arr, low, pi - 1);  quickSort(arr, pi + 1, high);  }  }  public static int partition(int[] arr, int low, int high) {  int pivot = arr[high];  int i = low - 1;  for (int j = low; j < high; j++) {  if (arr[j] <= pivot) {  i++;  int temp = arr[i];  arr[i] = arr[j];  arr[j] = temp;  }  }  int temp = arr[i + 1];  arr[i + 1] = arr[high];  arr[high] = temp;  return i + 1;  }  public static void main(String[] args) {  int[] arr = {10, 7, 8, 9, 1, 5};  quickSort(arr, 0, arr.length - 1);  System.out.println("Sorted array: ");  for (int num : arr) System.out.print(num + " ");  }  } |

#### **Execution Steps**

1. Select 5 as the pivot.
2. Partition the array around 5.
3. Recursively sort the left and right subarrays.

### ****6. Heap Sort****

#### **Description**

Heap Sort uses a binary heap data structure to sort an array. The array is first converted into a max-heap, and the largest element is swapped with the last element. This process is repeated for the reduced heap.

#### **Time Complexity**

* Best Case: O(nlog⁡n)O(n \log n)O(nlogn)
* Average Case: O(nlog⁡n)O(n \log n)O(nlogn)
* Worst Case: O(nlog⁡n)O(n \log n)O(nlogn)

#### **Space Complexity**

* Space: O(1)O(1)O(1) (in-place sorting)

#### **Steps**

1. Build a max heap from the array.
2. Swap the root (largest element) with the last element.
3. Reduce the heap size and heapify the root.
4. Repeat until the heap size is 1.

#### **Example in Java**

|  |
| --- |
| public class HeapSortExample {  public static void heapSort(int[] arr) {  int n = arr.length;  // Build 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--) {  // Swap arr[0] with arr[i]  int temp = arr[0];  arr[0] = arr[i];  arr[i] = temp;  // Heapify the root  heapify(arr, i, 0);  }  }  public static void heapify(int[] arr, int n, int i) {  int largest = i;  int left = 2 \* i + 1;  int right = 2 \* i + 2;  if (left < n && arr[left] > arr[largest]) largest = left;  if (right < n && arr[right] > arr[largest]) largest = right;  if (largest != i) {  int swap = arr[i];  arr[i] = arr[largest];  arr[largest] = swap;  heapify(arr, n, largest);  }  }  public static void main(String[] args) {  int[] arr = {12, 11, 13, 5, 6, 7};  heapSort(arr);  System.out.println("Sorted array: ");  for (int num : arr) System.out.print(num + " ");  }  } |

#### **Execution Steps**

1. Build a max heap: [13,11,12,5,6,7][13, 11, 12, 5, 6, 7][13,11,12,5,6,7].
2. Swap 13 with the last element and heapify.
3. Repeat until sorted.

### ****7. Radix Sort****

#### **Description**

Radix Sort is a non-comparative integer sorting algorithm. It sorts numbers by processing individual digits. It processes the digits from the least significant digit (LSD) to the most significant digit (MSD). It uses a stable sorting algorithm like **Counting Sort** to sort numbers based on each digit.

#### **Time Complexity**

* Best Case: O(nk)O(nk)O(nk)
* Average Case: O(nk)O(nk)O(nk)
* Worst Case: O(nk)O(nk)O(nk)

Where:

* nnn is the number of elements in the array.
* kkk is the number of digits in the largest number.

#### **Space Complexity**

* Space: O(n+k)O(n + k)O(n+k), where nnn is the number of elements and kkk is the range of digits.

#### **Steps**

1. Find the maximum number to determine the number of digits.
2. Perform **Counting Sort** on each digit, starting from the least significant digit.
3. Repeat the process for each digit until the entire array is sorted.

#### **Example in Java**

|  |
| --- |
| public class RadixSortExample {  // Function to perform counting sort based on digit place  public static void countingSort(int[] arr, int exp) {  int n = arr.length;  int[] output = new int[n];  int[] count = new int[10];  // Store count of occurrences of (arr[i] / exp) % 10  for (int i = 0; i < n; i++) {  count[(arr[i] / exp) % 10]++;  }  // Change count[i] so that it now contains actual position  for (int i = 1; i < 10; i++) {  count[i] += count[i - 1];  }  // Build the output array  for (int i = n - 1; i >= 0; i--) {  output[count[(arr[i] / exp) % 10] - 1] = arr[i];  count[(arr[i] / exp) % 10]--;  }  // Copy the output array to arr[], so that arr now contains sorted numbers  System.arraycopy(output, 0, arr, 0, n);  }  // Main function to implement radix sort  public static void radixSort(int[] arr) {  // Find the maximum number to determine the number of digits  int max = arr[0];  for (int i = 1; i < arr.length; i++) {  if (arr[i] > max) max = arr[i];  }  // Perform counting sort for every digit.  for (int exp = 1; max / exp > 0; exp \*= 10) {  countingSort(arr, exp);  }  }  public static void main(String[] args) {  int[] arr = {170, 45, 75, 90, 802, 24, 2, 66};  radixSort(arr);  System.out.println("Sorted array: ");  for (int num : arr) System.out.print(num + " ");  }  } |

#### **Execution Steps**

1. Start by sorting based on the least significant digit (LSD), for example, the digit 1 in the number 170.
2. Then, proceed to the next significant digit.
3. Continue this process until all digits are processed.

### ****8. Bucket Sort****

#### **Description**

Bucket Sort is a sorting algorithm that divides the array into several buckets, sorts each bucket individually (using another sorting algorithm like **Insertion Sort**), and then combines the sorted buckets.

It is useful when the input is uniformly distributed over a range.

#### **Time Complexity**

* Best Case: O(n+k)O(n + k)O(n+k) (where nnn is the number of elements and kkk is the number of buckets)
* Average Case: O(n+k)O(n + k)O(n+k)
* Worst Case: O(n2)O(n^2)O(n2) (if all elements are placed in one bucket)

#### **Space Complexity**

* Space: O(n+k)O(n + k)O(n+k)

#### **Steps**

1. Divide the input into kkk equally sized buckets.
2. Sort each bucket using a sorting algorithm (usually **Insertion Sort**).
3. Concatenate the sorted buckets to get the final sorted array.

#### **Example in Java**

|  |
| --- |
| import java.util.\*;  public class BucketSortExample {  // Function to perform bucket sort  public static void bucketSort(float[] arr) {  if (arr.length <= 0) return;  // 1. Create empty buckets  int n = arr.length;  @SuppressWarnings("unchecked")  List<Float>[] buckets = new List[n];  for (int i = 0; i < n; i++) {  buckets[i] = new ArrayList<>();  }  // 2. Distribute input array values into buckets  for (int i = 0; i < n; i++) {  int index = (int) (arr[i] \* n); // Bucket index  buckets[index].add(arr[i]);  }  // 3. Sort each bucket and concatenate  for (int i = 0; i < n; i++) {  Collections.sort(buckets[i]); // Sorting individual buckets  }  // 4. Merge all sorted buckets  int idx = 0;  for (int i = 0; i < n; i++) {  for (float num : buckets[i]) {  arr[idx++] = num;  }  }  }  public static void main(String[] args) {  float[] arr = {0.42f, 0.32f, 0.23f, 0.75f, 0.63f, 0.34f, 0.12f};  bucketSort(arr);  System.out.println("Sorted array: ");  for (float num : arr) System.out.print(num + " ");  }  } |

#### **Execution Steps**

1. Create nnn buckets, where each bucket will hold elements that fall within specific ranges.
2. Distribute the array elements into these buckets.
3. Sort each bucket (usually using **Insertion Sort** for small datasets).
4. Combine the sorted buckets into a single sorted array.

### ****Summary of Sorting Techniques****

* **Merge Sort:** Divide and conquer approach, O(nlog⁡n)O(n \log n)O(nlogn) time complexity.
* **Quick Sort:** Divide and conquer approach, O(nlog⁡n)O(n \log n)O(nlogn) average case, but O(n2)O(n^2)O(n2) worst case.
* **Heap Sort:** Utilizes a binary heap, in-place sorting with O(nlog⁡n)O(n \log n)O(nlogn) time complexity.
* **Radix Sort:** Non-comparative, O(nk)O(nk)O(nk) time complexity, stable sorting.
* **Bucket Sort:** Divides elements into buckets, sorts them individually, O(n+k)O(n + k)O(n+k) time complexity.

These are some of the most common and widely used searching sorting algorithms.