# 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。

冒泡排序的原理是：每一趟只能确定将一个数归位。即如果有n个数进行排序，只需将n-1个数归位，也就是说要进行n-1趟操作。

## 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

#### 基本版

**版本1：**

/\*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++)

//这里是data[i]与data[i+1]->data[L->length]都比较，不是相邻的两两比较

{

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）\*/

/\*算法核心部分是双重嵌套循环，因此时间复杂度是O(N2)\*/

{

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\*/

}

}

}

}

#### 改进版

**版本2：**

#include <iostream>

#include <vector>

using namespace std;

//冒泡排序 从大到小排序

void Bulldle(int\* array,int num)

{

int i,j,temp;

for(i=0;i<num-1;i++)

{

for(j = i+1;j<num;j++)

if(array[i] < array[j])

{

temp = array[i];

array[i] = array[j];

array[j] = temp;

}

}

}

// 冒泡排序的升级版 从小到大排序

void Bulldle2(int\* array,int num)

{

int i,j,temp;

int flag =1;

for(i=0;i<num-1 && flag;i++)

{

flag =0;

for(j=i+1;j<num;j++)

if(array[i] > array[j])

{

temp = array[i];

array[i] = array[j];

array[j] = temp;

flag =1;//增加标志位，即“哨兵”

}

}

}

int main()

{

int array[]={7,5,2,9,1,3};

Bulldle2(array,6);

int i;

for(i=0;i<sizeof(array)/sizeof(int);i++)

cout<<array[i]<<endl;

return 0;

}

### 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)

冒泡排序解决了桶排序空间复杂度高的问题，但是实际应用的很少，因为时间复杂度很高，不实用！因此引入快速排序。

# 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。

快速排序其实就是把冒泡排序中的相邻两数比较转换为与某一基准值的比较（可以理解为冒泡排序升级版）。

快速排序用到的算法思想是：递归与分治策略（二分思想）。

## 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

**版本1：**

**版本2：**

#include <stdio.h>

#include <stdlib.h>

//将一个数组以某一个基准点划分为两个子数组

int Partition(int\* array,int left,int right)

{

int key = array[right];//以最后一个元素为基准点

int i = left-1;

int temp;

//开始以基准点为标准分割序列

for(int j = left;j<right;j++)

{

if(array[j] < key)

{

i++;

temp = array[j];

array[j] = array[i];

array[i] = temp;

}

}

//将基准点放置到合适的位置上

temp = array[i+1];

array[i+1] = array[right];

array[right] = temp;

return i+1;

}

//快速排序

void quicksort(int\* array,int left,int right)

{

if(left < right)

{

int k = Partition(array,left,right);

quicksort(array,left,k-1);

quicksort(array,k+1,right);

}

}

//第二种分割的方法

void swap(int\* a,int\* b)

{

int temp = \*a;

\*a = \*b;

\*b = temp;

}

int partition2(int\* array,int low,int high)

{

int key = array[low];

while(low < high)

{

//从后面找到一个合适的值和前面的交换

while(low<high && array[high] >= key)

high--;

swap(&array[low],&array[high]);

//从前面找到一个合适的值和后面的交换

while(low< high && array[low] <= key)

++low;

swap(&array[low],&array[high]);

}

return low;

}

//第二种分割方法下的快速排序

void quicksort2(int\* array,int left,int right)

{

int i,j,key;

if(left < right)

{

i = left;

j = right;

key = array[i];//以最左边的元素作为划分的基准点

do

{

while(array[j] > key && i<j)

j--;//从右向左找第一个小于基准值的位置j

if(i<j) //找到了位置j

{

array[i] = array[j];

i++;

} //将第J个元素置于左端并重置i

while(array[i] < key && i<j)

i++;//从左向右找第一个大于标准的位置i

if(i<j)

{

array[j] = array[i];

j--;

} //将第I个元素置于右端并重置j

} while(i!= j);

array[i] = key;// 分割完成

quicksort2(array,left,i-1);

quicksort2(array,i+1,right);

}

}

int main()

{

int array[]={9,4,5,2,1,3,7,8,6};

quicksort2(array,0,8);

int i;

for(i=0;i<9;i++)

printf("%d\n",array[i]);

return 0;

}

### 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)

快速排序之所以比较快，是因为相比冒泡排序，每次交换都是跳跃式的。这样在每次交换的时候就不会像冒泡排序一样只能在相邻的数之间进行交换，交换的距离就大得多了。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）。但是，其比较和元素移动次数要比冒泡排序少，效率略高。

## 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

#include<stdio.h>

void SelectSort(int k[], int n)

{

int i,j,min,temp;

for(i=0;i<n-1;i++)

{

min = i;

for( j=i+1; j<n;j++)

{

min = i;

for(j=i+1;j<n;j++)

{

if(k[j] < k[min])

{

min = j;

}

}

if( min != i)

{

temp = k[min];

k[min] = k[i];

k[i] = temp;

}

}

}

}

### Python

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;

}

## 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)

注：比较和移动次数要比冒泡排序少。

**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。

注：如果构造从小到大的排序，使用大顶堆。

## 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

#include <stdio.h>

#include <stdlib.h>

//调整堆

void HeapAdjust(int\* array,int s,int length)

{

int temp = array[s];//待调整的元素

int child = 2\*s+1;//待调整节点的左孩子的位置

//开始调整

while(child < length)

{

//如果右孩子大于左孩子，找到比当前待调整节点大的孩子节点

if(child+1 < length&& array[child] < array[child+1])

child++;

//如果较大的孩子大于待调整的节点

if(array[s] < array[child])

{

array[s] = array[child];//那么较大的节点向上移

s = child;//更新待调整节点的位置

child = 2\*s+1;//更新待调整节点的左孩子

}

else

break;

array[s] = temp;

}

}

//堆排序

void HeapSort(int\* array,int length)

{

int i;

//构造最大堆

for(i=(length-1)/2;i>=0;i--)

{

HeapAdjust(array,i,length);

}

//从最后一个元素开始对序列进行调整

for(i=length-1;i>0;i--)

{

//交换堆顶元素和堆中最后一个元素

int tmp = array[i];

array[i] = array[0];

array[0] = tmp;

//每次交换元素之后，就需要重新调整堆

HeapAdjust(array,0,i);

}

}

int main()

{

int array[]={3,4,5,1,9,8,6,2,7,10};

HeapSort(array,10);

int i;

for(i=0;i<10;i++)

printf("%d\n",array[i]);

return 0;

}

或：

void swap(int k[], int i, int j)

{

int temp;

temp = k[i];

k[i] = k[j];

k[j] = temp;

}

void HeapAdjust(int k[], int s, int n)

{

int i,temp;

temp = k[s];

for(i=2\*s;i<=n;i\*=2)

{

if(i<n && k[i]<k[i+1])

{

i++;

}

if(temp>=k[i])

{

break;

}

k[s] = k[i];

s= i;

}

k[s] = temp;

}

void HeapSort(int k[], int n)

{

int i;

for(i=n/2;i>0;i--)

{

HeapAdjust(k,i,n);

}

for(i=n;i>1;i--)

{

swap(k,1,i);

HeapAdjust(k,1,i-1);

}

}

### 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

### C语言

**版本1：**

void InsertSort(int k[], int n)

{

int i,j,temp;

for(i=1;i<n;i++)

{

if(k[i]<k[i-1])

{

temp = k[i];

for(j=i-1;k[j]>temp;j--)

{

k[j+1] = k[j];

}

k[j+1] = temp;

}

}

}

**版本2：**

#include <iostream>

using namespace std;

//插入排序 （思想：在一个已经排序好的序列中，为下一个元素找合适的插入位置）

void InsertSort(int\* array,int n)

{

int i,j;

int temp;

//从第二个元素开始进行插入排序 为其在前面排序好的序列中找合适的位置

for(i=1;i<n;i++)

{

//为当前元素选择合适的位置

temp = array[i];

for(j=i-1;j>=0;j--)

if(array[j] > temp)

array[j+1] = array[j];

else

break;

//找到合适的位置后填充当前值

array[j+1] = temp;

}

for(i=0;i<n;i++)

cout<<array[i]<<endl;

}

int main()

{

int array[]={7,5,2,9,1,3};

InsertSort(array,6);

return 0;

}

### 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)

# 希尔排序

## 基本思想

## 具体步骤

## 代码实现

### C语言

void ShellSort(int k[], int n)

{

int i,j,temp;

int gap = n;

do

{

gap = gap/3 + 1;

for(i=gap;i<n;i++)

{

if(k[i]<k[i-gap])

{

temp = k[i];

for(j=i-gap;k[j]>temp;j-=gap)

{

k[j+gap] = k[j];

}

k[j+gap] = temp;

}

}

}while(gap>1)

}

### Python

## 性能分析

# 归并排序

## 基本思想

## 具体步骤

1. 分治
2. 合并

## 代码实现

### C语言

**版本1：**

#include <stdio.h>

#include <stdlib.h>

//合并两个有序子数组

void Mergearray(int\* a,int left,int mid,int right,int\* temp)

{

int i = left,j = mid+1;

int m = mid,n = right;

int k=0;

//二路归并

while(i<=m && j<=n)

{

if(a[i]<=a[j])

temp[k++] = a[i++];

else

temp[k++] = a[j++];

}

//处理子数组中剩余的元素

while(i<=m)

temp[k++] = a[i++];

while(j<=n)

temp[k++] = a[j++];

//从临时数组中拷贝到目标数组

for(i=0;i<k;i++)

a[left+i] = temp[i];

}

//归并排序的核心工作

int MergeSort(int\* a,int left,int right,int\* temp)

{

if(left < right)

{

int mid = left + (right-left)/2;

//归并排序 使得左边序列有序

MergeSort(a,left,mid,temp);

//归并排序 使得右边序列有序

MergeSort(a,mid+1,right,temp);

//合并两个有序序列

Mergearray(a,left,mid,right,temp);

}

}

int main()

{

int array[]={6,3,2,1,4,10,9,8,7,5};

int b[10];

MergeSort(array,0,9,b);

int i;

for(i=0;i<10;i++)

printf("%d\n",array[i]);

return 0;

}

**版本2：**

原地归并排序

#include <iostream>

#include <vector>

using namespace std;

/\*

题目要求：

两个有序子序列a[0,mid-1]和a[mid,num-1]， 将两个子数组归并，使其整体有序空间复杂度为O(1)

\*/

/\*

思路：将同一个数组中两段有序序列合并为一个 可以使用插入的排序方式

将后半部分的元素插入到前半部分内 对后半部分的每一个元素 都在前半部分找到合适的位置然后插入（类似于插入排序）

\*/

void Merge2(int\* array,int begin,int end)

{

int middle = begin + (end-begin)/2;

int i= begin;

int temp;

//将后半部分中的元素依序插入前面有序子序列中

while(middle <= end)

{

temp = array[middle];//待插入的元素

if(array[i] < array[middle]) //找到合适的位置

{

i++;

}

else //进行插入的操作

{

int index = middle;

while(index != i)

{

array[index] = array[index-1];

index--;

}

array[i++] = temp;

middle++;

}

}

} //这个函数能否再精简？

/\*

原地归并的思想

假如有一个数组，里面有两段已经排序好的数组不使用额外的空间将这两段有序序列进行合并有两段有序空间分别为A B，从A中找第一个大于B[0]的元素并记录为A[i] 从B中找第一个大于A[i]的元素并记录为B[j] 在这段空间中进行右移 然后就变得有序

\*/

//对称交换

void Revere(int\* array,int begin,int end)

{

int temp;

while(begin < end)

{

temp = array[begin];

array[begin]=array[end];

array[end] = temp;

begin++;

end--;

}

}

//右旋转

/\*

array为待旋转的数组

begin 为开始位置

middle是旋转中心

end 为后半段

\*/

void Rotate\_right(int\* array,int begin,int middle,int end)

{

Revere(array,begin,middle);

Revere(array,middle+1,end);

Revere(array,begin,end);

}

//原地合并

void Merge\_second(int\* array,int begin,int end)

{

//[begin,mid-1] [mid,end]分别有序

int mid = begin + (end-begin)/2+1;

int i = begin;

int index;

//直到将两个有序子序列其中的元素排序完成 那么两个有序子序列就合

并完成

while(mid <= end && i<mid)

{

//从前面的有序序列中找到一个元素，其大于后序有序序列中的第一

个元素

while(array[i] <= array[mid]&& i < mid)

{

i++;

}

//记录当前两个有序子序列的分界点

index = mid;

//从后面有序子序列中找到一个元素，其大于前面有序序列的第i个

元素

while(mid <= end && array[mid]<= array[i])

{

mid++;

}

//以index为分界点，也就是以当前两个有序子序列的分界点为旋转

中心,右旋转

Rotate\_right(array,i,index-1,mid-1);

//更新前面子序列中未排序元素的位置

i += (mid-index);

}

}

//归并排序总体函数

void Inplace\_MergeSort(int\* arr,int beg,int end)

{

if(beg < end)

{

int mid = (beg+end)/2;

Inplace\_MergeSort(arr,beg,mid);

Inplace\_MergeSort(arr,mid+1,end);

Merge\_second(arr,beg,end);

}

}

int main()

{

int array[]= {2,4,6,8,10,1,3,5,7,9};

int i;

Inplace\_MergeSort(array,0,9);

for(i=0;i<10;i++)

cout<<array[i]<<endl;

return 0;

}

### Python

## 性能分析

## 适用场景

# 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

桶排序其实就是拿空间换时间。

## 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。