SORTING

ALGORITHMS IN JAVA

Sercan Külcü | Algorithms In Java | 10.05.2023

Contents

Introduction

Sorting algorithms are used to arrange a collection of elements in a specific order, typically in ascending or descending order. There are numerous sorting algorithms available, each with its own advantages and disadvantages in terms of time complexity, space complexity, and stability. Here are some commonly used sorting algorithms:

- Bubble Sort: A simple comparison-based algorithm that repeatedly swaps adjacent elements if they are in the wrong order. It has a time complexity of O(n^2) in the average and worst cases.
- Insertion Sort: This algorithm builds the final sorted array one element at a time by repeatedly inserting an element into the right position within the sorted portion of the array. It also has a time complexity of O(n^2) but performs well for small lists or nearly sorted lists.
- Selection Sort: In this algorithm, the list is divided into two parts: the sorted part and the unsorted part. It repeatedly selects the smallest (or largest) element from the unsorted part and moves it to the sorted part. The time complexity is O(n^2), and it performs a constant number of swaps, making it useful when the cost of swapping elements is high.
- Merge Sort: It is a divide-and-conquer algorithm that divides the input list into two halves, sorts them independently, and then merges them to obtain the sorted output. It has a time complexity of O(n log n) in all cases and is considered a stable sorting algorithm.
- Quick Sort: Another divide-and-conquer algorithm that selects a "pivot" element and partitions the other elements into two sub-arrays, according to whether they are less than or greater than the pivot. It then recursively sorts the sub-arrays. Its average time complexity is O(n log n), but it can degrade to O(n^2) in the worst case.

- Heap Sort: This algorithm uses a binary heap data structure to sort the elements. It first builds a max-heap (or min-heap), then repeatedly extracts the maximum (or minimum) element and rebuilds the heap until the array is sorted. It has a time complexity of O(n log n) and is not considered stable.
- Radix Sort: Radix sort is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping the keys by individual digits. It can achieve linear time complexity O(kn), where n is the number of elements and k is the average number of digits in the keys.

These are just a few examples of sorting algorithms, and there are many more variations and hybrid approaches available. The choice of the sorting algorithm depends on the specific requirements of the problem, the size of the input, and the desired time and space complexity.

Bubble Sort

Bubble sort is a simple sorting algorithm that repeatedly compares adjacent elements and swaps their positions if they are in the wrong order. The algorithm starts at the beginning of the list and compares the first two elements. If the first element is greater than the second element, then the two elements are swapped. The algorithm then moves on to the next two elements and repeats the process. This continues until the end of the list is reached.

The bubble sort algorithm is not very efficient. It has a time complexity of O(n^2), which means that the running time of the algorithm is proportional to the square of the number of elements in the list. However, the bubble sort algorithm is easy to understand and implement.

Here is the implementation of the bubble sort algorithm in Java:

```
public class BubbleSort {


  public static void sort(int[] list) {

    for (int i = 0; i < list.length - 1; i++) {

      for (int j = 0; j < list.length - i - 1; j++) {

        if (list[j] > list[j + 1]) {

          // Swap the elements

          int temp = list[j];

          list[j] = list[j + 1];

          list[j + 1] = temp;

        }
      }
    }
  }

  public static void main(String[] args) {
    // Create a list of integers
    int[] list = {5, 3, 1, 2, 4};

    // Bubble sort the list
    sort(list);

    // Print the sorted list
    System.out.println(Arrays.toString(list));
  }
}
```

This code will print the [1, 2, 3, 4, 5] output.

Insertion Sort

Insertion sort is a simple sorting algorithm that works by repeatedly inserting the next element into the sorted part of the list. The algorithm starts at the beginning of the list and compares the first element to the empty sorted part of the list. If the first element is less than or equal to the last element in the sorted part, then the first element is inserted into the sorted part. The algorithm then moves on to the next element and repeats the process. This continues until the end of the list is reached.

The insertion sort algorithm is not very efficient. It has a time complexity of O(n^2), which means that the running time of the algorithm is proportional to the square of the number of elements in the list. However, the insertion sort algorithm is easy to understand and implement.

Here is the implementation of the insertion sort algorithm in Java:

```
public class InsertionSort {


  public static void sort(int[] list) {

    for (int i = 1; i < list.length; i++) {

      int current = list[i];

      int j = i - 1;

      while (j >= 0 && list[j] > current) {

        list[j + 1] = list[j];

        j--;

      }

      list[j + 1] = current;
    }
  }

  public static void main(String[] args) {
    // Create a list of integers
    int[] list = {5, 3, 1, 2, 4};

    // Insertion sort the list
    sort(list);

    // Print the sorted list
    System.out.println(Arrays.toString(list));
  }
}
```

This code will print the same output as the previous example.

Advantages and Disadvantages

The insertion sort algorithm has the following advantages:

- It is easy to understand and implement.
- It is stable, which means that the relative order of equal elements is preserved.

The insertion sort algorithm has the following disadvantages:

- It is not very efficient.
- It is not a good choice for sorting large lists.

Insertion sort is a simple sorting algorithm that is easy to understand and implement. It is stable, which means that the relative order of equal elements is preserved. However, it is not very efficient and is not a good choice for sorting large lists.

Selection Sort

Selection sort is a simple sorting algorithm that works by repeatedly finding the smallest element in the unsorted part of the list and swapping it with the first element of the unsorted part. The algorithm starts at the beginning of the list and finds the smallest element. The smallest element is then swapped with the first element in the unsorted part. The algorithm then moves on to the next element in the unsorted part and repeats the process. This continues until the end of the list is reached.

The selection sort algorithm is not very efficient. It has a time complexity of O(n^2), which means that the running time of the algorithm is proportional to the square of the number of elements in the list. However, the selection sort algorithm is easy to understand and implement.

Here is the implementation of the selection sort algorithm in Java:

```
public class SelectionSort {


  public static void sort(int[] list) {

    for (int i = 0; i < list.length - 1; i++) {

      int minIndex = i;

      for (int j = i + 1; j < list.length; j++) {

        if (list[j] < list[minIndex]) {

          minIndex = j;

        }

      }

      int temp = list[minIndex];
      list[minIndex] = list[i];
      list[i] = temp;
    }
  }

  public static void main(String[] args) {
    // Create a list of integers
    int[] list = {5, 3, 1, 2, 4};

    // Selection sort the list
    sort(list);

    // Print the sorted list
    System.out.println(Arrays.toString(list));
  }
}
```

This code will print the same output as the previous example.

Advantages and Disadvantages

The selection sort algorithm has the following advantages:

- It is easy to understand and implement.
- It is not as inefficient as bubble sort.

The selection sort algorithm has the following disadvantages:

- It is not a very efficient algorithm.

- It is not a good choice for sorting large lists.

Selection sort is a simple sorting algorithm that is easy to understand and implement. It is not as inefficient as bubble sort, but it is still not a very efficient algorithm. It is not a good choice for sorting large lists.

Merge Sort

Merge sort is a divide-and-conquer algorithm for sorting a list. It works by recursively dividing the list into two halves, sorting each half, and then merging the sorted halves back together.

The merge sort algorithm is efficient. It has a time complexity of O(n log n), which means that the running time of the algorithm is proportional to the logarithm of the square of the number of elements in the list.

The merge sort algorithm works as follows:

- The list is divided into two halves, the left half and the right half.
- The left half is sorted recursively.
- The right half is sorted recursively.
- The sorted left half and the sorted right half are merged back together.

The merge step works as follows:

- Two pointers are initialized, one for the left half and one for the right half.
- The elements pointed to by the two pointers are compared.
- The smaller element is placed in the output list.
- The pointer to the element that was placed in the output list is incremented.
- Steps 2-4 are repeated until one of the pointers reaches the end of its half.
- The remaining elements of the other half are placed in the output list.

The merge sort algorithm is a stable algorithm, which means that the relative order of equal elements is preserved.

Here is the implementation of the merge sort algorithm in Java:

```
public class MergeSort {

  public static void mergeSort(int[] list) {
    if (list.length <= 1) {
      return;
    }

    int mid = list.length / 2;
    int[] left = new int[mid];
    int[] right = new int[list.length - mid];

    for (int i = 0; i < mid; i++) {
      left[i] = list[i];
    }

    for (int i = mid; i < list.length; i++) {
      right[i - mid] = list[i];
    }

    mergeSort(left);
    mergeSort(right);
    merge(left, right, list);
  }

  private static void merge(int[] left, int[] right, int[] list) {
    int i = 0;
    int j = 0;
    int k = 0;

    while (i < left.length && j < right.length) {
      if (left[i] <= right[j]) {
        list[k++] = left[i++];
      } else {
        list[k++] = right[j++];
      }
    }

    while (i < left.length) {
      list[k++] = left[i++];
    }

    while (j < right.length) {
      list[k++] = right[j++];
    }
  }

  public static void main(String[] args) {
    // Create a list of integers
    int[] list = {5, 3, 1, 2, 4};

    // Merge sort the list
    mergeSort(list);

    // Print the sorted list
    System.out.println(Arrays.toString(list));
  }
}
```

This code will print the following output:

```
 [1, 2, 3, 4, 5]
```

Advantages and Disadvantages

The merge sort algorithm has the following advantages:

- It is efficient.
- It is stable.

The merge sort algorithm has the following disadvantages:

- It is not as easy to understand and implement as some other sorting algorithms.

- It requires additional space to store the sorted halves of the list.

Merge sort is a powerful sorting algorithm that is efficient and stable. It is a good choice for sorting large lists.

Quick Sort

Quick sort is a divide-and-conquer algorithm for sorting a list. It works by recursively partitioning the list around a pivot element, and then recursively sorting the two resulting sublists.

The quick sort algorithm is efficient. It has a time complexity of O(n log n), which means that the running time of the algorithm is proportional to the logarithm of the square of the number of elements in the list.

The quick sort algorithm works as follows:

- A pivot element is chosen from the list.
- The list is partitioned around the pivot element, such that all elements smaller than the pivot element are placed before the pivot element, and all elements larger than the pivot element are placed after the pivot element.
- The quick sort algorithm is recursively applied to the two resulting sublists.

The pivot element can be chosen in any way, but it is typically chosen randomly. The choice of pivot element can have a significant impact on the efficiency of the quick sort algorithm.

Here is the implementation of the quick sort algorithm in Java:

```
public class QuickSort {


  public static void quickSort(int[] list) {

    if (list.length <= 1) {

      return;

    }
```

```
int pivot = list[list.length / 2];

int i = 0;
int j = list.length - 1;

while (i < j) {
  while (list[i] < pivot) {
    i++;
  }

  while (list[j] > pivot) {
    j--;
  }

  if (i < j) {
    int temp = list[i];
    list[i] = list[j];
    list[j] = temp;
  }
}

quickSort(list, 0, i - 1);
```

```
  quickSort(list, i + 1, list.length - 1);
}

private static void quickSort(int[] list, int low, int high) {
  if (low < high) {
    int pivot = list[low + (high - low) / 2];

    int i = low;
    int j = high;

    while (i < j) {
      while (list[i] < pivot) {
        i++;
      }

      while (list[j] > pivot) {
        j--;
      }

      if (i < j) {
        int temp = list[i];
        list[i] = list[j];
        list[j] = temp;
      }
    }

    quickSort(list, low, j - 1);
    quickSort(list, i + 1, high);
  }
}

public static void main(String[] args) {
  // Create a list of integers
  int[] list = {5, 3, 1, 2, 4};

  // Quick sort the list
  quickSort(list);

  // Print the sorted list
  System.out.println(Arrays.toString(list));
}
}
```

This code will print the following output:

```
[1, 2, 3, 4, 5]
```

Advantages and Disadvantages

The quick sort algorithm has the following advantages:

- It is efficient.
- It is in-place, which means that it does not require additional space to sort the list.

The quick sort algorithm has the following disadvantages:

- It is not stable, which means that the relative order of equal elements is not preserved.
- It can be inefficient for lists with a small number of elements.

Quick sort is a powerful sorting algorithm that is efficient and in-place. It is a good choice for sorting large lists.

Heap Sort

Heap sort is a comparison-based sorting algorithm. It works by first building a heap data structure from the list to be sorted. A heap is a binary tree in which the value of each node is greater than or equal to the values of its children. Once the heap has been built, the elements are removed from the heap one at a time, and the heap is rebuilt after each removal. The elements are removed in decreasing order, so the heap sort algorithm sorts the list in ascending order.

The heap sort algorithm is efficient. It has a time complexity of O(n log n), which means that the running time of the algorithm is proportional to the logarithm of the square of the number of elements in the list.

The heap sort algorithm works as follows:

- The list is converted into a heap.
- The elements are removed from the heap one at a time, and the heap is rebuilt after each removal.
- The elements are removed in decreasing order, so the heap sort algorithm sorts the list in ascending order.

The heap sort algorithm can be implemented in Java as follows:

```
public class HeapSort {


public static void heapSort(int[] list) {

// Convert the list to a heap

buildHeap(list);

// Remove elements from the heap and rebuild the heap after each removal

for (int i = list.length - 1; i >= 0; i--) {
  // Swap the root element with the last element in the heap
  int temp = list[0];
  list[0] = list[i];
  list[i] = temp;

  // Heapify the root element
  heapify(list, 0);
}
}

private static void buildHeap(int[] list) {
// Start at the last non-leaf node
for (int i = (list.length - 2) / 2; i >= 0; i--) {
// Heapify the subtree rooted at i
heapify(list, i);
}
}

private static void heapify(int[] list, int i) {
// The left and right children of the node at index i
int left = 2 * i + 1;
int right = 2 * i + 2;
```

```
// Find the largest of the node at index i and its children
int largest = i;
if (left < list.length && list[left] > list[largest]) {
  largest = left;
}
if (right < list.length && list[right] > list[largest]) {
  largest = right;
}

// If the largest element is not the node at index i, swap them and heapify the subtree rooted at the largest element
if (largest != i) {
  int temp = list[i];
  list[i] = list[largest];
  list[largest] = temp;
  heapify(list, largest);
}
}

public static void main(String[] args) {
// Create a list of integers
int[] list = {5, 3, 1, 2, 4};
// Heap sort the list
```

```
heapSort(list);


// Print the sorted list
System.out.println(Arrays.toString(list));
}
}
```

This code will print the following output:

```
[1, 2, 3, 4, 5]
```

Advantages and Disadvantages

The heap sort algorithm has the following advantages:

- It is efficient.
- It is in-place, which means that it does not require additional space to sort the list.

The heap sort algorithm has the following disadvantages:

- It is not stable, which means that the relative order of equal elements is not preserved.
- It can be inefficient for lists with a small number of elements.

Radix Sort

Radix sort is a non-comparative sorting algorithm that sorts data by the individual digits of their place values. The algorithm works by repeatedly dividing the data into groups based on the value of the least significant digit, then sorting each group individually. This process is repeated until all digits have been sorted.

Radix sort is an efficient sorting algorithm that has a time complexity of O(kn), where k is the number of digits in the data. Radix sort is also a stable sorting algorithm, which means that the relative order of equal elements is preserved.

The radix sort algorithm works as follows:

- The data is divided into groups based on the value of the least significant digit.
- Each group is sorted individually using a comparison-based sorting algorithm, such as quicksort or mergesort.
- The groups are then merged back together to form the sorted list.

The radix sort algorithm can be implemented in Java as follows:

```
public class RadixSort {


  public static void radixSort(int[] list) {

    // Find the maximum number of digits in the data

    int maxDigits = Integer.toString(list[0]).length();


    // Iterate over each digit

    for (int i = 0; i < maxDigits; i++) {
      // Create a counting sort bucket for each digit value
      int[] buckets = new int[10];

      // Distribute the data into the buckets based on the value of the current digit
      for (int j = 0; j < list.length; j++) {
        buckets[list[j] / (int) Math.pow(10, i)]++;
      }

      // Iterate over the buckets and add the elements back to the list in sorted order
      int j = 0;
      for (int k = 0; k < buckets.length; k++) {
        for (int l = 0; l < buckets[k]; l++) {
          list[j++] = k * (int) Math.pow(10, i);
        }
      }
    }
  }

  public static void main(String[] args) {
    // Create a list of integers
    int[] list = {5, 3, 1, 2, 4};
    // Radix sort the list
    radixSort(list);


    // Print the sorted list
    System.out.println(Arrays.toString(list));
  }
}
```

This code will print the following output:

```
[1, 2, 3, 4, 5]
```

Advantages and Disadvantages

The radix sort algorithm has the following advantages:

- It is efficient.
- It is stable.
- It can be used to sort data of any size.

The radix sort algorithm has the following disadvantages:

- It is not in-place, which means that it requires additional space to sort the data.
- It can be inefficient for data with a small number of digits.

Radix sort is a powerful sorting algorithm that is efficient, stable, and can be used to sort data of any size. It is not in-place, which means that it requires additional space to sort the data. It can also be inefficient for data with a small number of digits.