问题

排序算法是数据结构与算法中的基础，也是面试官经常考察的代码基础之一，所以想总结一下几种经典的排序算法原理，用一种方便自己理解的思路整理出来，并用C++代码实现（建议理解后把这几种排序算法代码默写一遍哈哈）。

排序算法

首先总结一下几种经典算法特点：

算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	稳定
希尔排序	/	/	/	$O(1)$	不稳定
堆排序	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(1)$	不稳定
快速排序	$O(nlogn)$	$O(nlogn)$	$O(n^2)$	$O(logn)$	不稳定
归并排序	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(n)$	稳定

补充： **稳定性**是指，排序前后两个相等的元素原始的相对位置不会改变；不稳定即会交换位置。

冒泡排序

算法原理：

①两两比较，先比较前两个元素，若第一个元素大于第二个元素，则交换位置，否则不交换，继续下一组比较

②按照上面的比较规则，继续比较第二、第三个元素，以此类推，直至最后两个元素比较完，则完成**一趟排序**。

③一趟排序后，最后一个元素变成最大(即已排序)，下一次排序中将不再参与排序。

④对**剩下未排序的元素**按照①-③的规则，继续新一轮的排序，重复此操作，直至所有元素完成排序。

算法分析：

最普通的冒泡排序， 需要进行n-1趟排序，第 i 趟排序要进行n-i次比较；

经过优化的冒泡排序，使用一个标志flag来记录本趟排序中是否发生交换，若没有，说明所有元素已全部有序。此时，最好的情况是所有元素已经有序，只需进行一趟排序（n-1次比较，0次交换），时间复杂度为$O(n)$；最坏的情况是刚好倒序的情况，需要进行n-1趟排序（第 i 趟排序要进行n-i次比较、n-i次交换），时间复杂度为$O(n^2)$。

稳定性：冒泡排序在两个元素相等时不会交换位置，因此是**冒泡排序是稳定的**。

C++代码：

```
void bubble_sort(vector<int>& nums) {
    int size = nums.size();
    if(size <= 1) return;
    bool flag = true;
    for(int i = 0; i < size - 1 && flag; i++) {
        flag = false;
        for(int j = size - 2; j >= i; j--) {
            if(nums[j] > nums[j+1]) {
                swap(nums[j],nums[j+1]);
                flag = true;
            }
        }
    }
}
```

选择排序

算法原理：

①第一趟排序，先从未排序序列中找出最小（大）元素，放到序列的起始位置

②接着，在剩下未排序的序列中找出最小（大）元素，放到已排序序列的末尾

③重复②，直至所有元素排序完成

算法分析：

无论序列是否有序，选择排序都要进行n-1趟排序，第 i 趟排序要进行n-i次比较，因此最好最坏的情况时间复杂度都是$O(n^2)$，这也说明**选择排序的性能并不受序列本身的顺序所影响**。

稳定性：**选择排序是不稳定的**。举个例子，序列[4,5,6,4,3,2]，一趟排序后变成[2,5,6,4,3,4]，排序前后两个4的相对位置发生了变化，所以是不稳定的。

C++代码：

```
//选择排序（由小到大排序）
void select_sort(vector<int>& nums) {
    int size = nums.size();
    if(size <= 1) return;
    for(int i = 0; i < size - 1; i++) {
        int min = i;    // 假设当前索引下的数最小
        for(int j = i + 1; j < size; j++) {
            if(nums[min] > nums[j]) min = j;
        }
        if(i != min) swap(nums[i], nums[min]);
    }
}
```

插入排序

算法原理：

①首先，将待排序序列(用A表示)中第一个元素当作有序序列B，剩下的当作未排序序列C

②每趟排序都是将**未排序序列C的第一个元素c**与有序序列B的元素**(由后向前)**依次比较，若 c 小于B中某个元素，则插入到B中该元素前面，否则插入到后面，至此完成**一趟排序**。

③重复②操作，即可完成整个序列的排序。

算法分析：

可见，最好的情况就是序列本身已经是有序时，只需进行n-1次比较并插入即可，时间复杂度为O(n)；最坏得情况是序列本身是倒序，需要进行$\frac{n(n-1)}{2}$次比较，时间复杂度为$O(n^2)$。

稳定性：插入排序只会将较小的元素插入到某个元素的前面，当两个元素相等时不会改变原来的位置，因此**插入排序是稳定的**。

C++代码：

```
//插入排序
vector<int> insertSort(vector<int> &arr){
    for(int i=1;i<arr.size();++i){
        int preIndx = i-1;
        int currentVal = arr[i];  //每次取出未排序序列的第一个元素
        while(preIndx>=0 && currentVal<arr[preIndx]){
            arr[preIndx+1] = arr[preIndx];  //比前面的值小，将前面的值后移
            --preIndx;
        }
        arr[preIndx+1] = currentVal;  //比前一个值大，在后面插入
    }
    return arr;
}
```

希尔排序

由前面分析可知，插入排序在序列本身就是有序的时候时间复杂度为O(n)，此时算法表现很好。但实际中，序列本身已经有序的情况很少，另外当数据量较大时，直接插入排序比较和交换的次数会很多，效率变得很低下。

希尔排序，也可以理解为增量递减排序，其实是在直接插入排序的基础上做了一些改进，使其更加高效。**其主要思想就是，先进行一些处理，让整个序列变得基本有序，再进行插入排序即可。**

算法原理：

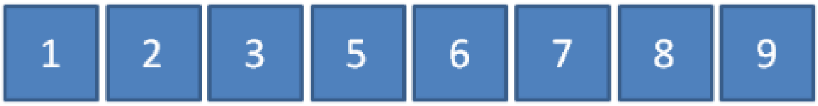
①选择一种递减的增量序列，如每次折半的增量序列：...，8，4，2，1

②先按照初始增量(如step=8)将整个待排序序列分成若干组，对每组分别进行插入排序

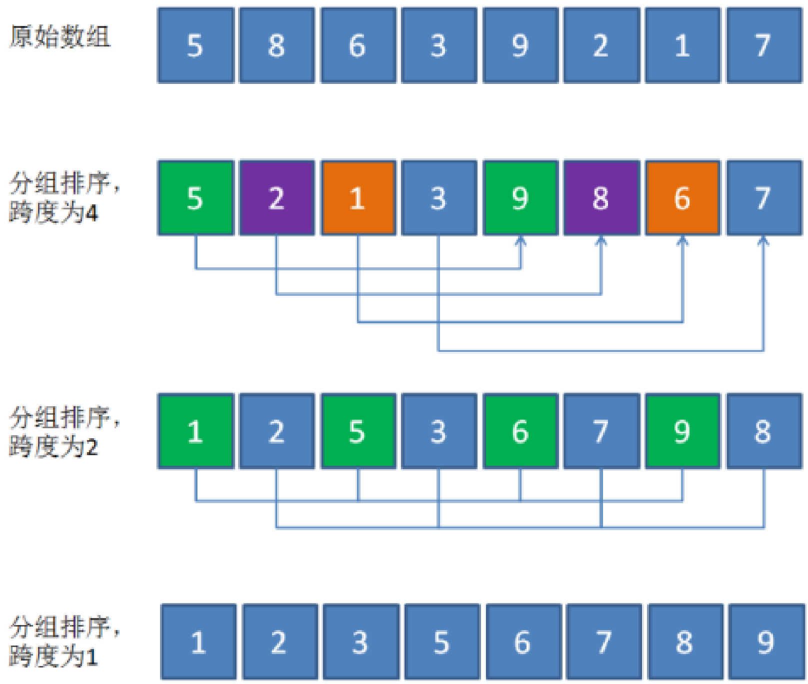
③递减增量(step=step/2)，重复②-③

④直至整个序列基本有序(如step=1)，然后对整个序列进行插入排序

（借助下面这张图来直观地理解一下，图片截自这篇博客）



让我们重新梳理一下分组排序的整个过程：



补充理解：

1.举个例子，序列长度len=8，初始增量为step=len/2=4，则将序列分成了4组每组两个元素，然后每组的两个元素先分别进行插入排序，然后再将增量改为step=2，序列分成两组每组4个元素，然后继续操作下去......字后增量step=1，也就是整个序列为一组，直接进行插入排序即可。

2.事实上，增量序列不仅可以是[1,2,4,8,...]，还可以是别的形式，如[1,3,7,15,...]，即2^n-1的形式，还可以是[1,5,19,41,109,...]等，不同形式时间复杂度亦有不同

算法分析：

希尔排序根据增量序列选取的不同，其时间复杂度亦有所不同，增量序列为[1,2,4,8,...]时最坏的时间复杂度为$O(n^2)$，增量序列为[1,3,7,15,...]时最坏为$O(n^{1.5})$，增量序列为[1,5,19,41,109,...]时最坏为$O(n^{1.3})$。

稳定性：希尔排序中，相等的数在不同的分组中时，排序后可能会改变其相对位置，如[3,6,5,5]在增量为2时，排序后为[3,5,5,6]，两个5的相对位置发生了改变，所以**希尔排序是不稳定的**。

C++代码：

```
//希尔排序（增量序列以2的倍数递减，如4，2，1）
vector<int> shellSort(vector<int> &arr){
    int len = arr.size();
    for(int step=len/2;step>0;step/=2){   //比插入排序多了一层循环，表示不同增量下的插入排序
        //内层for循环与step=1的插入排序代码一模一样，可以理解为增量step将整个序列分成了若干组
        //在当前增量下，每step次++i的过程就是对各组数据轮流进行一次插入排序，
        //而不是一组全部排序完再对另一组排序
        for(int i=step;i<len;++i){
            int preIndx = i - step;
            int currentVal = arr[i];
```

```
            while(preIndx>=0 && currentVal<arr[preIndx]){
                arr[preIndx+step] = arr[preIndx];
                preIndx -= step;
            }
            arr[preIndx+step] = currentVal;
        }
    }
    return arr;
}
```

堆排序

一些概念：

大顶堆：`arr[i] >= arr[i\*2+1] && arr[i] >= arr[i\*2+2]`　// 这里默认从数组的0坐标开始

小顶堆：`arr[i] <= arr[i\*2+1] && arr[i] <= arr[i\*2+2]`

升序排序：采用大顶堆　（因为堆排序中有一个将堆顶元素与最后一个元素交换的过程，即把最大元素放在最后）

降序排序：采用小顶堆　（同理）

算法原理：

①创建一个堆（大顶堆或小顶堆）

②交换堆顶元素与序列最后一个元素（可以理解为一趟排序结束）

③重新调整除了最后一个元素之外的序列，使其满足堆的性质

④重复②-③，直至当前堆的长度为1，即完成所有元素的排序

具体思路：

（以下没有图解的思路可能只适合个人理解，具体思路建议参考这篇博客：[图解排序算法（三）之堆排序]）

1. **怎么创建一个堆？**（以大顶堆为例）

   堆可以理解为一种完全二叉树，用数组下标来表示父节点与孩子节点的关系就是，父节点为arr[i]的左右孩子分别为`arr[2\*i+1]`和`arr[2\*i+2]`。另外，若堆的大小（即数组长度）为len，则最后一个叶子节点为`arr[len/2-1]`。了解了这些以后，就可以用无序序列开始创建堆了：

   首先，从最后一个非叶子结点开始，自底向上调整元素。怎么调整？主要的思想就是在每个非叶子节点上，都把其左右孩子中最大的孩子取出来，并让它往上移动，这也是为什么要自底向上构建的原因，先把最底层的元素中较大的往上移，再把倒数第二层较大的元素往上移，逐层向上，就可以保证每个父节点的值都大于其孩子节点的值，也就满足了大顶堆的性质。

2. **如何调整堆？**（在已经构建好大堆顶的基础上调整）

   重新调整堆之前，需先将前一次调整好的堆顶元素与序列最后一个元素交换（相当于该堆顶元素已排序，接下来只需将剩下的无序序列重新调整成新的堆，重复操作即可）。接下来自顶向下地调整：

   ①**先保存当前根节点值temp**，然后自顶向下地进行以下操作：

   ②选出左右子节点中较大的节点，与父节点比较，若父节点小，则将较大的子节点放在父节点的位置，但不是交换（相当于每向下一层都试图把较大的孩子对应的元素往上移，当不能继续向下时，就把之前保存的根节点值temp填补在最终位置）

算法分析：

创建初始堆的时间复杂度为$O(n)$。具体推导可以参考[排序算法之 堆排序 及其时间复杂度和空间复杂度]这篇博客。

> **推算过程：**
>
> **首先要理解怎么计算这个堆化过程所消耗的时间，可以直接画图去理解；**
>
> **假设高度为k，则从倒数第二层右边的节点开始，这一层的节点都要执行子节点比较然后交换（如果顺序是对的就不用交换）；倒数第三层呢，则会选择其子节点进行比较和交换，如果没交换就可以不用再执行下去了。如果交换了，那么又要选择一支子树进行比较和交换；**
>
> **那么总的时间计算为：s = 2^(i - 1) * (k - i)；其中 i 表示第几层，2^(i - 1) 表示该层上有多少个元素，(k - i) 表示子树上要比较的次数，如果在最差的条件下，就是比较次数后还要交换；因为这个是常数，所以提出来后可以忽略；**
>
> **S = 2^(k-2) * 1 + 2^(k-3)*2.....+2*(k-2)+2^(0)*(k-1) ===> 因为叶子层不用交换，所以i从 k-1 开始到 1；**
>
> **这个等式求解，我想高中已经会了：等式左右乘上2，然后和原来的等式相减，就变成了：**
>
> **S = 2^(k - 1) + 2^(k - 2) + 2^(k - 3) ..... + 2 - (k-1)**
>
> **除最后一项外，就是一个等比数列了，直接用求和公式：S = { a1[1- (q^n)] } / (1-q)；**
>
> **S = 2^k -k -1；又因为k为完全二叉树的深度，所以 (2^k) <= n < (2^k -1)，总之可以认为：k = logn （实际计算得到应该是 log(n+1) < k <= logn ）；**
>
> **综上所述得到：S = n - longn -1，所以时间复杂度为：O(n)**

堆排序整个过程中要进行n-1次调整堆，每次都是自顶向下，根据二叉树的性质可知每次调整的时间复杂度为$O(logn)$，因此所有调整过程时间复杂度为$O(nlogn)$。

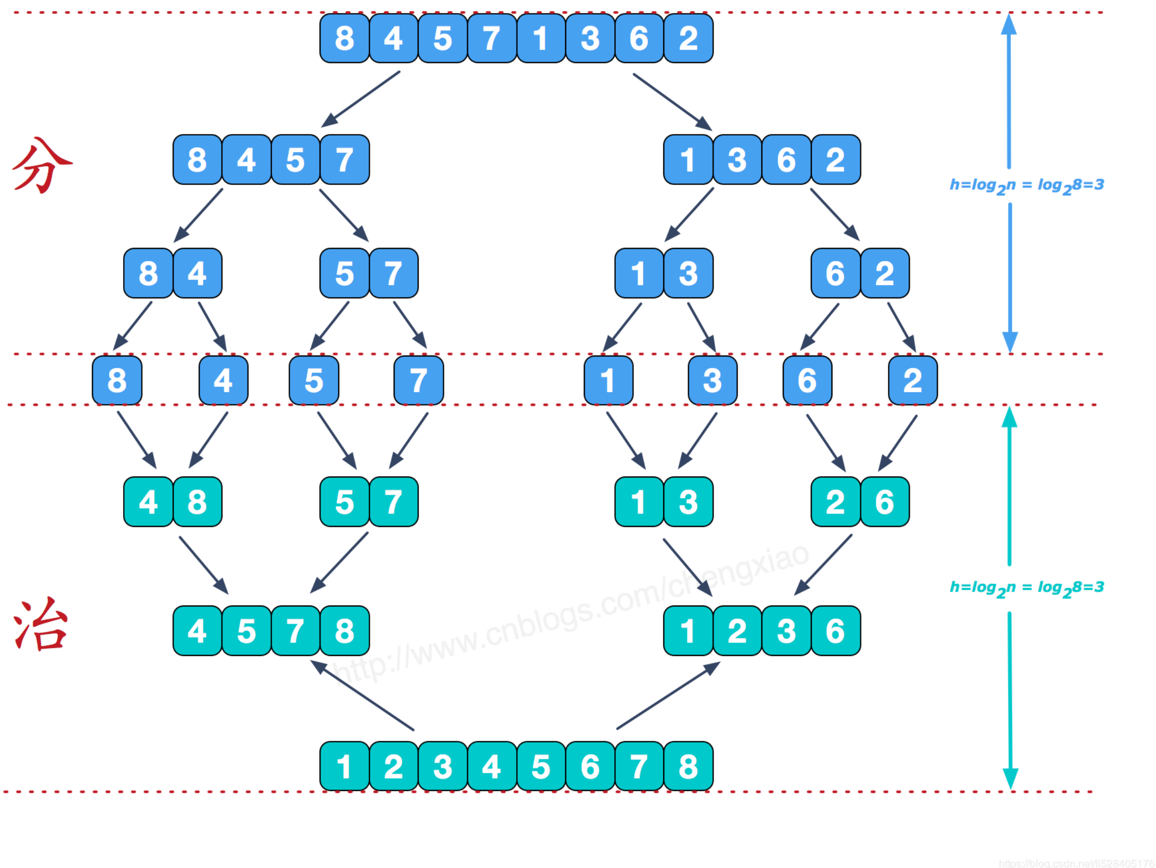
因此堆排序的时间复杂度为$O(nlogn)$，最好和最坏情况也都是如此。

C++代码：

```
//调整堆(排序结束后为升序--采用大顶堆)
void adjustHeap(vector<int> &arr,int start,int end){
    int temp = arr[start];  //temp创建堆时为非叶子结点，重新调整堆时就是堆顶元素(根节点)
    for(int i=2*start+1;i<=end;i=2*i+1){
        if(i<end && arr[i]<arr[i+1]) ++i;   //从左右孩子中选出最大的
        if(temp<arr[i]){  //若最大孩子比父节点大，则将其值赋予父节点，但不交换
            arr[start] = arr[i];
            start = i;
        }
        else break;
    }
    arr[start] = temp;  //将原来根节点的值放到最大孩子的位置
}
//堆排序
void heapSort(vector<int> &arr){   // 这里的len是数组的长度
    //创建大顶堆（从最后一个非叶子结点开始自底向上调整）
    int len = arr.size();
    for(int i=len/2-1;i>=0;i--) adjustHeap(arr,i,len-1); // 注意这里是 len - 1
    //交换堆顶元素和最后一个元素后，重新调整剩余堆元素（自顶向下），排序结束后为升序
    for(int j=len-1;j>=0;j--){
        swap(arr[0],arr[j]);
        adjustHeap(arr,0,j-1);
    }
}
```

归并排序

归并排序运用了分而治之的思想，可以使用自顶向下递归的方法来处理。借助一张图来更好地理解归并的思想，图片引自[这篇博客](#)。



算法原理：

①（自顶向下二分）将待排序序列二分成两部分，分别对两部分进行排序，再将其合并

②当然，对每个子部分，都可以递归处理，继续二分成两部分，直至每个部分只有一个元素的时候不可再分

③（自底向上合并）此时**需要借助一个与待排序序列大小相同的额外空间，来存放合并后的序列**。

④两个序列的合并时，可以使用两个指针 i 和 j 分别指向两个序列的起始位置，比较对应元素的大小，将较小的元素（假设是 i 指向的元素）存放到额外的数组中，然后将其指针后移（即 i++）

⑤重复④，直至其中一个序列全部合并完，然后将另一个序列直接复制到已排序序列尾部，即完成排序。

算法分析：

归并排序算法的平均时间复杂度为$O(nlogn)$，和快速排序一样都是利用了递归的方法，递归的时间复杂度公式为$T(n) = 2T(n/2) + f(n)$。不同的是，归并排序最好和最坏的情况时间复杂度都是$O(nlogn)$，因此**归并排序的性能也不受序列本身顺序的影响**。

C++代码：

```
//归并操作（治）
```

```cpp
void merge(vector<int>& nums, int left, int mid, int right) {
    int l = left;
    int r = mid + 1;
    int end = right;
    vector<int> temp;  // 用于存储该段的排序结果
    while(l <= mid && r <= end) {
        if(nums[l] <= nums[r]) {
            temp.emplace_back(nums[l]);
            l++;
        }
        else {
            temp.emplace_back(nums[r]);
            r++;
        }
    }
    while(l <= mid) {
        temp.emplace_back(nums[l]);
        l++;
    }
    while(r <= end) {
        temp.emplace_back(nums[r]);
        r++;
    }
    int j = 0;
    for(int i = left; i <= right; i++) {
        nums[i] = temp[j++];
    }
    return;
}

// 归并排序（分）
void gb_sort(vector<int>& nums, int left, int right) {
    if(left >= right) return;  // 这里这里的等号
    int mid = left + ((right - left) >> 1);
    gb_sort(nums, left, mid);
    gb_sort(nums, mid+1, right);
    merge(nums, left, mid, right);
}
```

快速排序

相比于前面几种排序算法，快速排序稍微难理解一些，**主要思想**是：选定基准数，每趟排序后，其左边的所有数都比它小，右边的都比它大，再用递归的方法对基准数的左右两个区间进行快速排序

算法原理：

①在待排序序列中选取一个基准数，如第一个元素。例如在[5,2,6,7,1,4,3,8]中选取5作为基准数

②**先从右往左**找到第一个小于基准的数，**再从左往右**找到第一个大于基准的数，然后交换它们，这里可以设置首尾两个指针 i 和 j 来实现。（注意这里顺序不能颠倒，为什么呢？可以先理解了整个思想之后再来思考这个问题）

③然后 j 继续左移，i 继续右移，分别找到小于、大于基准的数，然后交换，直至两个指针相遇，则将该处的值与基准数交换，至此，完成了**一趟快速排序**。(为什么相遇后要交换？)

④接着，分别递归排序基准数左区间和右区间，递归终止条件是 i、j 两个指针相遇，当待排序序列不能再划分时整个快速排序结束。

算法分析：

快速排序的实现利用了递归的思想，递归算法的时间复杂度公式为$T(n) = aT(n/b) + f(n)$，因此，应用在快速排序中最优的情况就是每次都是将序列平分再递归，即时间复杂度公式为$T(n) = 2T(n/2) + f(n)$，经过推算后可得最好情况下的时间复杂度为$O(nlogn)$。最坏的情况就是每次选取的基准数刚好是最小（最大）的，则每次只能排好一个元素，时间复杂度为$O(n^2)$。

此处参考博客：[快速排序 及其时间复杂度和空间复杂度]

C++代码：

```
//快速排序（由小到大排序）
void quickSort(vector<int> &arr, int left, int right){
    if(left>=right) return;
    int i, j;
    i = left; j = right;
    int base = arr[left];  //选取序列第一个元素作为基准数
    while(i<j){
        while(base <= arr[j] && i<j) --j;  //左移找到第一个小于基准数的数
        while(base >= arr[i] && i<j) ++i;  //右移找到第一个大于基准数的数
        if(i<j) swap(arr[i], arr[j]);      //左右指针相遇前交换
    }

    arr[left] = arr[i];  //两个指针相遇后，将其所对应的元素与基准数交换
    arr[i] = base;
    quickSort(arr, left, i-1);  //递归排序小于基准数的部分
    quickSort(arr, i+1, right);  //递归排序大于基准数的部分
}
```

补充：现在可以解决上面留下的两个问题啦~

1.为什么选择第一个元素为基准数后，指针 j 要先从右往左移，而不能先从左到右移动指针 i？

因为如果先右移指针 i，就会使得当 i 和 j 相遇时，指针指向的元素比基准数大，再与基准数交换之后，就不能满足基准数左边所有元素均比它小的条件了。既然这样，不与交换基准数不就好了吗？当然不行，不交换，基准数就会一直不动，显然排序是不正确的。那么，如果选择最右边的元素作为基准数可以吗？可以，不过指针要先从左向右移动指针 i，再从右到左移动指针 j 即可。

2.两个指针相遇后为什么要将该元素与基准数交换？

因为指针 i、j 相遇后所指的元素必定比基准数小，交换的结果是使得基准数左边的所有元素均比它小，右边的所有元素均比它大，**也就是说，一趟快速排序其实就是将选取的基准数放到本该属于它的位置。**

拓展

快速排序和归并排序哪个更优？

快速排序和归并排序时间复杂度都是$O(nlogn)$，而最坏情况下快排的时间复杂度为$O(n^2)$，归并排序最坏的情况仍然只有$O(nlogn)$，是否就能说明归并排序一定更快？不能。事实上，对不规则的数据，当数据量越大的时候，快速排序的速度**可能会更快**（注意是可能）。这里列举一些快速排序更快的说法：

①一个说法是，快速排序最坏的时间复杂度为$O(n^2)$，而平均时间复杂度为$O(nlogn)$，因此有很多时候时间复杂度小于$O(nlogn)$，比归并排序始终为$O(nlogn)$更快。

②另一个说法是，当数据量很大时，归并排序在合并的过程中访问读写数组的次数更多，速度会变慢。