排序算法说明

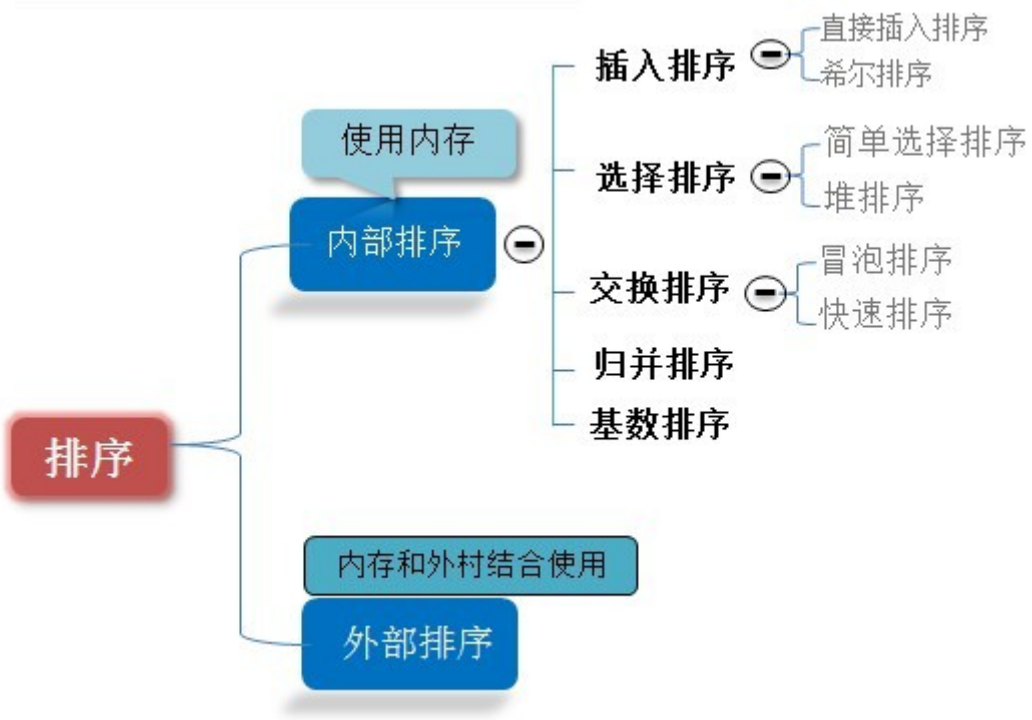
排序的定义

对一序列对象根据某个关键字进行排序。

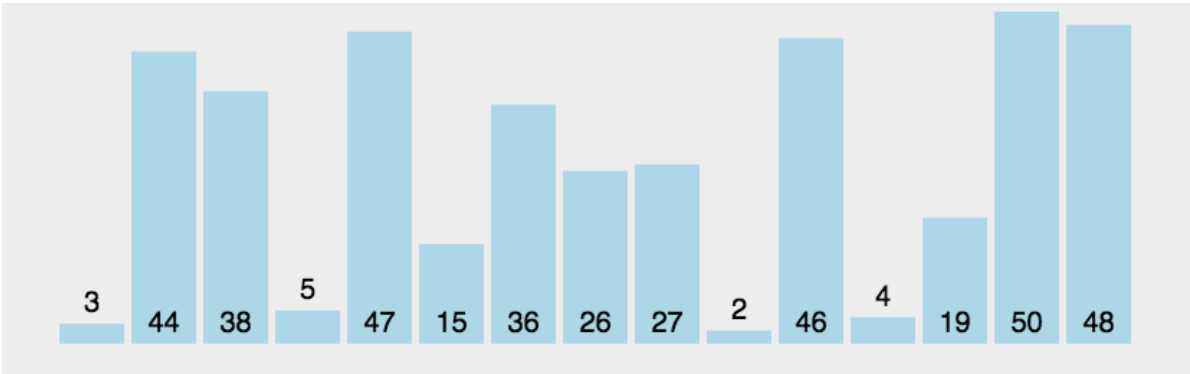
术语说明

- **稳定**：如果a原本在b前面，而a=b，排序之后a仍然在b的前面；
- **不稳定**：如果a原本在b的前面，而a=b，排序之后a可能会出现在b的后面；
- **内排序**：所有排序操作都在内存中完成；
- **外排序**：由于数据太大，因此把数据放在磁盘中，而排序通过磁盘和内存的数据传输才能进行；
- **时间复杂度**： 一个算法执行所耗费的时间。
- **空间复杂度**：运行完一个程序所需内存的大小。

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

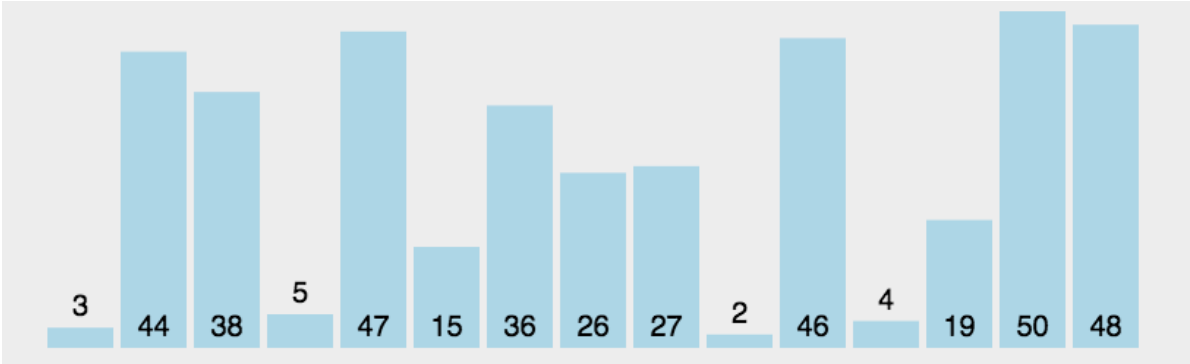


BubbleSort




```
public class BubbleSort {
    public static void main(String[] args) {
        int[] data = new int[]{1,7,4,5,3,2,18,33,24,54,31,22};
        for (int i = 0; i < data.length; i++) {
            for (int j = 0; j < data.length - 1 - i; j++) {
                if (data[j] > data[j + 1]) {
                    int temp = data[j];
                    data[j] = data[j + 1];
                    data[j + 1] = temp;
                }
            }
        }
    }
}
```

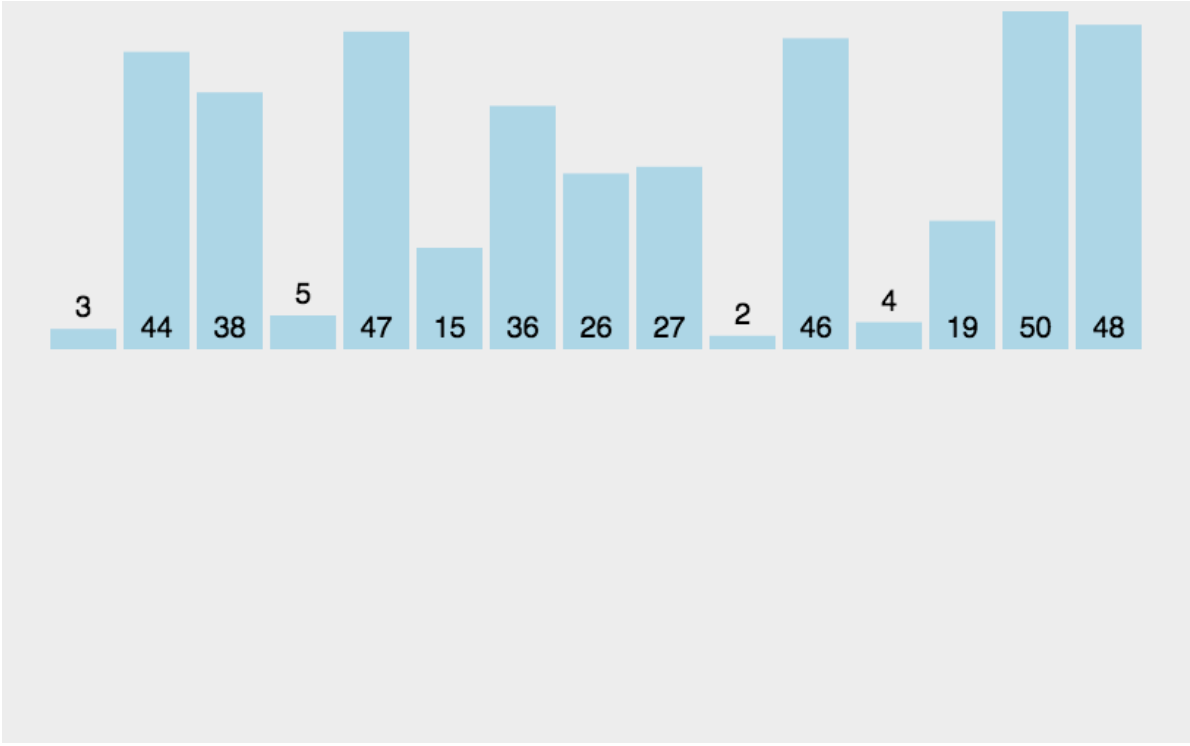
SelectionSort




```
public class SelectionSort {
    public static void main(String[] args) {
        int[] data = new int[]{1,7,4,5,3,2,18,33,24,54,31,22};
        for (int i = 0; i < data.length; i++) {
            int min = data[i];
            int index = i;
            for (int j = i + 1; j < data.length; j++) {
                if (data[j] < min) {
                    index = j;
                }
            }
            data[i] = data[index];
            data[index] = min;
        }

    }
}
```

InsertionSort



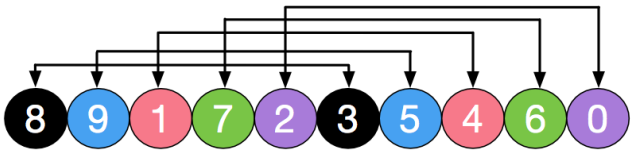

```
public class BubbleSort {
    public static void main(String[] args) {
```

```
        int[] data = new int[]{1,7,4,5,3,2,18,33,24,54,31,22};
        for (int i = 0; i < data.length; i++) {
            int index = i;
            int temp = data[i];
            while (index > 0 && temp < data[index - 1]) {
                data[index] = data[index - 1];
                index--;
            }
            data[index] = temp;
        }
    }
}
```

ShellSort

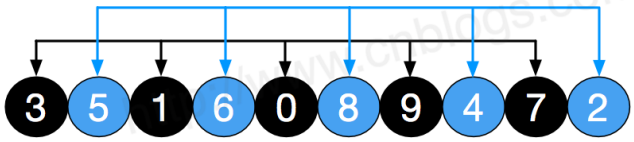
原始数组 以下数据元素颜色相同为一组

初始增量 gap=length/2=5，意味着整个数组被分为5组，[8,3] [9,5] [1,4] [7,6] [2,0]



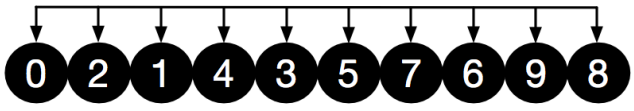
对这5组分别进行直接插入排序，结果如下，可以看到，像3，5，6这些小元素都被调到前面了，

然后缩小增量 gap=5/2=2,数组被分为2组 [3,1,0,9,7] [5,6,8,4,2]



对以上2组再分别进行直接插入排序，结果如下，可以看到，此时整个数组的有序程度更进一步啦。

再缩小增量gap=2/2=1，此时，整个数组为1组[0,2,1,4,3,5,7,6,9,8]，如下



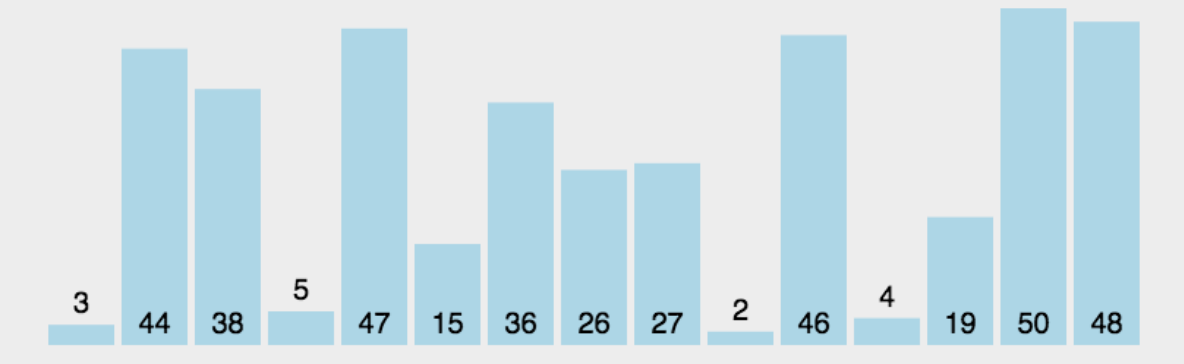
经过上面的“宏观调控”，整个数组的有序化程度成果喜人。

此时，仅仅需要对以上数列简单微调，无需大量移动操作即可完成整个数组的排序。


```
public class ShellSort {
    public static void main(String[] args) {
        int[] a = {83, 23, 2, 1, 5, 3, 54, 9, 0, -1, 22, 4, -9, 34, 51, 33, 22,
11, 6, 8};
```

```
        int gap = a.length / 2;
        while (gap > 0) {
            for (int i = 0; i < gap; i++) {
                for (int j = i; j + gap < a.length; j += gap) { //分段
                    for (int k = i; k + gap < a.length; k += gap) { //把抽取出来的
数进行冒泡排序
                        int temp;
                        if (a[k] > a[k + gap]) {
                            temp = a[k];
                            a[k] = a[k + gap];
                            a[k + gap] = temp;
                        }
                    }
                }
            }
            gap /= 2; //增量减半
        }
        for (int i = 0; i < a.length; i++) {
            System.out.print(" " + a[i]);
        }
    }
}
```

QuickSort




```
public class QuickSort {
    private void quickSortTemp(int[] a, int low, int high) {
        if (low < high) {
            int middle = getMiddle(a, low, high);//找到基准位置
            quickSortTemp(a, low, middle - 1);
            quickSortTemp(a, middle + 1, high);
        } else {
            return;
        }
    }

    private int getMiddle(int[] a, int low, int high) {
        int temp = a[low];
        while (low < high) {
            while (low < high && a[high] > temp) {
                high--;
            }
            a[low] = a[high];
```

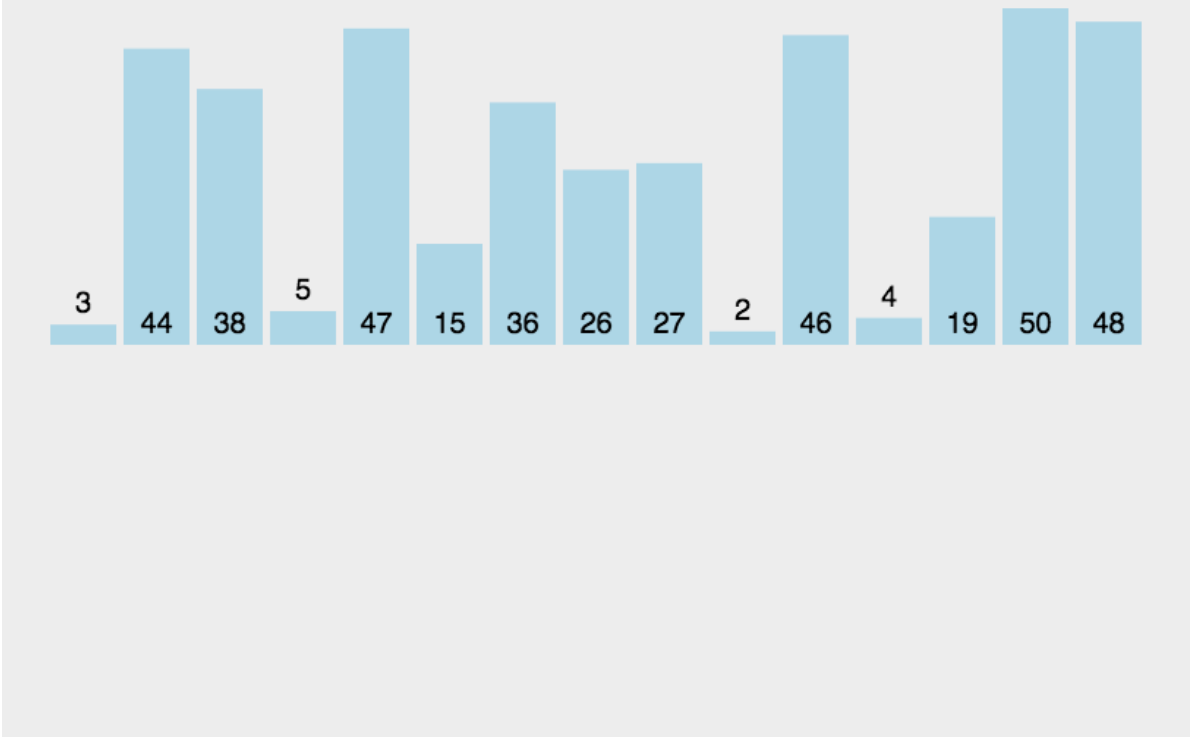
```
            while (low < high && a[low] <= temp) {
                low++;
            }
            a[high] = a[low];
        }
        a[low] = temp; //将基准插回去
        return low; //返回基准所在位置
    }
    public void quickSort(int[] a) {
        quickSortTemp(a, 0, a.length - 1);
    }

    public static void main(String[] args) {
        QuickSort quickSort = new QuickSort();
        int[] array = {19,2,5,7,4,3,44,55,72,54,-1,2,7,44,23,9,88,64,34};
        quickSort.quickSort(array);
        for (int i : array) {
            System.out.print(" " + i);
        }
    }
}
```

MergeSort

算法描述

- 把长度为n的输入序列分成两个长度为n/2的子序列；
- 对这两个子序列分别采用归并排序；
- 将两个排序好的子序列合并成一个最终的排序序列。




```
public class MergeSort {
    public void mergeSort(int[] a, int left, int right) {
        if (left < right) {
```

```
            int middle = (left + right) / 2;
            mergeSort(a, left, middle);
            mergeSort(a, middle + 1, right);
            merge(a, left, middle, right); //排序并合并拆分的2个数组
        }
    }

    private void merge(int[] a, int left, int middle, int right) {
        int rightStart = middle + 1;
        int[] array = new int[a.length];
        int temp = left;
        int third = left;
        //对2个数组进行排序并合并到一个数组中
        while (left <= middle && rightStart <= right) {
            if (a[left] <= a[rightStart]) {
                array[temp++] = a[left++];
            } else {
                array[temp++] = a[rightStart++];
            }
        }
        while (left <= middle) {
            array[temp++] = a[left++];
        }
        while (rightStart <= right) {
            array[temp++] = a[rightStart++];
        }
        //将排序合并好的数组设置到原来的待排序数组中
        while (third <= right) {
            a[third] = array[third++];
        }
    }

    public static void main(String[] args) {
        MergeSort mergeSort = new MergeSort();
        int[] a = new int[]{90, 3, 2, 67, 44, -9, 87, 65, 11, 9, 2, 8};
        mergeSort.mergeSort(a, 0, a.length - 1);
        for (int n : a) {
            System.out.print(" " + n);
        }
    }
}
```