逝者如斯, 不舍昼夜
尘世中一个迷途小书童，读书太少，想得太多

常用排序算法总结(一)
Posted on 2016-03-28 22:13 SteveWang 阅读(26057) 评论(3) 编辑 收藏

目录

我们通常所说的排序算法往往指的是内部排序算法，即数据记录在内存中进行排序。

排序算法大体可分为两种：

一种是比较排序，时间复杂度O(nlogn) ~ O(n^2)，主要有：冒泡排序，选择排序，插入排序，归并排序，堆排序，快速排序等。

另一种是非比较排序，时间复杂度可以达到O(n)，主要有：计数排序，基数排序，桶排序等。

这里我们来探讨一下常用的比较排序算法，非比较排序算法将在后续文章中介绍。下表给出了常见比较排序算法的性能：

表 9-10-1

排序方法	平均情况	最好情况	最坏情况	辅助空间	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
直接插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
希尔排序	$O(n\log n) \sim O(n^2)$	$O(n^{1.3})$	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(1)$	不稳定
归并排序	$O(n\log n)$	$O(n\log n)$	$O(n\log n)$	$O(n)$	稳定
快速排序	$O(n\log n)$	$O(n\log n)$	$O(n^2)$	$O(\log n) \sim O(n)$	不稳定

这里有一点我们很容易忽略的是排序算法的稳定性(腾讯校招2016笔试题曾考过)。

排序算法稳定性的简单形式化定义为：如果$A_i = A_j$，排序前A_i在A_j之前，排序后A_i还在A_j之前，则称这种排序算法是稳定的。通俗地讲就是保证排序前后两个相等的数的相对顺序不变。

对于不稳定的排序算法，只要举出一个实例，即可说明它的不稳定性；而对于稳定的排序算法，必须对算法进行分析从而得到稳定的特性。需要注意的是，排序算法是否为稳定的是由具体算法决定的，不稳定的算法在某种条件下可以变为稳定的算法，而稳定的算法在某种条件下也可以变为不稳定的算法。

例如，对于冒泡排序，原本是稳定的排序算法，如果将记录交换的条件改成A[i] >= A[i + 1]，则两个相等的记录就会交换位置，从而变成不稳定的排序算法。

其次，说一下排序算法稳定性的好处。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，第一个键排序的结果可以为第二个键排序所用。基数排序就是这样，先按低位排序，逐次按高位排序，低位排序后元素的顺序在高位也相同时是不会改变的。

冒泡排序(Bubble Sort)

冒泡排序是一种极其简单的排序算法，也是我所学的第一个排序算法。它重复地走访过要排序的元素，一次比较相邻两个元素，如果他们的顺序错误就把他们调换过来，直到没有元素再需要交换，排序完成。这个算法的名字由来是因为越小(或越大)的元素会经由交换慢慢"浮"到数列的顶端。

冒泡排序算法的运作如下：

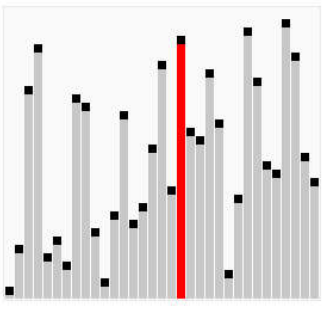
1. 比较相邻的元素，如果前一个比后一个大，就把它们两个调换位置。
2. 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。这步做完后，最后的元素会是最大的数。

3. 针对所有的元素重复以上的步骤，除了最后一个。
4. 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。

由于它的简洁，冒泡排序通常被用来对于程序设计入门的学生介绍算法的概念。冒泡排序的代码如下：

```
#include <stdio.h>

// 分类 ———————— 内部比较排序
// 数据结构 ———————— 数组
// 最差时间复杂度 —— O(n^2)
// 最优时间复杂度 —— 如果能在内部循环第一次运行时,使用一个旗标来表示有无需要交换的可能,可以把最优时间复杂度降低到O(n)
// 平均时间复杂度 —— O(n^2)
// 所需辅助空间 ———— O(1)
// 稳定性 ———————— 稳定

void exchange(int A[], int i, int j)        // 交换A[i]和A[j]
{
    int temp = A[i];
    A[i] = A[j];
    A[j] = temp;
}

int main()
{
    int A[] = { 6, 5, 3, 1, 8, 7, 2, 4 };    // 从小到大冒泡排序
    int n = sizeof(A) / sizeof(int);
    for (int j = 0; j < n - 1; j++)             // 每次最大元素就像气泡一样"浮"到数组的最后
    {
        for (int i = 0; i < n - 1 - j; i++)    // 依次比较相邻的两个元素,使较大的那个向后移
        {
            if (A[i] > A[i + 1])            // 如果条件改成A[i] >= A[i + 1],则变为不稳定的排序算法
            {
                exchange(A, i, i + 1);
            }
        }
    }
    printf("冒泡排序结果: ");
    for (int i = 0; i < n; i++)
    {
        printf("%d ", A[i]);
    }
    printf("\n");
    return 0;
}
```

上述代码对序列{ 6, 5, 3, 1, 8, 7, 2, 4 }进行冒泡排序的实现过程如下



使用冒泡排序为一列数字进行排序的过程如右图所示：

尽管冒泡排序是最容易了解和实现的排序算法之一，但它对于少数元素之外的数列排序是很没有效率的。

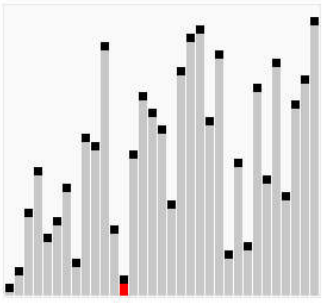
冒泡排序的改进：鸡尾酒排序

冒泡排序的改进：鸡尾酒排序

鸡尾酒排序，也叫定向冒泡排序，是冒泡排序的一种改进。此算法与冒泡排序的不同处在于从低到高然后从高到低，而冒泡排序则仅从低到高去比较序列里的每个元素。他可以得到比冒泡排序稍微好一点的效能。

鸡尾酒排序的代码如下：

```
#include <stdio.h>

// 分类 ————————— 内部比较排序
// 数据结构 ———————— 数组
// 最差时间复杂度 —— O(n^2)
// 最优时间复杂度 —— 如果序列在一开始已经大部分排序过的话,会接近O(n)
// 平均时间复杂度 —— O(n^2)
// 所需辅助空间 ——— O(1)
// 稳定性 ————————— 稳定

void exchange(int A[], int i, int j)        // 交换A[i]和A[j]
{
    int temp = A[i];
    A[i] = A[j];
    A[j] = temp;
}

int main()
{
    int A[] = { 6, 5, 3, 1, 8, 7, 2, 4 };   // 从小到大定向冒泡排序
    int n = sizeof(A) / sizeof(int);
    int left = 0;                           // 初始化边界
    int right = n - 1;
    while (left < right)
    {
        for (int i = left; i < right; i++)  // 前半轮,将最大元素放到后面
            if (A[i] > A[i + 1])
            {
                exchange(A, i, i + 1);
            }
        right--;
        for (int i = right; i > left; i--)  // 后半轮,将最小元素放到前面
            if (A[i - 1] > A[i])
            {
                exchange(A, i - 1, i);
            }
        left++;
    }
    printf("鸡尾酒排序结果：");
    for (int i = 0; i < n; i++)
    {
        printf("%d ", A[i]);
    }
    printf("\n");
    return 0;
}
```



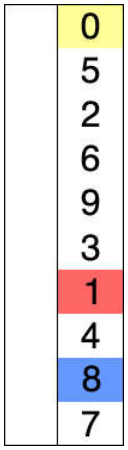
使用鸡尾酒排序为一列数字进行排序的过程如右图所示：

以序列(2,3,4,5,1)为例，鸡尾酒排序只需要访问一次序列就可以完成排序，但如果使用冒泡排序则需要四次。但是在乱数序列的状态下，鸡尾酒排序与冒泡排序的效率都很差劲。

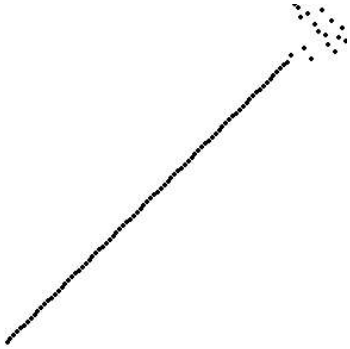
选择排序(Selection Sort)

选择排序也是一种简单直观的排序算法。它的工作原理很容易理解：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，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。代码如下：

```c
#include <stdio.h>

// 分类 ————————— 内部比较排序
// 数据结构 ————————— 数组
// 最差时间复杂度 —— O(n^2)
// 最优时间复杂度 —— O(n^2)
// 平均时间复杂度 —— O(n^2)
// 所需辅助空间 ——— O(1)
// 稳定性 ————————— 不稳定

void exchange(int A[], int i, int j)        // 交换A[i]和A[j]
{
    int temp = A[i];
    A[i] = A[j];
    A[j] = temp;
}

int main()
{
    int A[] = { 8, 5, 2, 6, 9, 3, 1, 4, 0, 7 }; // 从小到大选择排序
    int n = sizeof(A) / sizeof(int);
    int i, j, min;
    for (i = 0; i <= n - 2; i++)                // 已排序序列的末尾
    {
        min = i;
        for (j = i + 1; j <= n - 1; j++)        // 未排序序列
        {
            if (A[j] < A[min])// 依次找出未排序序列中的最小值,存放到已排序序列的末尾
            {
                min = j;
            }
        }
        if (min != i)
        {
            exchange(A, min, i);    // 该操作很有可能把稳定性打乱,所以选择排序是不稳定的排序算法
        }
    }
    printf("选择排序结果: ");
    for (i = 0; i < n; i++)
    {
        printf("%d ",A[i]);
    }
    printf("\n");
    return 0;
}
```



上述代码对序列{ 8, 5, 2, 6, 9, 3, 1, 4, 0, 7 }进行选择排序的实现过程如右图



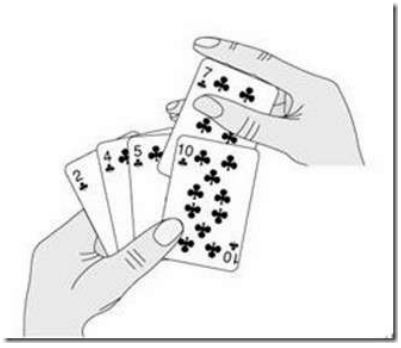
使用选择排序为一列数字进行排序的宏观过程：

选择排序是不稳定的排序算法，不稳定发生在最小元素与**A[i]**交换的时刻。

比如序列：{ 5, 8, 5, 2, 9 }，一次选择的最小元素是2，然后把2和第一个5进行交换，从而改变了两个元素5的相对次序。

插入排序(Insertion Sort)

插入排序是一种简单直观的排序算法。它的工作原理非常类似于我们抓扑克牌



对于未排序数据(右手抓到的牌)，在已排序序列(左手已经排好序的手牌)中从后向前扫描，找到相应位置并插入。

插入排序在实现上，通常采用in-place排序（即只需用到O(1)的额外空间的排序），因而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，为最新元素提供插入空间。

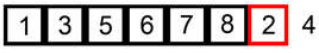
具体算法描述如下：

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

插入排序的代码如下：

```
#include <stdio.h>

// 分类 ———————— 内部比较排序
// 数据结构 ———————— 数组
// 最差时间复杂度 —— 最坏情况为输入序列是降序排列的,此时时间复杂度O(n^2)
// 最优时间复杂度 —— 最好情况为输入序列是升序排列的,此时时间复杂度O(n)
// 平均时间复杂度 —— O(n^2)
// 所需辅助空间 ———— O(1)
// 稳定性 ———————— 稳定

int main()
{
    int A[] = { 6, 5, 3, 1, 8, 7, 2, 4 };// 从小到大插入排序
    int n = sizeof(A) / sizeof(int);
    int i, j, get;

    for (i = 1; i < n; i++)             // 类似抓扑克牌排序
    {
        get = A[i];                     // 右手抓到一张扑克牌
        j = i - 1;                      // 拿在左手上的牌总是排序好的
        while (j >= 0 && A[j] > get)    // 将抓到的牌与手牌从右向左进行比较
        {
            A[j + 1] = A[j];            // 如果该手牌比抓到的牌大,就将其右移
            j--;
        }
        A[j + 1] = get;// 直到该手牌比抓到的牌小(或二者相等),将抓到的牌插入到该手牌右边(相等元素的相对次序未变,所以插入排序是稳定的)
    }
    printf("插入排序结果：");
    for (i = 0; i < n; i++)
    {
        printf("%d ", A[i]);
    }
    printf("\n");
    return 0;
}
```

上述代码对序列{ 6, 5, 3, 1, 8, 7, 2, 4 }进行插入排序的实现过程如下



使用插入排序为一列数字进行排序的宏观过程：

插入排序不适合对于数据量比较大的排序应用。但是，如果需要排序的数据量很小，比如量级小于千，那么插入排序还是一个不错的选择。 插入排序在工业级库中也有着广泛的应用，在STL的sort算法和stdlib的qsort算法中，都将插入排序作为快速排序的补充，用于少量元素的排序（通常为8个或以下）。

插入排序的改进：二分插入排序

对于插入排序，如果比较操作的代价比交换操作大的话，可以采用二分查找法来减少比较操作的数目，我们称为二分插入排序，代码如下：

```
#include <stdio.h>

// 分类 ————————— 内部比较排序
// 数据结构 ———————— 数组
// 最差时间复杂度 —— O(n^2)
// 最优时间复杂度 —— O(nlogn)
// 平均时间复杂度 —— O(n^2)
// 所需辅助空间 ——— O(1)
// 稳定性 ———————— 稳定

int main()
{
    int A[] = { 5, 2, 9, 4, 7, 6, 1, 3, 8 };// 从小到大二分插入排序
    int n = sizeof(A) / sizeof(int);
    int i, j, get, left, right, middle;

    for (i = 1; i < n; i++)                 // 类似抓扑克牌排序
    {
        get = A[i];                         // 右手抓到一张扑克牌
        left = 0;                           // 拿在左手上的牌总是排序好的,所以可以用二分法
        right = i - 1;                      // 手牌左右边界进行初始化
        while (left <= right)               // 采用二分法定位新牌的位置
        {
            middle = (left + right) / 2;
            if (A[middle] > get)
                right = middle - 1;
            else
                left = middle + 1;
        }
        for (j = i - 1; j >= left; j--)     // 将欲插入新牌位置右边的牌整体向右移动一个单位
        {
            A[j + 1] = A[j];
        }
        A[left] = get;                      // 将抓到的牌插入手牌
    }
    printf("二分插入排序结果：");
    for (i = 0; i < n; i++)
    {
        printf("%d ", A[i]);
    }
    printf("\n");
    return 0;
}
```

当n较大时，二分插入排序的比较次数比直接插入排序的最差情况好得多，但比直接插入排序的最好情况要差，所当以元素初始序列已经接近升序时，直接插入排序比二分插入排序比较次数少。二分插入排序元素移动次数与直接插入排序相同，依赖于元素初始序列。

插入排序的更高效改进：希尔排序(Shell Sort)

希尔排序，也叫递减增量排序，是插入排序的一种更高效的改进版本。希尔排序是不稳定的排序算法。

希尔排序是基于插入排序的以下两点性质而提出改进方法的：

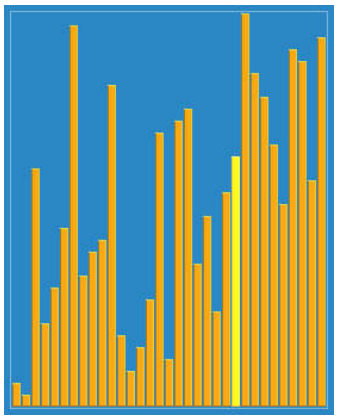
- 插入排序在对几乎已经排好序的数据操作时，效率高，即可以达到线性排序的效率
- 但插入排序一般来说是低效的，因为插入排序每次只能将数据移动一位

希尔排序通过将比较的全部元素分为几个区域来提升插入排序的性能。这样可以让一个元素可以一次性地朝最终位置前进一大步。然后算法再取越来越小的步长进行排序，算法的最后一步就是普通的插入排序，但是到了这步，需排序的数据几乎是已排好的了（此时插入排序较快）。

假设有一个很小的数据在一个已按升序排好序的数组的末端。如果用复杂度为O(n^2)的排序（冒泡排序或直接插入排序），可能会进行n次的比较和交换才能将该数据移至正确位置。而希尔排序会用较大的步长移动数据，所以小数据只需进行少数比较和交换即可到正确位置。

希尔排序的代码如下：

```
#include <stdio.h>

// 分类 ———————— 内部比较排序
// 数据结构 ———————— 数组
// 最差时间复杂度 —— 根据步长序列的不同而不同。已知最好的为0(n(logn)^2)
// 最优时间复杂度 —— O(n)
// 平均时间复杂度 —— 根据步长序列的不同而不同。
// 所需辅助空间 ———— O(1)
// 稳定性 ———————— 不稳定

int main()
{
    int A[] = { 5, 2, 9, 4, 7, 6, 1, 3, 8 };// 从小到大希尔排序
    int n = sizeof(A) / sizeof(int);
    int i, j, get;
    int h = 0;
    while (h <= n)                          // 生成初始增量
    {
        h = 3*h + 1;
    }
    while (h >= 1)
    {
        for (i = h; i < n; i++)
        {
            j = i - h;
            get = A[i];
            while ((j >= 0) && (A[j] > get))
            {
                A[j + h] = A[j];
                j = j - h;
            }
            A[j + h] = get;
        }
        h = (h - 1) / 3;                    // 递减增量
    }
    printf("希尔排序结果: ");
    for (i = 0; i < n; i++)
    {
        printf("%d ", A[i]);
    }
    printf("\n");
    return 0;
}
```



以23, 10, 4, 1的步长序列进行希尔排序：

希尔排序是不稳定的排序算法，虽然一次插入排序是稳定的，不会改变相同元素的相对顺序，但在不同的插入排序过程中，相同的元素可能在各自的插入排序中移动，最后其稳定性就会被打乱。

比如序列：{ 3, 5, 10, 8, 7, 2, 8, 1, 20, 6 }，h=2时分成两个子序列 { 3, 10, 7, 8, 20 } 和 { 5, 8, 2, 1, 6 } ，未排序之前第二个子序列中的8在前面，现在对两个子序列进行插入排序，得到 { 3, 7, 8, 10, 20 } 和 { 1, 2, 5, 6, 8 } ，即 { 3, 1, 7, 2, 8, 5, 10, 6, 20, 8 } ，两个8的相对次序发生了改变。

归并排序(Merge Sort)

归并排序是创建在归并操作上的一种有效的排序算法，效率为O(nlogn)，1945年由冯·诺伊曼首次提出。

归并排序的实现分为递归实现与非递归(迭代)实现。递归实现的归并排序是算法设计中分治策略的典型应用，我们将一个大问题分割成小问题分别解决，然后用所有小问题的答案来解决整个大问题。非递归(迭代)实现的归并排序首先进行是两两归并，然后四四归并，然后是八八归并，一直下去直到归并了整个数组。

归并排序算法主要依赖归并(Merge)操作。归并操作指的是将两个已经排序的序列合并成一个序列的操作,归并操作步骤如下：

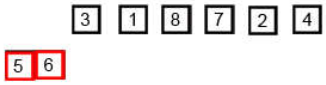
1. 申请空间，使其大小为两个已经排序序列之和，该空间用来存放合并后的序列
2. 设定两个指针，最初位置分别为两个已经排序序列的起始位置
3. 比较两个指针所指向的元素，选择相对小的元素放入到合并空间，并移动指针到下一位置
4. 重复步骤3直到某一指针到达序列尾
5. 将另一序列剩下的所有元素直接复制到合并序列尾

归并排序的代码如下：

```
#include <stdio.h>
#include <limits.h>                // 包含极限值的头文件，这里用到了无穷大INT_MAX

// 分类 ————————— 内部比较排序
// 数据结构 ———————— 数组
// 最差时间复杂度 —— O(nlogn)
// 最优时间复杂度 —— O(nlogn)
// 平均时间复杂度 —— O(nlogn)
// 所需辅助空间 ——— O(n)
// 稳定性 ————————— 稳定

int L[10];    // 两个子数组定义成全局变量（辅助存储空间,大小正比于元素的个数）
int R[10];

void merge(int A[], int left, int middle, int right)// 合并两个已排好序的数组A[left...middle]和A[middle+1...right]
{
    int n1 = middle - left + 1;     // 两个数组的大小
    int n2 = right - middle;
    for (int i = 0; i < n1; i++)    // 把两部分分别拷贝到两个数组中
        L[i] = A[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = A[middle + j + 1];
    L[n1] = INT_MAX;                // 使用无穷大作为哨兵值放在子数组的末尾
    R[n2] = INT_MAX;                // 这样可以免去检查某个子数组是否已读完的步骤
    int i = 0;
    int j = 0;
    for (int k = left; k <= right; k++) // 依次比较两个子数组中的值，每次取出更小的那一个放入原数组
    {
        if (L[i] <= R[j])
        {
            A[k] = L[i];
            i++;
        }
        else
        {
            A[k] = R[j];
            j++;
        }
    }

}

void mergesort_recursion(int A[], int left, int right) // 递归实现的归并排序(自顶向下)
{
    int middle = (left + right) / 2;
    if (left < right)          // 当待排序的序列长度为1时(left == right)，递归“开始回升”
    {
        mergesort_recursion(A, left, middle);
        mergesort_recursion(A, middle + 1, right);
        merge(A, left, middle, right);
    }
}

void mergesort_iteration(int A[], int left, int right)  // 非递归(迭代)实现的归并排序(自底向上)
{
    int low, middle, high;    // 子数组索引,前一个为A[low...middle],后一个子数组为A[middle+1...high]
    for (int size = 1; size <= right - left; size *= 2) // 子数组的大小初始为1,每轮翻倍
    {
        low = left;
        while (low + size - 1 <= right - 1 )// 后一个子数组存在(需要归并)
        {
            middle = low + size - 1;
            high = middle + size;
            if (high > right)               // 后一个子数组大小不足size
                high = right;
            merge(A, low, middle, high);
            low = high + 1;                 // 前一个子数组索引向后移动
```

```
        }
    }
}

int main()
{
    int A1[] = { 6, 5, 3, 1, 8, 7, 2, 4 };    // 从小到大归并排序
    int A2[] = { 6, 5, 3, 1, 8, 7, 2, 4 };
    int n1 = sizeof(A1) / sizeof(int);
    int n2 = sizeof(A2) / sizeof(int);
    mergesort_recursion(A1, 0, n1 - 1);       // 递归实现
    mergesort_iteration(A2, 0, n2 - 1);       // 非递归实现
    printf("递归实现的归并排序结果：");
    for (int i = 0; i < n1; i++)
    {
        printf("%d ",A1[i]);
    }
    printf("\n");
    printf("非递归实现的归并排序结果：");
    for (int i = 0; i < n2; i++)
    {
        printf("%d ", A2[i]);
    }
    printf("\n");
    return 0;
}
```

上述代码对序列{ 6, 5, 3, 1, 8, 7, 2, 4 }进行归并排序的实例如下



使用归并排序为一列数字进行排序的宏观过程：

堆排序(Heapsort)

堆排序是指利用堆这种数据结构所设计的一种排序算法。堆是一个近似完全二叉树的结构（通常堆是通过一维数组来实现的），并同时满足堆的性质：即子结点的键值总是小于（或者大于）它的父节点。

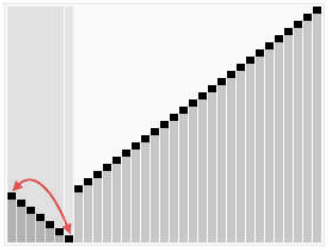
我们可以很容易的定义堆排序的过程：

1. 创建一个堆
2. 把堆顶元素(最大值)和堆尾元素互换
3. 把堆的尺寸缩小1，并调用heapify(A, 0)从新的堆顶元素开始进行堆调整
4. 重复步骤2，直到堆的尺寸为1

堆排序的代码如下：

```
#include <stdio.h>

// 分类 ———————— 内部比较排序
// 数据结构 ———————— 数组
// 最差时间复杂度 —— O(nlogn)
// 最优时间复杂度 —— O(nlogn)
// 平均时间复杂度 —— O(nlogn)
// 所需辅助空间 ———— O(1)
// 稳定性 ———————— 不稳定

int heapsize;    // 堆大小

void exchange(int A[], int i, int j)    // 交换A[i]和A[j]
```

```
void exchange(int A[], int i, int j)    // 交换A[i]和A[j]
{
    int temp = A[i];
    A[i] = A[j];
    A[j] = temp;
}

void heapify(int A[], int i)            // 堆调整函数(这里使用的是最大堆)
{
    int leftchild = 2 * i + 1;          // 左孩子索引
    int rightchild = 2 * i + 2;         // 右孩子索引
    int largest;                        // 选出当前结点与左右孩子之中的最大值
    if (leftchild < heapsize && A[leftchild] > A[i])
        largest = leftchild;
    else
        largest = i;
    if (rightchild < heapsize && A[rightchild] > A[largest])
        largest = rightchild;
    if (largest != i)
    {
        exchange(A, i, largest);        // 把当前结点和它的最大(直接)子节点进行交换
        heapify(A, largest);            // 递归调用，继续从当前结点向下进行堆调整
    }
}

void buildheap(int A[], int n)          // 建堆函数
{
    heapsize = n;
    for (int i = heapsize / 2 - 1; i >= 0; i--) // 对每一个非叶结点
        heapify(A, i);                  // 不断的堆调整
}

void heapsort(int A[], int n)
{
    buildheap(A, n);
    for (int i = n - 1; i >= 1; i--)
    {
        exchange(A, 0, i); // 将堆顶元素(当前最大值)与堆的最后一个元素互换(该操作很有可能把后面元素的稳定性打乱,所以堆排序是不稳定的排序算法)
        heapsize--;                     // 从堆中去掉最后一个元素
        heapify(A, 0);                  // 从新的堆顶元素开始进行堆调整
    }
}

int main()
{
    int A[] = { 5, 2, 9, 4, 7, 6, 1, 3, 8 };// 从小到大堆排序
    int n = sizeof(A) / sizeof(int);
    heapsort(A, n);
    printf("堆排序结果：");
    for (int i = 0; i < n; i++)
    {
        printf("%d ", A[i]);
    }
    printf("\n");
    return 0;
}
```



堆排序算法的演示：

动画中在排序过程之前简单的表现了创建堆的过程以及堆的逻辑结构。

堆排序是不稳定的排序算法，不稳定发生在堆顶元素与**A[i]**交换的时刻。

比如序列：{ 9, 5, 7, 5 }，堆顶元素是9，堆排序下一步将9和第二个5进行交换，得到序列 { 5, 5, 7, 9 }，再进行堆调整得到 { 7, 5, 5, 9 }，重复之前的操作最后得到{ 5, 5, 7, 9 }从而改变了两个5的相对次序。

快速排序(Quicksort)

快速排序是由东尼·霍尔所发展的一种排序算法。在平均状况下，排序n个元素要O(nlogn)次比较。在最坏状况下则需要O(n^2)次比较，但这种状况并不常见。事实上，快速排序通常明显比其他O(nlogn)算法更快，因为它的内部循环可以在大部分的架构上很有效率地被实现出来。

快速排序使用分治策略(Divide and Conquer)来把一个序列分为两个子序列。步骤为：

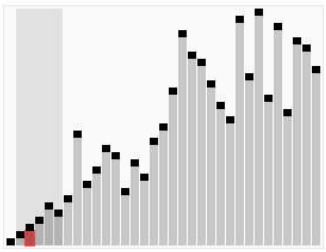
1. 从序列中挑出一个元素，作为"基准"(pivot).
2. 把所有比基准值小的元素放在基准前面，所有比基准值大的元素放在基准的后面（相同的数可以到任一边），这个称为分区(partition)操作。

(partition)操作。

3.对每个分区递归地进行步骤1~3，递归的结束条件是序列的大小是0或1，这时整体已经被排好序了。

快速排序的代码如下：

```
#include <stdio.h>

// 分类 ————————— 内部比较排序
// 数据结构 ——————— 数组
// 最差时间复杂度 —— 每次选取的基准都是最大的元素（或者每次都是最小），导致每次只划分出了一个子序列，需要进行n-1次划分才能结束递归，时间复杂度为O(n^2)
// 最优时间复杂度 —— 每次选取的基准都能使划分均匀，只需要logn次划分就能结束递归，时间复杂度为O(nlogn)
// 平均时间复杂度 —— O(nlogn)
// 所需辅助空间 ——— O(logn)～O(n),主要是递归造成的栈空间的使用(用来保存left和right等局部变量),取决于递归树的深度
//                  一般为O(logn),最差为O(n)（基本有序的情况）
// 稳定性 ————————— 不稳定

void exchange(int A[], int i, int j)        // 交换A[i]和A[j]
{
    int temp = A[i];
    A[i] = A[j];
    A[j] = temp;
}

int partition(int A[], int left, int right)  // 划分函数
{
    int pivot = A[right];                    // 选择最后一个元素作为基准
    int tail = left - 1;                     // tail为小于基准的子数组最后一个元素的索引
    for (int i = left; i < right; i++)       // 遍历基准以外的其他元素
    {
        if (A[i] <= pivot)                   // 把小于等于基准的元素放到前一个子数组中
        {
            tail++;
            exchange(A, tail, i);
        }
    }
    exchange(A, tail + 1, right);            // 最后把基准放到前一个子数组的后边,剩下的子数组既是大于基准的子数组
                                             // 该操作很有可能把后面元素的稳定性打乱,所以快速排序是不稳定的排序算法
    return tail + 1;                         // 返回基准的索引
}

void quicksort(int A[], int left, int right)
{
    int pivot_index;                         // 基准的索引
    if (left < right)
    {
        pivot_index = partition(A, left, right);
        quicksort(A, left, pivot_index-1);
        quicksort(A, pivot_index+1, right);
    }

}

int main()
{
    int A[] = { 5, 2, 9, 4, 7, 6, 1, 3, 8 };// 从小到大快速排序
    int n = sizeof(A) / sizeof(int);
    quicksort(A, 0, n - 1);
    printf("快速排序结果: ");
    for (int i = 0; i < n; i++)
    {
        printf("%d ",A[i]);
    }
    printf("\n");

    return 0;
}
```



使用快速排序法对一列数字进行排序的过程：

快速排序是不稳定的排序算法，不稳定发生在基准元素与**A[tail+1]**交换的时刻。

比如序列：{ 1, 3, 4, 2, 8, 9, 8, 7, 5 }，基准元素是5，一次划分操作后5要和第一个8进行交换，从而改变了两个元素8的相对次序。

分类: 算法

标签: 算法, 排序

好文要顶 关注我 收藏该文

SteveWang
关注 - 5
粉丝 - 23
+加关注

4 0

« 上一篇：找出数组中出现次数最多的那个数——主元素问题
» 下一篇：常用排序算法总结(二)

Feedback

#1楼
2017-01-04 18:24 by 331902579

图文并茂，很容易理解！
快速排序中：
if (A[i] <= pivot)
{
tail++;
exchange(A, tail, i);
}
可以修改成：
if (A[i] <= pivot)
{
tail++;
if(tail != i)
exchange(A, tail, i);
}
提高效率。

支持(0) 反对(0)

#2楼
2017-03-22 18:12 by 喜欢兰花山丘

"一种是比较排序，时间复杂度最少可达到O(n log n)，主要有：冒泡排序，选择排序 "
这句话 怪怪的, 可以改成 时间复杂度O(nlogn) ~ O(n^2) . 哈哈

支持(0) 反对(0)

#3楼[楼主]
2017-03-24 20:47 by SteveWang

@ 喜欢兰花山丘
引用
"一种是比较排序，时间复杂度最少可达到O(n log n)，主要有：冒泡排序，选择排序 "

这句话 怪怪的, 可以改成 时间复杂度O(nlogn) ~ O(n^2) . 哈哈

感谢您的建议 已更正

支持(0) 反对(0)

刷新评论 刷新页面 返回顶部

注册用户登录后才能发表评论，请 登录 或 注册，访问网站首页。