# 排序

## 分类

### 内排序/外排序

所谓的内排序是指所有的数据已经读入内存，在内存中进行排序的算法。排序过程中不需要对磁盘进行读写。同时，内排序也一般假定所有用到的辅助空间也可以直接存在于内存中。与之对应地，另一类排序称作外排序，即内存中无法保存全部数据，需要进行磁盘访问，每次读入部分数据到内存进行排序。

下面给出它们的具体分类：

内部排序算法：交换排序（冒泡排序、快速排序），选择排序（直接选择排序、堆排序），插入排序（直接插入排序、希尔排序），归并排序。

外部排序算法：计数排序、基数排序、桶排序等。

### 线性时间比较类排序/线性时间非比较类排序

非线性时间比较类排序：通过比较来决定元素间的相对次序，由于其时间复杂度不能突破O(nlogn)，因此称为非线性时间比较类排序。

线性时间非比较类排序：不通过比较来决定元素间的相对次序，它可以突破基于比较排序的时间下界，以线性时间运行，因此称为线性时间非比较类排序。

## 性能

### 稳定性

稳定性：如果i=j，排序前i在j的前面，排序后i仍然在j的前面，即相等的两个数字的相对位置在排序前后不变，则该算法是稳定的，否则不稳定。

排序算法如果是稳定的，那么从一个键上排序，然后再从另一个键上排序，前一个键排序的结果可以为后一个键排序所用。可能比较难理解，这里再举个例子方便理解，比如在基数排序中，先将低位排序，再逐次按高位排序，稳定的话就可以保证排序后低位元素的顺序在高位相同时是不会改变的。

### 时间复杂度

时间复杂度：指执行算法所需要的工作量，即对待排序数据的总操作次数，我们用它来描述算法的运行时间。

### 空间复杂度

空间复杂度：指执行算法所需的内存空间。

## 数据结构

在排序算法中，常用顺序表数据结构表示待排序的数据，其定义如下：

#define MAXSIZE 100 /\*用于要排序数组个数最大值，可根据需要修改\*/

typedef struct

{

int data[MAXSIZE+1]; /\*存储要排序数组，data[0]用作哨兵或临时变量\*/

int length; /\*用于记录顺序表的长度\*/

}SqList;

另外，排序操作中经常用到数组元素的交换，将这些操作封装为函数，如下所示：

/\*交换L中数组data的下标为i和j的值\*/

void swap(SqList \*L, int i, int j)

{

int temp = L->data[i];

L->data[i] = L->data[j];

L->data[j] = temp;

}

# 冒泡排序

## 基本思想

冒泡排序（Bubble Sort）是一种交换排序，基本思想是：两两比较相邻记录的关键字，如果反序则交换，直到没有反序的记录为止。因为每遍历一次列表，最大（或最小）的元素会经过交换一点点“浮”到列表的一端（顶端），所以形象的称这个算法为冒泡算法。

## 具体步骤

1、比较两个相邻元素，如果前一个比后一个大，则交换这两个相邻元素；

2、从头至尾对每一对相邻元素进行步骤1的操作，完成1次对整个待排序数字列表的遍历后，最大的元素就放在了该列表的最后一个位置上了；

3、对除最后一个元素的所有元素重复上述步骤，这第二次遍历后第二大的元素就也放在了正确的位置（整个列表的倒数第二位置上）；

4、不断重复上述步骤，每次遍历都会将一个元素放在正确的位置上，从而下次遍历的元素也会随之减少一个，直至没有任何一对数字需要比较。

## 实现动图

## 代码实现

### C语言

/\*对顺序表L作交换排序（冒泡排序初级版）\*/

void BubbleSort0(SqList \*L)

{

int i,j;

for(i=1;i<L->length;i++)

{

for(j=i+1;j<=L->length;j++)

{

if(L->data[i]>L-data[j])

{

swap(L.i.j); /\*交换L->data[i]和L->data[j]的值\*/

}

}

}

}

这段代码严格讲，不算是标准的冒泡排序算法，因为它不满足“两两比较相邻记录”的冒泡排序思想，应该算是最简单的交换排序算法。它的思想是让每一个关键字，都和它后面的每一个关键字进行比较，如果大则交换，这样第一位置的关键字在一次循环后就变成最小值。但是这种方法效率比较低下，采用如下的改进算法：

/\*对顺序表L作冒泡排序\*/

void BubbleSort(SqList \*L)

{

int i,j;

for(i=1;i<L->length;i++)

{

for(j=L->length;j>=i;j--) /\*注意j是从后往前循环\*/

{

if(L->data[j]>L->data[j+1])

/\*若前者大于后者（注意这里与上一算法差异）\*/

{

swap(L,j,j+1);

}

}

}

}

对上述算法还可以做进一步的优化：

/\*对顺序表L作冒泡排序\*/

void BubbleSort(SqList \*L)

{

int i,j;

bool flag = true; /\*flag用作标记\*/

for(i=1;i<L->length && flag;i++) /\*若flag为true则退出循环\*/

{

flag = false; /\*初始化为false\*/

for(j=L->length;j>=i;j--) /\*注意j是从后往前循环\*/

{

if(L->data[j]>L->data[j+1])

{

swap(L,j,j+1);

flag = true; /\*如果有数据交换，则flag为true\*/

}

}

}

}

### Python

## 性能分析

**稳定性：**稳定

**时间复杂度：**

平均情况：O(n^2)

最好情况：O(n)

最坏情况：O(n^2)

**空间复杂度：**O(1)

# 快速排序

## 基本思想

通过一趟排序将待排序列表分割成独立的两部分，其中一部分的所有元素都比另一部分小，然后再按此方法将独立的两部分分别继续重复进行此操作，这个过程我们可以通过递归实现，从而达到最终将整个列表排序的目的。

## 具体步骤

1、从待排序列表（数组）中选择一个元素作为基准（pivot），这里我们选择最后一个元素元素；

2、遍历列表，将所有小于基准的元素放在其前面，这样就可以将待排序列表分成两部分了；

3、递归地对每个部分进行1、2操作，这里递归结束的条件是序列的大小为0或1，此时递归结束，排序就已经完成了。

## 代码实现

### C语言

### Python

## 性能分析

**稳定性：**不稳定

**时间复杂度：**

平均情况：O(nlogn)

最好情况：O(nlong)

最坏情况：O(n^2)

其实不难理解，快排的最坏情况就已经退化为冒泡排序了！

**空间复杂度：**

平均情况：O(logn)

最好情况：O(logn)

最坏情况：O(n)

# 简单选择排序

## 基本思想

先在待排序列表中找到最小（大）的元素，把它放在起始位置作为已排序序列；然后，再从剩余待排序序列中找到最小（大）的元素放在已排序序列的末尾，以此类推，直至完毕。

这个有点像暴力解决的意思，所以它的效率不高（最坏、最好、平均都一样很差）。

## 具体步骤

1、初始状态整个待排序序列为无序序列，有序序列为空；

2、每次遍历无序序列将最小元素交换到有序序列之后；

3、n-1趟遍历后排序完成。

## 代码实现

### C语言

function selectionSort(arr) {

var len = arr.length;

var minIndex, temp;

for (var i = 0; i < len - 1; i++) {

minIndex = i;

for (var j = i + 1; j < len; j++) {

if (arr[j] < arr[minIndex]) { // 寻找最小的数

minIndex = j; // 将最小数的索引保存

}

}

temp = arr[i];

arr[i] = arr[minIndex];

arr[minIndex] = temp;

}

return arr;

}

### Python

## 性能分析

**稳定性：**不稳定

**时间复杂度：**

平均情况：O(n^2)

最好情况：O(n^2)

最坏情况：O(n^2)

**空间复杂度：**O(1)

# 堆排序

## 基本思想

堆的结构相当于一个完全二叉树，最大堆满足下面的性质：父结点的值总大于它的孩子结点的值。

## 具体步骤

1、将待排序列表构造成一个最大堆，作为初始无序堆（即初始无序列表）；

2、将堆顶元素（最大值）与堆尾元素互换；

3、将该堆（无序区）尺寸缩小1，并对缩小后的堆重新调整为最大堆形式；

4、重复上述步骤，直至堆（无序区）的尺寸变为1，此时排序完成。

## 代码实现

### C语言

### Python

## 性能分析

**稳定性：**不稳定

**时间复杂度：**

平均情况：O(nlogn)

最好情况：O(nlogn)

最坏情况：O(nlogn)

堆排序效率很好，所以大家一定要会熟练将堆排序应用到各种场景中～

**空间复杂度：**O(1)

# 直接插入排序

## 基本思想

对于未排序元素，在已排序序列中从后向前扫描，找到相应位置把它插入进去；在从后向前扫描过程中，需要反复把已排序元素逐步向后挪位，为新元素提供插入空间。

## 具体步骤

1、从第一个元素开始，默认该元素已被排好序；

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

## 性能分析

**稳定性：**稳定

**时间复杂度：**

平均情况：O(n^2)

最好情况：O(n)

最坏情况：O(n^2)

**空间复杂度：**O(1)

# 希尔排序

# 归并排序

# 计数排序

## 基本思想

对于给定的输入序列中的每一个元素x，确定该序列中值小于x的元素的个数（此处并非比较各元素的大小，而是通过对元素值的计数和计数值的累加来确定）。一旦有了这个信息，就可以将x直接存放到最终的输出序列的正确位置上。

## 具体步骤

## 代码实现

## 性能分析

# 桶排序

## 基本思想

## 具体步骤

## 代码实现

## 性能分析

**稳定性：**稳定

**时间复杂度：**

平均情况：O(m+n)

最好情况：

最坏情况：

**空间复杂度：**O(n)

# 基数排序

基数排序（radix sort）是一种非线性时间的排序算法。