八种排序算法总结之 C++版本

五种简单排序算法

一、 冒泡排序 【稳定的】

```
void BubbleSort( int* a,int Count )  //实现从小到大的最终结果
{
    int temp;
    for(int i=1; i<Count; i++)  //外层每循环一次，将最小的一个移动到最前面
        for(int j=Count-1; j>=i; j--)
            if( a[j] < a[j-1] )
            {
                temp = a[j];
                a[j] = a[j-1];
                a[j-1] = temp;
            }
}
```

现在注意，我们给出 O 方法的定义：
若存在一常量 K 和起点 n0，使当 n>=n0 时，有 f(n)<=K*g(n),则 f(n) = O(g(n))。（呵呵，不要说没学好数学呀，对于编程数学是非常重要的!!!）
现在我们来看 1/2*(n-1)*n，当 K=1/2，n0=1，g(n)=n*n 时，1/2*(n-1)*n<=1/2*n*n=K*g(n)。所以 f(n) =O(g(n))=O(n*n)。所以我们程序循环的复杂度为 **O(n*n)**。

二、 交换排序 【稳定的】

```
void ExchangeSort( int *a,int Count)
{
    int temp;
    for(int i=0; i<Count-1; i++)
        for(int j=i+1; j<Count; j++)
            if( a[j] < a[i] )
            {
                temp = a[j];
                a[j] = a[i];
                a[i] = temp;
            }
}
```

时间复杂度为 **O(n*n)**。

三、 选择法 【不稳定的】

```
void SelectSort( int *a,int Count)
```

```
{
    int temp; //一个存储值
    int pos;  //一个存储下标
    for(int i=0; i<Count; i++)
    {
        temp = a[i];
        pos  = i;
        for(int j=i+1; j<Count; j++)
            if( a[j] < temp ) //选择排序法就是用第一个元素与最小的元素交换
            {
                temp = a[j];
                pos  = j;  //下标的交换赋值，记录当前最小元素的下标位置
            }
        a[pos] = a[i];
        a[i] = temp;
    }
}
```

遗憾的是算法需要的循环次数依然是 1/2*(n-1)*n。所以算法复杂度为 O(n*n)。
我们来看他的交换。由于每次外层循环只产生一次交换（只有一个最小值）。所以 f(n)<=n
所以我们有 f(n)=O(n)。所以，在数据较乱的时候，可以减少一定的交换次数。

四、　插入法　【稳定的】

```
void InsertSort( int *a,int Count)
{
    int temp; //一个存储值
    int pos;  //一个存储下标
    for(int i=1; i<Count; i++)  //最多做n-1趟插入
    {
        temp = a[i];    //当前要插入的元素
        pos  = i-1;
        while( pos>=0 && temp<a[pos] )
        {
            a[pos+1] = a[pos];  //将前一个元素后移一位
            pos--;
        }
        a[pos+1] = temp;
    }
}
```

其复杂度仍为 O(n*n)。
最终，我个人认为，在简单排序算法中，直接插入排序是最好的。

五、　希尔排序法　【不稳定的】

```
/*
*   希尔排序，n为数组的个数
*/
void ShellSort( int arr[], int n )
{
    int temp,pos;
    int d = n;      //增量初值
    do{
        d = d/3 + 1 ;
        for(int i= d; i<n; i++ )
        {
            temp = arr[i];
            pos = i-d;
            while( pos>=0 && temp < arr[pos] ) {    //实现增量为d的插入排序
                arr[ pos + d ] = arr[pos];
                pos -= d;
            }
            arr[ pos + d ] = temp;
        }
    } while( d > 1 );
}
```

三种高级排序算法

一、　快速排序　辅助空间复杂度为 O（1）　【不稳定的】

```
void QuickSort( int *a,int left, int right)
{
    int i,j,middle,temp;
    i = left;
    j = right;
    middle = a[ (left+right)/2 ];
    do
    {
        while( a[i]<middle && i<right ) //从左扫描大于中值的数
            i++;
        while( a[j]>middle && j>left ) //从右扫描小于中值的数
            j--;
        if( i<=j )  //找到了一对值
        {
            temp = a[i];
            a[i] = a[j];
```

```
            a[j] = temp;
            i++;
            j--;
        }

    } while ( i<j ); //如果两边的下标交错，就停止(完成一次)
    //当左半边有值（left<j）,递归左半边
    if( left < j )
        QuickSort( a, left, j);
    //当右半边有值（right>i）,递归右半边
    if( i < right )
        QuickSort( a, i, right);
}
```

这里我没有给出行为的分析，因为这个很简单，我们直接来分析算法：首先我们考虑最理想的情况
1.数组的大小是 2 的幂，这样分下去始终可以被 2 整除。假设为 2 的 k 次方，即 k=log2(n)。
2.每次我们选择的值刚好是中间值，这样，数组才可以被等分。
第一层递归，循环 n 次，第二层循环 2*(n/2).....
所以共有 n+2(n/2)+4(n/4)+...+n*(n/n) = n+n+n+...+n=k*n=log2(n)*n
所以算法复杂度为 O(log2(n)*n)
其他的情况只会比这种情况差，最差的情况是每次选择到的 middle 都是最小值或最大值，那么他将变成交换法（由于使用了递归，情况更糟），但是糟糕的情况只会持续一个流程，到下一个流程的时候就很可能已经避开了该中间的最大和最小值，因为数组下标变化了，于是中间值不在是那个最大或者最小值。但是你认为这种情况发生的几率有多大？？呵呵，你完全不必担心这个问题。实践证明，大多数的情况，快速排序总是最好的。
如果你担心这个问题，你可以使用堆排序，这是一种稳定的 O(log2(n)*n)算法，但是通常情况下速度要慢于快速排序（因为要重组堆）。

二、 归并排序（两种实现方法均要掌握） 【稳定的】

归并排序是一种极好的内部排序方法，即针对数据保存在磁盘上而不是高速内存中的问题。

//以下程序参考数据结构课本 P286 页的模板,为使用指针链表实现的

```
#include <iostream>
using namespace std;

struct node{        //链表的节点数据
    int value;
    node *next;
};
```

```
node * divide_from( node * head )
{
    node * position, * midpoint, * second_half;
    if( (midpoint=head) == NULL ) //List is empty
        return NULL;
    position = midpoint->next;
    while( position != NULL ) //Move position twice for midpoint's one move
    {
        position = position->next;
        if( position != NULL )
        {
            midpoint = midpoint->next;
            position = position->next;
        }
    }
    second_half =  midpoint->next;
    midpoint->next = NULL; //在这里将原链拆断，分为两段
    return second_half;
}

node * merge( node * first, node * second)
{
    node * last_sorted; //当前已经链接好的有序链中的最后一个节点
    node combined;      //哑节点
    last_sorted = &combined;
    while( first!=NULL && second!=NULL )
    {
        if( first->value < second->value ) {
            last_sorted->next = first;
            last_sorted = first;
            first = first->next;
        }else {
            last_sorted->next = second;
            last_sorted = second;
            second = second->next;
        }
    }
    if( first==NULL )
        last_sorted->next = second;
    else
        last_sorted->next = first;
    return combined.next; //返回哑节点的后继指针，即为合并后的链表的头指针
}
```

```
//这里的参数必须是引用调用，需要这个指引去允许函数修改调用自变量
void MergeSort( node * &head)
{
    if( head != NULL && head->next != NULL ) //如果只有一个元素，则不需排序
    {
        node * second_half = divide_from( head );
        MergeSort( head );
        MergeSort( second_half );
        head = merge( head, second_half );
    }
}

int main()
{
    node a,b,c,d;
    node *p1, *p2, *p3, *p4,*head;
    p1 = &a;
    p2 = &b;
    p3 = &c;
    p4 = &d;
    a.value = 2;
    b.value = 4;
    c.value = 3;
    d.value = 1;
    a.next = p2;
    b.next = p3;
    c.next = p4;
    d.next = NULL;
    //调用归并排序前的结果
    head = p1;
    while( head != NULL )
    {
        cout<<head->value<<" ";
        head = head->next;
    }
    cout<<endl;
    MergeSort( p1 );
    //调用归并排序后的结果
    head = p1;
    while( head != NULL )
    {
        cout<<head->value<<" ";
        head = head->next;
    }
```

```
    cout<<endl;
}
```

//以下程序为使用数组实现的归并排序,辅助空间复杂度为 O（n）

```
#include <iostream>
using namespace std;

void Merge( int data[], int left, int mid, int right )
{
    int n1,n2,k,i,j;
    n1 = mid - left + 1;
    n2 = right - mid;
    int *L = new int[n1];  //两个指针指向两个动态数组的首地址
    int *R = new int[n2];
    for( i=0,k=left; i<n1; i++,k++)
        L[i] = data[k];
    for( i=0,k=mid+1; i<n2; i++,k++)
        R[i] = data[k];
    for( k=left,i=0,j=0; i<n1 && j<n2; k++) {
        if( L[i] < R[j] ) { //取小者放前面
            data[k] = L[i];
            i++;
        } else {
            data[k] = R[j];
            j++;
        }
    }
    if( i<n1 )   //左边的数组尚未取尽
        for( j=i; j < n1; j++,k++)
            data[k] = L[j];
    else
    //if( j<n2 ) //右边的数组尚未取尽  ,这句话可要可不要
        for( i=j; i<n2; i++,k++)
            data[k] = R[i];
    delete []L;   //回收内存
    delete []R;
}
/*
*   left:数组的开始下标，一般为0；right：数组的结束下标，一般为 （n-1）
*/

void MergeSort( int data[], int left, int right )
{
```

```
    if( left < right )
    {
        int mid = left + ( right-left ) / 2; //mid=(right+left)/2, 防止溢出
        MergeSort( data, left, mid );
        MergeSort( data , mid+1, right );
        Merge( data , left, mid , right );
    }
}

int main()
{
    int data[] = {9,8,7,2,5,6,3,55,1};
    //排序前的输出
    for(int i=0; i<9; i++)
        cout<<data[i]<<" ";
    cout<<endl;
    MergeSort( data, 0, 8);
    //排序后的输出
    for(int i=0; i<9; i++)
        cout<<data[i]<<" ";
    cout<<endl;
}
```

三、 堆排序 【不稳定的】

```
/*
*   向堆中插入current元素的函数
*/
void insert_heap( int data[], const int &current, int low, int high )
{
    int large;  //元素data[low]左右儿子中，大者的位置
    large = 2*low + 1;
    while( large <= high ) {
        if( large < high && data[large] < data[ large+1] )
            large++;
        if( current > data[ large ] ) //待插入元素的值比它的两个儿子都大
            break;
        else {
            data[ low ] = data[ large ]; //将其左右儿子的大者上移
            low = large;
            large = 2 * large + 1;
        }
    }
```

```
    data[ low ] = current;
}
/*
*   建立堆函数，num为数组data的元素个数
*   只有一个结点的<2-树>自动满足堆的属性，因此不必担心树中的任何树叶，即
*   不必担心表的后一半中的元素。如果从表的中间点开始并从后向前工作，就
*   能够使用函数insert_heap去将每个元素插入到包含了所有后面元素的部分堆
*   中，从而创建完整的堆。
*/
void build_heap( int data[], int num )
{
    int current;
    for( int low = num/2 - 1; low>=0; low-- ) {
        current = data[ low ];
        insert_heap( data, current, low, num-1 );
    }
}
/*
*   堆排序主函数,num为数组data的元素个数
*/
void heap_sort( int data[], int num )
{
    int current, last_sorted;
    build_heap( data, num );    //建立堆
    for( last_sorted = num-1; last_sorted>0; last_sorted-- ) { //逐个元素处理
        current = data[ last_sorted ];
        //data[0]在整个数组排序结束前，存储的是待排序元素中最大的元素
        data[last_sorted] = data[0];
        insert_heap( data, current, 0, last_sorted-1 );
    }
}
int main()
{
    //用于排序算法的输入输出
    int a[8] = {5,7,1,2,9,4,6,3,};
    for(int i=0; i< sizeof(a)/sizeof(int); i++)
        cout<<a[i]<<" ";
    cout<<endl;
    heap_sort( a, 8 );  //调用堆排序
    for(int i=0; i< sizeof(a)/sizeof(int); i++)
        cout<<a[i]<<" ";
    cout<<endl;
    return 0;
}
```