算法复杂度

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **排序算法** | **时间复杂度（平均）** | **时间复杂度（最坏）** | **时间复杂度（最好）** | **空间复杂度** | **稳定性** |
| 冒泡排序 |  |  |  |  |  |
| 选择排序 |  |  |  |  |  |
| 插入排序 |  |  |  |  | 稳定 |
| 快速排序 |  |  |  |  |  |
| 希尔排序 |  |  |  |  |  |
| 归并排序 |  |  |  |  |  |
| 堆排序 |  |  |  |  |  |
| 桶排序 |  |  |  |  |  |
| 计数排序 |  |  |  |  |  |
| 基数排序 |  |  |  |  |  |

说明：

稳定：如果一个原本在b前面，而A = B，排序之后一个仍然在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。

时间复杂度：对排序数据的总的操作次数反映当Ñ变化时，操作次数呈现什么规律。

空间复杂度：是指算法在计算机内执行时所需存储空间的度量，它也是数据规模Ñ的函数。

# 冒泡排序（Bubble Sort）

1. 比较相邻的元素如果第一个比第二个大，就交换它们两个。
2. 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对，这样在最后的元素应该会是最大的数;
3. 针对所有的元素重复以上的步骤，除了最后一个;
4. 重复步骤1〜3，直到排序完成。

代码实现：

package com.synway.dfworks.sort.bubble;

import com.alibaba.druid.support.json.JSONUtils;

*/\*\**

*\* @Title: Bubble.java*

*\* @Description: 冒泡排序*

*\* @author: 夜阑珊*

*\* @date: 2019/12/9 14:41*

*\* @version: v1.0*

*\* @Copyright (c): 2019 synway.com All rights reserved.*

*\*/*

public class Bubble {

*/\*\**

*\* 待排序数组*

*\*/*

private static int[] datas = {6, 4, 5, 3, 8, 1};

public static void main(String[] *args*) {

System.out.println(JSONUtils.toJSONString(datas));

bubbleSort2(datas);

System.out.println(JSONUtils.toJSONString(datas));

}

*/\*\**

*\* 冒泡排序 <br />*

*\* 实现方式一 <br />*

*\**

*\* @param: <br />*

*\* @return: <br />*

*\* @createDate 2019/12/9 15:57 <br />*

*\* @author: 夜阑珊 <br />*

*\* @see [类、类#方法、类#成员]*

*\*/*

private static void bubbleSort(int[] *arrs*) {

for (int i = 0; i < arrs.length; i++) {

for (int j = 0; j < arrs.length - 1 - i; j++) {*//这里-i主要是每遍历一次都把最大的i个数沉到最底下去了，没有必要再替换了*

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

swap(arrs, j, j + 1);

}

}

}

}

*/\*\**

*\* 冒泡排序 <br />*

*\* 实现方式二 <br />*

*\**

*\* @param: <br />*

*\* @return: <br />*

*\* @createDate 2019/12/9 15:57 <br />*

*\* @author: 夜阑珊 <br />*

*\* @see [类、类#方法、类#成员]*

*\*/*

private static void bubbleSort2(int[] *arrs*) {

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

for (int j = arrs.length - 1; j > i; j--) {

if (arrs[j] < arrs[j - 1]) {

swap3(arrs, j - 1, j);

}

}

}

}

*/\*\**

*\* 交换数据方法一*

*\**

*\* @param arrs*

*\* @param i*

*\* @param j*

*\*/*

private static void swap(int[] *arrs*, int *i*, int *j*) {

int temp = arrs[i];

arrs[i] = arrs[j];

arrs[j] = temp;

}

*/\*\**

*\* 交换数据方法二*

*\**

*\* @param arrs*

*\* @param i*

*\* @param j*

*\*/*

private static void swap2(int[] *arrs*, int *i*, int *j*) {

arrs[i] = arrs[i] + arrs[j];

arrs[j] = arrs[i] - arrs[j];

arrs[i] = arrs[i] - arrs[j];

}

*/\*\**

*\* 交换数据方法三*

*\**

*\* @param arrs*

*\* @param i*

*\* @param j*

*\*/*

private static void swap3(int[] *arrs*, int *i*, int *j*) {

arrs[i] = arrs[i] ^ arrs[j];

arrs[j] = arrs[i] ^ arrs[j];

arrs[i] = arrs[i] ^ arrs[j];

}

}

# 选择排序（Selection Sort）

首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后放到已排序序列的末尾。以此类推，直到所有元素均排序完毕。

初始状态：无序区为[R [1..N]，有序区为空;

第我趟排序（I = 1,2,3，... N-1）开始时，当前有序区和无序区分别为[R [1..i-1]和R（i..n）。该趟排序从当前无序区中 - 选择关键字最小的记录R [k]，将它与无序区的第1个记录R交换，使R [1..i]和R [i + 1 .. n）的分别变为记录个数增加1个的新有序区和记录个数减少1个的新无序区;

n-1个趟结束，数组有序化了。

代码实现：

package com.synway.dfworks.sort.selection;

import com.alibaba.druid.support.json.JSONUtils;

import com.synway.dfworks.sort.Swap;

*/\*\**

*\* @Title: Selection.java*

*\* @Description: 选择排序*

*\* @author: 夜阑珊*

*\* @date: 2019/12/9 17:05*

*\* @version: v1.0*

*\* @Copyright (c): 2019 synway.com All rights reserved.*

*\*/*

public class Selection {

private static int[] datas = {6, 4, 5, 3, 8, 1};

public static void main(String[] *args*) {

System.out.println(JSONUtils.toJSONString(datas));

selectionSort(datas);

System.out.println(JSONUtils.toJSONString(datas));

}

private static void selectionSort(int[] *arrs*) {

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

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

if (arrs[i] > arrs[j]) {

Swap.swap(arrs, i, j);

}

}

}

}

}

# 插入排序（Insertion Sort）

插入排序（插入分页）的算法描述是一种简单直观的排序算法。它的工作原理是通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。

算法描述

一般来说，插入排序都采用就地在数组上实现具体算法描述如下：

1. 从第一个元素开始，该元素可以认为已经被排序;
2. 取出下一个元素，在已经排序的元素序列中从后向前扫描;
3. 如果该元素（已排序）大于新元素，将该元素移到下一位置;
4. 重复步骤3，直到找到已排序的元素小于或者等于新元素的位置;
5. 将新元素插入到该位置后;
6. 重复步骤2〜5。

代码实现：

package com.synway.dfworks.sort.insertion;

import com.alibaba.druid.support.json.JSONUtils;

import com.synway.dfworks.sort.Swap;

*/\*\**

*\* @Title: Insertion.java*

*\* @Description: TODO (用一句话描述该文件做什么)*

*\* @author: 夜阑珊*

*\* @date: 2019/12/9 17:59*

*\* @version: v1.0*

*\* @Copyright (c): 2019 synway.com All rights reserved.*

*\*/*

public class Insertion {

private static int[] datas = {6, 4, 5, 3, 8, 1};

public static void main(String[] *args*) {

System.out.println(JSONUtils.toJSONString(datas));

insertionSort(datas);

System.out.println(JSONUtils.toJSONString(datas));

}

private static void insertionSort(int[] *arrs*) {

for (int i = 0; i < arrs.length; i++) {

int preIndex = i - 1;

int currentNode = arrs[i];

while (preIndex >= 0 && arrs[preIndex] > currentNode) {

arrs[preIndex + 1] = arrs[preIndex];

preIndex--;

}

arrs[preIndex + 1] = currentNode;

}

}

}

# 快速排序（Quick Sort）

快速排序的基本思想：通过一趟排序将待排记录分隔成独立的两部分，其中一部分记录的关键字均比另一部分的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序。

算法描述

快速排序使用分治法来把一个串（名单）分为两个子串（子列表）具体算法描述如下：

1. 从数列中挑出一个元素，称为“基准”（pivot）;
2. 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置这个称为分区（分区）操作。
3. 递归地（递归）把小于基准值元素的子数列和大于基准值元素的子数列排序。

代码实现：

package com.synway.dfworks.sort.quick;

import com.alibaba.druid.support.json.JSONUtils;

*/\*\**

*\* @Title: Quick.java*

*\* @Description: 快速排序算法一*

*\* @author: 夜阑珊*

*\* @date: 2019/12/10 9:35*

*\* @version: v1.0*

*\* @Copyright (c): 2019 synway.com All rights reserved.*

*\*/*

public class Quick {

private static int[] datas = {2, 6, 4, 5, 3, 8, 1};

public static void main(String[] *args*) {

System.out.println(JSONUtils.toJSONString(datas));

quickSort(datas, 0, datas.length - 1);

System.out.println(JSONUtils.toJSONString(datas));

}

private static void quickSort(int[] *arrs*, int *left*, int *right*) {

if (left < right) {

int middle = getMiddle(arrs, left, right);

quickSort(arrs, 0, middle - 1);

quickSort(arrs, middle + 1, right);

}

}

*/\*\**

*\* 进行一次具体的快速排序算法*

*\**

*\* @param arrs*

*\* @param left*

*\* @param right*

*\*/*

private static int getMiddle(int[] *arrs*, int *left*, int *right*) {

int temp = arrs[left];*//基准元素*

while (left < right) {

*//从右边开始找第一个比基准元素小的元素位置*

while (left < right && arrs[right] >= temp) {

right--;

}

arrs[left] = arrs[right];

*//从左边开始找第一个比基准元素大的元素*

while (left < right && arrs[left] <= temp) {

left++;

}

arrs[right] = arrs[left];

}

arrs[left] = temp;

return left;

}

}

package com.synway.dfworks.sort.quick;

import com.alibaba.druid.support.json.JSONUtils;

import com.synway.dfworks.sort.Swap;

*/\*\**

*\* @Title: Quick2.java*

*\* @Description: 快速排序算法二*

*\* @author: 夜阑珊*

*\* @date: 2019/12/11 9:34*

*\* @version: v1.0*

*\* @Copyright (c): 2019 synway.com All rights reserved.*

*\*/*

public class Quick2 {

private static int[] datas = {2, 6, 4, 5, 3, 8, 1};

public static void main(String[] *args*) {

System.out.println(JSONUtils.toJSONString(datas));

quickSort(datas, 0, datas.length - 1);

System.out.println(JSONUtils.toJSONString(datas));

}

public static void quickSort(int[] *arrs*, int *left*, int *right*) {

if (left >= right) {

return;

}

int index = partition(arrs, left, right);

quickSort(arrs, left, index - 1);

quickSort(arrs, index + 1, right);

}

public static int partition(int[] *arrs*, int *left*, int *right*) {

int keyValue = arrs[left];

int key = left;

while (left < right) {

*//从右边开始找第一个比基准数小的元素*

while (left < right && arrs[right] >= keyValue) {

right--;

}

*//从左边开始找第一个比基准数大的元素*

while (left < right && arrs[left] <= keyValue) {

left++;

}

*//交换这2个元素*

Swap.swap(arrs, left, right);

}

*//将基准元素放在*

Swap.swap(arrs, key, left);

return left;

}

}

# 希尔排序（Shell Sort）

*/\*\**

*\* @Title: Shell.java*

*\* @Description: 希尔排序*

*\* @author: 夜阑珊*

*\* @date: 2019/12/9 18:35*

*\* @version: v1.0*

*\* @Copyright (c): 2019 synway.com All rights reserved.*

*\*/*

public class Shell {

private static int[] datas = {6, 4, 5, 3, 8, 1, 2, 7};

public static void main(String[] *args*) {

System.out.println(JSONUtils.toJSONString(datas));

shellSort(datas);

System.out.println(JSONUtils.toJSONString(datas));

}

private static void shellSort(int[] *arrs*) {

*//首先根据数组长度确定增量的最大值*

int len = arrs.length;

int gap = 1;

*// 动态定义间隔序列，按gap \* 3 + 1得到增量序列的最大值*

while (gap < len / 3) {

gap = gap \* 3 + 1;

}

*//进行增量查找和排序*

while (gap > 0) {

for (int i = gap; i < len; i++) {

for (int k = i; k < len; k = k + gap) {

*//判断是否需要重新排序，如果小于k-gap处的值，需要重新排序*

if (arrs[k] < arrs[k - gap]) {

int temp = arrs[k];

int j = k;

for (; j >= i && temp < arrs[j - gap]; j -= gap) {

arrs[j] = arrs[j - gap];

System.out.println(ArrayUtils.toString(arrs));

}

arrs[j] = temp;

}

}

}

gap = (gap - 1) / 3;

}

}

}

# 归并排序（Merge Sort）

# 堆排序（堆排序）

# 桶排序（Bucket Sort）

# 计数排序（Counting Sort）

# 基数排序（Radix Sort）

D