直接插入排序

算法流程

认为第1个元素是有序的，from 2 to end

在表头存储当前待排元素，防止向前搜索合适的插入位置时数组越界，即哨兵。

遍历从当前待排元素位置的前一个位置到表头，直到找到第一个不大于待排元素a[0]的元素，期间扫描到的大于a[0]的元素全部后移一位（会覆盖当前元素的位置而丢失a[i]）, 所以表头保存待排元素是必须的。找到后，在其后插入a[0]，完成排序。

源代码

/\*

带哨兵，数组a，0号单元不能存放待排序的元素

n是数组的长度

\*/

void InsertSort(int \*a, int n) {

if (n <= 1)

return;

int i, j;

for (i = 2; i <= n - 1; ++i) {

a[0] = a[i];

j = i - 1;

while (a[j] > a[0]) {

a[j + 1] = a[j];

--j;

}

a[j + 1] = a[0];

}

}

复杂度分析

n个元素需要往前扫描n - 1次，每次需要比较和后移（比较次数和后移次数相等），当原始数据大致有序时，比较和移动的次数最少，时间复杂度是O(n)。

当原始数据完全逆序时，每次排第i个元素时，往前扫描比较和移动的次数都是i - 1, 也就是

n\*((1 + 2 + 3 + ... + n) / n) = O(n ^ 2); 外循环n次乘以平均每次循环一次执行的内循环的次数。

或者，第1次循环执行1次，第2次2次，第n次n次，所以1 + 2 + 3 + ... + n;

假设第i次循环，往前扫描0.25\*i个元素，时间复杂度是，0.25\*(1 + 2 + 3 + ... + n), 所以平均复杂度是O(n ^ 2).

稳定性

稳定

最适情况

大致有序的原始序列

补充说明

适用于单链表，待排结点q，每次从头结点开始扫描找到第一个大于q->data的结点，插在它前面。该过程只比较，插入复杂度是O(1).

直接插入排序是基于比较，非一次性放置到最终位置的排序方法。

归并排序

算法流程

MergeSort功能是使[1...n]数组有序，使其有序的步骤是：

1，找到中间索引 mid = 1+(n-1)/2.

2，调用自己，使[1...mid]有序，使[mid...n]有序，然后合并两个有序数组。

3，此算法MergeSort包含一个子函数Merge，子函数Merge的功能是合并两个有序数组。

两个有序数组满足一个附加条件：[1..mid..r];

第一个数组是[1..mid],第二个数组[mid+1..r],也就是第一个数组的尾巴标号与第二个数组的开头标号连贯。

源代码

/\*二路归并排序，是经典的递归算法\*/

voidmergeSort(int\*arr,intlen){

if(len<=1)

return;

intmid=len/2;

mergeSort(arr,mid);

mergeSort(arr+mid,len-mid);

merge(arr,len,mid);

}

//mid是一个数组的分界线，两侧都是有序字数组，合并两个字数组仍保持有序

voidmerge(int\*arr,intlen,intmid){

if(arr==NULL||len<=0||mid>=len)

return;

int\*temp=(int\*)malloc(sizeof(int)\*len);

if(temp==NULL)

exit(-1);

inti=0;

intj=mid;

intk=0;

while(i<mid&&j<len){

if(arr[i]<arr[j]){

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

}

elseif(arr[i]>arr[j]){

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

}

else{

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

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

}

}

if(i>=mid){

for(;j<len;++j){

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

}

}

else{

for(;i<mid;++i){

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

}

}

for(intm=0;m<len;++m){

arr[m]=temp[m];

}

free(temp);

temp=NULL;

}

复杂度分析

利用主定理，时间复杂度是O(nlogn), 最差时间复杂度和最优时间复杂度都为 O(nlogn)。

**归并的空间复杂度就是那个临时的数组和递归时压入栈的数据占用的空间：n + logn；所以空间复杂度为: O(n)**

**稳定性**

**稳定**

**最适情况**

归并排序虽然比较稳定，在时间上也是非常有效的，但是这种算法很消耗空间，一般来说在内部排序不会用这种方法，而是用快速排序；外部排序才会考虑到使用这种方法。

补充说明

归并排序可应用于单链表

**题目**：单链表的归并排序，返回排序后的链表。传统的归并都是数组，可以随机访问元素，链表则需要顺序遍历找中间结点。

**思路**：

设置两个指针，一个步长为1， 一个步长为2，当快指针到达尾结点时，慢指针指向中间结点，时间复杂度为O(N)；

平分为左链表L1和右链表L2，递归分裂，直到链表为空或者只有一个结点；

将链表L2的每个结点插入到链表L1中，时间复杂度为O(m+n)，m、n分别为两条链表的长度。

画出递归树，可知总的时间复杂度为O(N \* lgN)。

快速排序

算法流程

挖出第一个元素作为空位，保存在temp，从后往前扫描到比temp更小的元素扔进上个空位，同时被扫到的位置变成了空位，再从前往后扫描，扫到比temp大的，扔进空位子；如此循环，直到两个扫描指针重叠，a[i] = temp;完成一次分离，i == j且指向基准且

基准左侧小于基准，右侧大于基准。

递归排序[left,i-1]和[i+1,right]

源代码

unsigned partition(int \*a, unsigned low, unsigned high) {

int temp;

temp = a[low];//挖空数组第一个位置,临时保存第一个元素的值

while (low < high) {//直到两个扫描指针重合

/\*先往后从前扫描\*/

while (low < high && a[high] >= temp) {

--high;

}

a[low] = a[high];

/\*再从前往后扫描\*/

while (low < high && a[low] <= temp) {

++low;

}

a[high] = a[low];

}

a[low] = temp;//放置第一个元素

return low;//返回第一个元素的最终位置

}

/\*快速排序\*/

void quickSort(int \*a, unsigned low, unsigned high) {

if (low >= high)

return;

unsigned i = low, j = high, result;

result = partition(a, i, j);

quickSort(a, low, result - 1);

quickSort(a, result + 1, high);

}

复杂度分析

平均时间复杂度：O(nlogn)

最坏时间复杂度：O(n^2)

最佳时间复杂度：O(nlogn)

辅助空间：O(logn)、归并排序的辅助空间是**O(n)。**

最坏情况：每次选择的基准都是区间的最小的元素，每次分区结束后，基准

的某一侧只有一个元素；对于不加优化的选择

第一个元素作为基准的快速排序，当区间元素是有序时，退化成插入排序

时间复杂度是O(n^2);同时会有最糟糕的辅助空间：由树深logn退成线性n

最佳情况：每次选择的基准都是区间的中位数，即排序后，基准两边的元素

数目相等或差1个。这时递归的深度也最浅，是logn；

算法描述：设置临时变量temp，使temp = a[1](第一个元素)

换句话说：挖空第一个，造一个空位子；

【注释：第一个空位必须是待排序的第一个元素位置】

优化

算法本质和形式不变，扔选择第一个元素作为第一个空位子。

只是为了避免待排元素是基本有序的，要尽量随机化的选择基准。

优化策略一：生成一各[left,right]内的随机数，该随机数指向的元素

与第一个元素交换值。

优化策略二：当递归到某个深度时，待分区的待排元素数目不大，终止递归；

换用擅长小数据排序的排序算法，如直接插入排序，冒泡排序等。

方法：递归结束条件改成right - left <= num

优化策略三：取最左，左右，中间三个元素的中间值放到第一个元素的位置；

这然既减小了最坏情况的发生，又能哨兵扫描，使时间减少一半。

优化策略四：设置一个num。当待排元素数目超过num，把元素分成n份，并行处理在

n个处理器上。

最适情况

越无序越好，大数据。

补充说明

一次性把元素放到最终位置。

选择排序

算法流程

[1,n]个元素，在n个元素中找个最小的元素的索引，通过交换填到1的位置；在[2,n]中找到最小元素的索引，通过交换填到1的位置……直到填好第n-1个位置。

怎么找？

设置常驻变量k，使其一直保存最小元素的索引；比较了趟尾，与

填充位置交换值。

源代码

void SelectSort(int \*a, int n) {

int k = n;

int i, j, temp;

for (i = 1; i <= n; ++i) {//填充位

for (j = n; j >= i; --j) {

if (a[j] <= a[k]) {

k = j;

}

}

temp = a[k];

a[k] = a[i];

a[i] = temp;

k = n;//别tm忘了！！

}

}

复杂度分析

时间复杂度：任何时候都是O(n^2)

辅助空间O(1)

补充说明

使用情况：无序的小数据时候优越于冒泡排序

选择排序和冒泡排序思路一模一样，区别是：

选择排序比较次数和冒泡一样，但是值只交换一次，省时间。

冒泡排序虽然交换次数多浪费时间，【但是对于基本有序的数组，

冒泡排序也很少交换+可以设置标志位提前结束，但是选择排序还傻傻

的O(n^2)比较。】

根据情况选择冒泡排序还是选择排序。

最大堆排序

算法流程

源代码

void HeapAdjust(int \*a, int i, int n) {

/\*i是待调节的结点编号，n是最后一个元素的编号\*/

int maxchild, temp;

/\*maxchild指向左右孩子较大的那个，temp只是用来swap\*/

while (2 \* i <= n) {//到树底（没有孩子的结点）结束

maxchild = 2 \* i;//指向左子树

if (maxchild <n && a[maxchild + 1] > a[maxchild])

++maxchild;//如果有右孩子，且右孩子较大，改变maxchild指向

if (a[i] < a[maxchild]) {//最大的孩子大于树，就交换

temp = a[i];

a[i] = a[maxchild];

a[maxchild] = temp;

i = maxchild;//哪个孩子被交换了，就破坏了它的平衡，

}//但不影响其他结点的平衡。

else break;//循环到树底结束或到孩子和根没发生交换时结束

}

}

/\*从最后一个有孩子的结点向上逐渐构建最大堆

HeapAdjust只适用左右子树都已是最大堆的根结点\*/

void BuildHeap(int \*a, int n) {

int i = (n / 2);

while (i >= 1) {

HeapAdjust(a, i, n);

--i;

}

}

/\*采用了递归，二次复习时一定要写一下非递归

void HeapSort(int \*a, int n) {

if (1 == n)

return;

int temp, i;

BuildHeap(a, n);

/\*把最大的数a[1]放到最后一个单元\*/

temp = a[1];

a[1] = a[n];

a[n] = temp;

HeapSort(a, n - 1);

}

复杂度分析

时间复杂度：O(nlogn)

时间复杂度的来源，重建n-1次堆，挑最大的填到a[n],

重建小于完全二叉树的深度，是logn,重建n-1次，故O(nlogn)

代码是最大堆排序，根节点编号为1

初始条件：i号结点的左子树和右子树都是最大堆树，

i和左右孩子不满足最大堆关系。

功能实现：使i和其左右孩子满足最大堆关系，i的

左右子树扔为最大堆树。

希尔排序

/\*希尔排序的步长可以根据特定公式选择最优的步长，加速排序\*/

Q：直接插入排序和希尔排序性能比较。

A：直接插入排序是稳定的；而希尔排序是不稳定的。

直接插入排序更适合于原始记录基本有序的集合。

希尔排序的比较次数和移动次数都要比直接插入排序少，当N越大时，效果越明显。

在希尔排序中，增量序列gap的取法必须满足：最后一个步长必须是 1 。

直接插入排序也适用于链式存储结构；希尔排序不适用于链式结构。

插入排序的本质是，通过比较，找到待排序元素最终插入位置，插入。

void Swap(int &a, int &b) {

int temp;

temp = a;

a = b;

b = temp;

}

void shellsort(int a[], int n)

{

int i, j, gap;

for (gap = n / 2; gap > 0; gap /= 2)

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

for (j = i - gap; j >= 0 && a[j] > a[j + gap]; j -= gap)

Swap(a[j], a[j + gap]);

}

时间复杂度是O(n^1.25).

冒泡排序

1.平均时间复杂度：O(n^2)

2.最坏时间复杂度：O(n^2)

3.最好时间复杂度：O(n)

4.辅助空间O(1)

最坏情况：无序，进行n-1趟排序，每趟排序都频繁交换值

最好情况：基本有序，常数趟排序，每趟只交换少许次值

适用情况：少量数据，基本有序数据

（1）从前往后扫描，每次扫描到尾，即找出一个最大或最小元素放置在表尾；

（2）继续重复（1）过程，直到扫出来n-1个大数，显然，剩余的最后一个

元素是最小的咯。

n个待排序元素，需n-1趟排序 i = 1; i < n

每次扫出来一个大的放到表尾，那是该数据的终态位置，下次扫描不在扫它，

所以每趟扫描都会少扫一个元素： j = 0;j < n - i;

改良：n-1 趟排序，可能在第m趟（m < n)线性表已有序，剩下的n-m-1趟都是

徒劳的扫描，应当提前终止。

若m趟前已经有序，m趟扫描必然不会交换紧邻两个位置的元素。若某趟没发生交换

即说明线性表已经有序。

解决方法：设置标志位，与交换代码绑一起，交换就改变标志位。

void BubbleSort(int \*a, int n) {

if (n <= 0)

return;

int i, j;//控制两层循环

int temp;//交换两个变量值的临时变量

int flag;//标志位，某一趟没发生交换则排序提前完成

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

flag = 0;

for (j = 0; j <= i; ++j) {

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

temp = a[j];

a[j] = a[j + 1];

a[j + 1] = temp;

flag = 1;

}

}

if (0 == flag)

return;

}

}

基数排序

算法流程

从低位到高位，逐位判断。二维数组相当于10个桶[0,1,2...9]，相应位是啥放进对应桶内，放完数组中的所有数后，再从0-9号桶中取出依次放进原数组。

源代码

#define BUCKETSIZE 10

/\*获取数字的位数\*/

int GetLoopTimes(int num) {

int count = 0;

while (num) {

num /= 10;

++count;

}

return count;

}

/\*查询数组中最大的数\*/

int SearchMax(int \*a, int n) {

int k = 0;

int i;

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

if (a[i] > a[k])

k = i;

}

return a[k];

}

/\*针对第loop位入桶排序再取出\*/

bool LoopSort(int \*a, int n, int loop) {

int i, j, row\_index;

int buckets[10][BUCKETSIZE] = {};

int tmpsum = (int)(pow(10, loop - 1));

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

row\_index = (a[i] / tmpsum) % 10;

for (j = 0; j < BUCKETSIZE && buckets[row\_index][j] != NULL; ++j)

;

if (j == BUCKETSIZE)

return false;

buckets[row\_index][j] = a[i];

}

/\*从桶中取出数据恢复\*/

j = 0;

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

while (buckets[i][j] != NULL) {

\*a++ = buckets[i][j];

++j;

}

j = 0;

}

}

bool BucketSort(int \*a, int n) {

int looptimes = GetLoopTimes(SearchMax(a, n));

int i, status;

for (i = 1; i <= looptimes; ++i) {

status = LoopSort(a, n, i);

if(status == 0)

return false;

}

return true;

}

时间复杂度

O (nlog(r)m)，r是进制，m是桶数。

哈希排序

待排元素的所有关键字已知,将关键字用数组有序存储,然后扫描元素,链到对应关键字数组单元.

时间复杂度是O(n),空间复杂度O(n).

例如,按年龄排序10万员工,关键字介于[0,99],分配整型数组age[100],扫描所有员工.