排序算法

- 三月 9, 2013
- 排序算法已关闭评论

目录

[显示]

1. 摘要

排序算法是计算机技术中最基本的算法，许多复杂算法都会用到排序。尽管各种排序算法都已被封装成库函数供程序员使用，但了解排序算法的思想和原理，对于编写高质量的软件，显得非常重要。 本文介绍了常见的排序算法，从算法思想，复杂度和使用场景等方面做了总结。

2. 几个概念

（1）排序稳定：如果两个数相同，对他们进行的排序结果为他们的相对顺序不变。例如A={1,2,1,2,1}这里排序之后是A = {1,1,1,2,2} 稳定就是排序后第一个1就是排序前的第一个1，第二个1就是排序前第二个1，第三个1就是排序前的第三个1。同理2也是一样。不稳定就是他们的顺序与开始顺序不一致。
（2）原地排序：指不申请多余的空间进行的排序，就是在原来的排序数据中比较和交换的排序。例如快速排序，堆排序等都是原地排序，合并排序，计数排序等不是原地排序。
总体上说，排序算法有两种设计思路，一种是基于比较，另一种不是基于比较。《算法导论》一书给出了这样一个证明：“基于比较的算法的最优时间复杂度是O（N lg N）”。对于基于比较的算法，有三种设计思路，分别为：插入排序，交换排序和选择排序。非基于比较的排序算法时间复杂度为O(lg N)，之所以复杂度如此低，是因为它们一般对排序数据有特殊要求。如计数排序要求数据范围不会太大，基数排序要求数据可以分解成多个属性等。

3. 基于比较的排序算法

正如前一节介绍的，基于比较的排序算法有三种设计思路，分别为插入，交换和选择。对于插入排序，主要有直接插入排序，希尔排序；对于交换排序，主要有冒泡排序，快速排序；对于选择排序，主要有简单选择排序，堆排序；其它排序：归并排序。

3.1. 插入排序

3.1.1. 直接插入排序

特点：稳定排序，原地排序，时间复杂度O(N*N)
思想：将所有待排序数据分成两个序列，一个是有序序列S，另一个是待排序序列U，初始时，S为空，U为所有数据组成的数列，然后依次将U中的数据插到有序序列S中，直到U变为空。
适用场景：小规模数据，数据已经基本有序。
基本代码：

```
int sortPointer(void **array,int arrayLength,int(*cmp)(const void *,const void *))
{
    if(!array)
        return -1;
    if(arrayLength<=0)
        return -1;
    if(!cmp)
        return -1;
    int i;
    for(i=1;i<arrayLength;i++)
    {
        int j=i-1;
        while(j>=0 && cmp(array[j],array[i])>0)
        {
            array[j+1]=array[j];
            j--;
        }
        array[j+1]=array[i];
    }
    return 0;
}
```

```
int sortValues(void *base, int n, int s, int(*cmp)(const void *,const void *))
{
    if(!base)
        return -1;
    if(n<=0)
        return -1;
    if(s<=0)
        return -1;
    if(!cmp)
        return -1;
    int i;
    void *saved=malloc(s);
    if(!saved)
        return -1;
    for(i=1;i<n;i++)
    {
        int j=i-1;
        void *value=base+i*s;
        while(j>=0 && cmp(base + j*s,value) >0)
        {
            j--;
        }
        if(++j==i)
            continue;
        memmove(saved,value,s);
        memmove(base+(j+1)*s,base+j*s,s*(i-j));
        memmove(base+J*s,saved,s);
    }
    free(saved);
    return 0;
}
```

3.1.2. 希尔排序

特点：非稳定排序，原地排序，时间复杂度O(n^lamda)(1 < lamda < 2), lamda和每次步长选择有关。
思想：增量缩小排序。先将序列按增量划分为元素个数近似的若干组，使用直接插入排序法对每组进行排序，然后不断缩小增量直至为1，最后使用直接插入排序完成排序。
适用场景：因为增量初始值不容易选择，所以该算法不常用。

3.2. 交换排序

3.2.1. 冒泡排序

特点：稳定排序，原地排序，时间复杂度O(N*N)
思想：将整个序列分为无序和有序两个子序列，不断通过交换较大元素至无序子序列首完成排序。
适用场景：同直接插入排序类似

3.2.2. 快速排序

特点：不稳定排序，原地排序，时间复杂度O(N*lg N)
思想：不断寻找一个序列的枢轴点，然后分别把小于和大于枢轴点的数据移到枢轴点两边，然后在两边数列中继续这样的操作，直至全部序列排序完成。
适用场景：应用很广泛，差不多各种语言均提供了快排API
基本代码：

```
int partition(void **array,int low,int high,int (*cmp)(const void *,const void *))
{
    void *pivot=array[low];
    int i;
    int cursor=low+1;
    for(i=low+1;i<=high;i++)
    {
        if(cmp(pivot,array[i])>0)
        {
            void *store=array[i];
            array[i]=array[cursor];
            array[cursor]=store;
            cursor++;
        }
    }
    cursor--;
    array[low]=array[cursor];
    array[cursor]=pivot;
    return cursor;
}

int qsort(void **array,int low,int high,int (*cmp)(const void *,const void *))
{
    if(!array)
        return -1;
        if(low<0)
        return -1;
    if(high<0)
        return -1;
    if(low<high)
    {
        int cursor=partition(array,low,high,cmp);
        qsort(array,low,cursor-1,cmp);
        qsort(array,cursor+1,high,cmp);
    }
    return 0;
}
```

优化：
利用存储子任务的栈消除递归。
选择基于3中值分区的中枢值。
设定快排数组长度的最小值，小于此值时使用插入排序。
首先处理大的字数组，减少栈空间使用。

3.3. 选择排序

3.3.1. 简单选择排序

特点：不稳定排序（比如对3 3 2三个数进行排序，第一个3会与2交换），原地排序，时间复杂度O(N*N)
思想：将序列划分为无序和有序两个子序列，寻找无序序列中的最小（大）值和无序序列的首元素交换，有序区扩大一个，循环下去，最终完成全部排序。
适用场景：交换少

3.3.2. 堆排序

特点：非稳定排序，原地排序，时间复杂度O(N*lg N)
思想：小顶堆或者大顶堆
适用场景：不如快排广泛
基本代码：

```
void heapify(void **array,int(*cmp)(const void *,const void *),int index,int length)
{
    int left=2*index+1;
    int right=2*index+2;
    int largest;
    if(left<length && cmp(array[left],array[index])>0)
    {
        largest=left;
    }
    else
    {
        largest=index;
    }
    if(right<length && cmp(array[right],array[largest])>0)
    {
        largest=right;
    }
    if(largest != index)
    {
        void *tmp=array[index];
        array[index]=array[largest];
        array[largest]=tmp;
        heapify(array,cmp,largest,length);
    }
}
void buildHeap(void **array,int(*cmp)(const void *,const void *),int length)
{
    int i;
    for(i=length/2-1;i>=0;i--)
    {
        heapify(array,cmp,i,length);
    }
}
```

3.4. 其它排序

3.4.1. 归并排序

特点：稳定排序，非原地排序，时间复杂度O(N*N)
思想：首先，将整个序列（共N个元素）看成N个有序子序列，然后依次合并相邻的两个子序列，这样一直下去，直至变成一个整体有序的序列。
适用场景：外部排序

4. 非基于比较的排序算法

非基于比较的排序算法主要有三种，分别为：基数排序，桶排序和计数排序。这些算法均是针对特殊数据的，不如要求数据分布均匀，数据偏差不会太大。采用的思想均是内存换时间，因而全是非原地排序。

4.1. 基数排序

特点：稳定排序，非原地排序，时间复杂度O(N)
思想：把每个数据看成d个属性组成，依次按照d个属性对数据排序（每轮排序可采用计数排序），复杂度为O(d*N)
适用场景：数据明显有几个关键字或者几个属性组成

4.2. 桶排序

特点：稳定排序，非原地排序，时间复杂度O(N)
思想：将数据按大小分到若干个桶（比如链表）里面，每个桶内部采用简单排序算法进行排序。
适用场景：0

4.3. 计数排序

特点：稳定排序，非原地排序，时间复杂度O(N)
思想：对每个数据出现次数进行技术（用hash方法计数，最简单的hash是数组！），然后从大到小或者从小到大输出每个数据。
使用场景：比基数排序和桶排序广泛得多。

5. 总结

排序方法	平均时间	最坏情况	辅助存储
简单排序	O(n^2)	O(n^2)	O(1)
快速排序	O(nlogn)	O(n^2)	O(logn)
堆排序	O(nlogn)	O(nlogn)	O(1)
归并排序	O(nlogn)	O(nlogn)	O(n)
基数排序	O(d(n+rd))	O(d(n+rd))	O(rd)

对于基于比较的排序算法，大部分简单排序(直接插入排序，选择排序和冒泡排序)都是稳定排序，选择排序除外；大部分高级排序（除简单排序以外的）都是不稳定排序，归并排序除外，但归并排序

需要额外的存储空间。对于非基于比较的排序算法，它们都对数据规律有特殊要求，且采用了内存换时间的思想。排序算法如此之多，往往需要根据实际应用选择最适合的排序算法。

6. 参考资料

（1）博文《排序算法总结》：
http://www.cppblog.com/shongbee2/archive/2009/04/25/81058.html
（2）博文：《八大排序算法》：
http://blog.csdn.net/yexinghai/archive/2009/10/10/4649923.aspx

原创文章，转载请注明：转载自董的博客
本文链接地址:http://dongxicheng.org/structure/sort/
作者：**Dong**，作者介绍：http://dongxicheng.org/about/

抱歉，暂停评论。

输入关键字搜索 搜索

分类目录

- Coding (40)
- Tools (9)
- Bugs (10)
- Other (10)
- Principle (8)