排序算法是最基本的算法之一。排序即对一系列对象根据某个关键字进行排序。首先介绍一下排序术语：

稳定：a=b，排序前 a 在 b 之前，排序后 a 仍然在 b 之前
不稳定：a=b，排序前 a 在 b 之前，排序后 a 可能在 b 之后
内排序：所有排序操作都可以在内存中完成
外排序：当数据太大时需要将数据放在磁盘中
时间复杂度：一个算法执行所耗费时间的估计值
空间复杂度：一个算法执行所耗费空间的估计值

排序算法分类如下：

根据排序方式，排序算法整体上可以划分为内部排序和外部排序。内部排序是指数据记录在内存中进行排序，不需要申请额外的存储空间。外部排序是指在排序过程中需要申请额外的内存空间。

排序方式	排序算法	最好情况	最坏情况	平均时间复杂度	空间复杂度	稳定性
In-place	选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
	冒泡排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
	插入排序	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	稳定
	快速排序	$O(n \log n)$	$O(n^2)$	$O(n \log n)$	$O(\log n)$	不稳定
	希尔排序	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(n \log n)$	$O(1)$	不稳定
	堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	不稳定
Out-place	归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	稳定
	计数排序	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(k)$	稳定
	桶排序	$O(n+k)$	$O(n^2)$	$O(n+k)$	$O(n+k)$	稳定
	基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n+k)$	稳定

（注意：n 指的是数据规模；k 指的是“桶”的个数；In-place 指的是占用常数内存，不占用额外内存；Out-place 指的是占用额外内存）

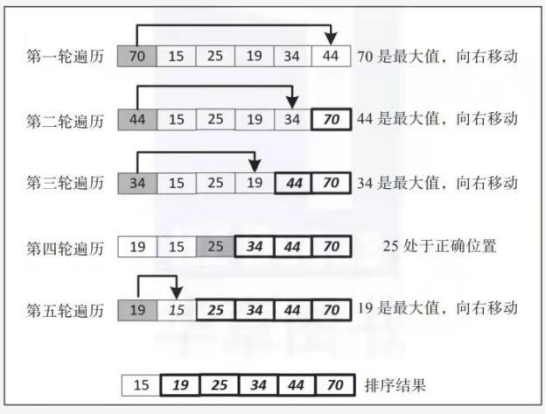
本章节接下来的测试代码可以在 Leetcode 912 题上编译运行，其中希尔排序（三分之一增量版本）、堆排序、归并排序不会超时。

一、内排序

1、选择排序

选择排序（selection-sort）是性能鲁棒性最好的排序算法之一，任何情况下耗费时间都为$O(n^2)$，而且不占用额外内存，适合于数据规模 n 较小的情景。

【算法思路】：从未排序序列中找到最小（大）元素，存放到排序序列的末尾位置，直至未排序序列为空。



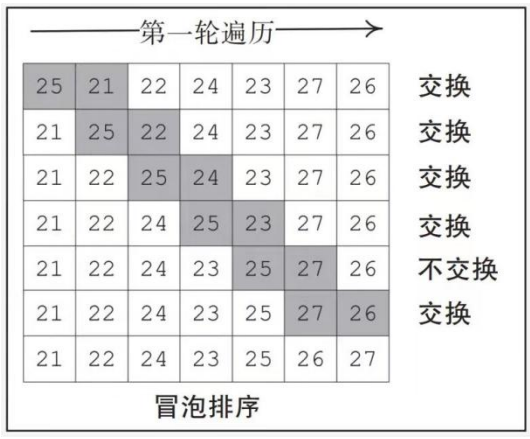
【编码思路】：双层循环，外层循环负责从左至右确定已排序序列，内层循环负责从左至右在未排序序列中查找最值，找到后交换两数即可。

```
/**选择排序**/
void selectSort(vector<int> &nums) {
    for(int i=0;i<nums.size();i++){
        // 查找未排序序列中的最值
        int minIndex = i;
        for(int j=i+1;j<nums.size();j++){
            if(nums[minIndex]>nums[j]){
                minIndex=j;
            }
        }
        // 交换nums[i]和nums[minIndex]
        if(minIndex!=i){
            int temp=nums[minIndex];
            nums[minIndex]=nums[i];
            nums[i]=temp;
        }
    }
}
```

2、冒泡排序

冒泡排序（bubble-sort）比选择排序性能稍好一定，而且稳定，排序不影响相同数的位置，平均时间复杂度为$O(n^2)$，同样不占用额外内存，适合于数据规模 n 较小的情景。

【算法思路】：连续进行 n 轮遍历，每一轮遍历都让相邻数两两交换，确定一个已排序的数，n 轮遍历后就会确定所有的数。



【编码思路】：双层循环，外层循环负责从右至左确定已排序序列，内存循环负责从左至右交换未排序序列中的相邻数。此处有个改进小技巧，内层循环维护一个 bool 值，如果当前一轮遍历没有交换过说明已完成排序，直接退出外层循环。

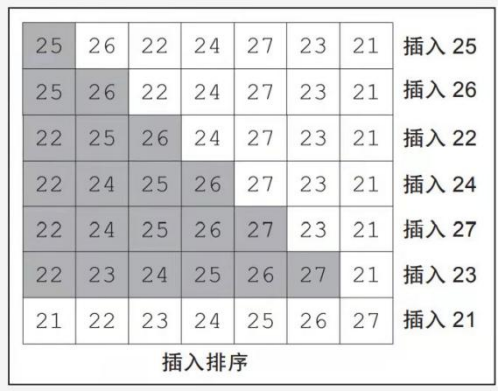
```cpp
/**冒泡排序**/
void bubbleSort(vector<int> &nums) {
    // 从右至左确定已排序序列
    for(int i=nums.size()-1;i>0;i--){
        // 从左至右交换未排序序列
        for(int j=0;j<i;j++){
            // 比较相邻数
            if(nums[j]>nums[j+1]){
                int temp=nums[j+1];
                nums[j+1]=nums[j];
                nums[j]=temp;
            }
        }
    }
}
```

```cpp
/**冒泡排序(改进后)**/
void bubbleSort(vector<int> &nums) {
    bool swapped=false;
    // 从右至左确定已排序序列
    for(int i=nums.size()-1;i>0;i--){
        swapped=false;
        // 从左至右交换未排序序列
        for(int j=0;j<i;j++){
            // 比较相邻数
            if(nums[j]>nums[j+1]){
                int temp=nums[j+1];
                nums[j+1]=nums[j];
                nums[j]=temp;
                swapped=true;
            }
        }
        if (!swapped) break;
    }
}
```

3、插入排序

插入排序（insert-sort）的性能和冒泡排序差不多，平均时间复杂度为$O(n^2)$，同样不占用额外内存，适合于数据规模 n 较小的情景。

【算法思路】：序列左侧视为已排序序列，序列右侧视为未排序序列，每次从未排序序列中移除一个元素到已排序序列，已排序序列每次加入新的元素就要重新交换一遍来确定顺序。



【编码思路】：双层循环，外层循环负责从左至右扩展已排序序列，内层循环负责从右至左在已排序序列中交换完成排序。

```
/**插入排序**/
void insertSort(vector<int> &nums) {
    int n = nums.size();
    // 从左至右扩展已排序序列
    for (int i = 1; i < n; i++) {
        // 从右至左在已排序序列交换完成排序
        for (int j = i; j > 0; j--) {
            // 交换元素 swap(nums[j], nums[j-1])
            if(nums[j] < nums[j-1]){
                int tmp = nums[j];
                nums[j] = nums[j-1];
                nums[j-1] = tmp;
            }
        }
    }
}
```

4、快速排序

快速排序（quick-sort）是性能鲁棒性最好的排序算法之一，任何情况下耗费时间都为 $O(n^2)$，而且不占用额外内存，适合于数据规模 n 较小的情景。

【算法思路】：将一个数组分成两个子数组，将两部分独立地排序。切分地位置取决于数组的内容。先随意取第一个元素作为切分元素，从数组左端开始向右扫描直到找到一个大于等于它的元素，在从数组右端开始向左扫描找到一个小于等于它的元素，交换这两个元素的位置，直到两个指针相遇，最后和切分元素交换位置即可。

【编码思路】：双层循环，外层循环负责划分已排序序列和未排序序列，内存循环负责从未排序序列中查找最值，找到后交换两数即可。

```cpp
// 分区函数
int partition (vector<int> &nums, int first, int last) {
    int key = nums[first];//默认第一个为基准值
    while (first < last) {
        // 从右往左找到第一个小于key的值
        while (first < last && nums[last] >= key) --last;
        nums[first] = nums[last];
        // 从左往右找到第一个大于key的值
        while (first < last && nums[first] <= key) ++first;
        nums[last] = nums[first];
    }
    nums[first] = key;// 基准值归位
    return first;
}
/**快速排序**/
void quickSort(vector<int> &nums, int left, int right) {
    if (left < right) {
        int index = partition(nums,left,right); // 分区
        quickSort(nums, left, index-1);
        quickSort(nums, index+1, right);
    }
}
```

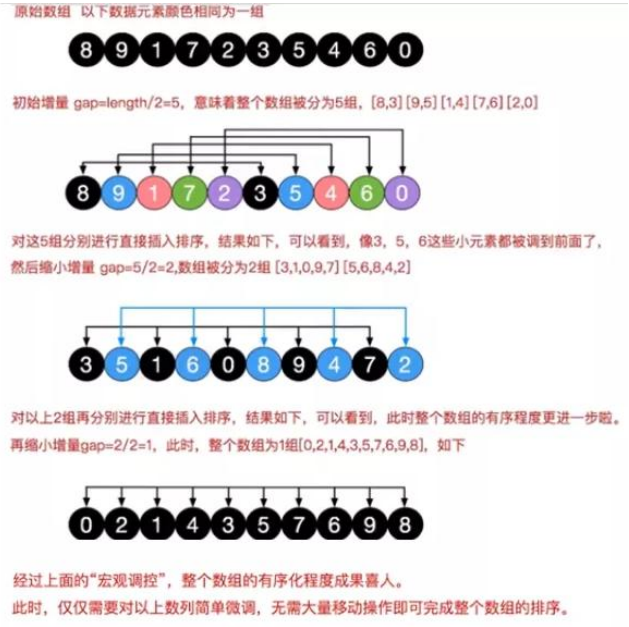
```cpp
#include<vector>
#include<iostream>
#include<string>
using namespace std;

/**快速排序（合并写法）**/
void quickSort(vector<int> &nums, int left, int right) {
    if (left >= right) return; //递归终止条件
    // 开始分区
    int first = left, last = right, key = nums[first];//默认第一个为基准
值
    while (first < last) {
        // 从右往左找到第一个小于key的值
        while (first < last && nums[last] >= key) --last;
        nums[first] = nums[last];
        // 从左往右找到第一个大于key的值
        while (first < last && nums[first] <= key) ++first;
        nums[last] = nums[first];
    }
    nums[first] = key;// 基准值归位
    // 结束分区
    quickSort(nums, left, first-1);
    quickSort(nums, first+1, right);
}

// 显示数组内容
void showArray(vector<int> &nums,string name="",int start=0){
    cout<<name;
    for (int i = start; i < nums.size(); i++)
    {
        cout<<nums[i]<<" ";
    }
    cout<<endl;
}

int main()
{
    //vector<int> nums={5,2,3,1,4,7,8,6};
    vector<int> nums={5,2,8,1,4,7,3,6};
    //vector<int> nums={7,8,6};

    showArray(nums,"排序前：");
    //quickSort(nums,0,nums.size()-1);
    quickSort1(nums,0,nums.size()-1);
    showArray(nums,"排序后：");
    system("pause");
    return 0;
}
```

5、希尔排序

希尔排序（shell-sort）是 1959 年希尔（Donald Shell）提出的一种排序算法，可以看做是插入算法的改进版本，也成为了**缩小增量排序**。假设从小到大排序，简单插入算法插入一个较小的数时需要后移的次数明显增多，

【算法思路】：整个数组按照增量 h 分成 h 个分组，各个分组独立进行插入排序；然后不断缩小增量 h，当 h 等于 1 的时候整个数组就完成了排序。。增量 h 的取值也叫做 Shell

增量序列，如最常见的折半增量序列{n,n/2,...,1}。



【编码思路】：三层循环，第一层循环负责增量 h 的取值，第二层循环负责切换不同 h 分组最后一个元素索引的取值，第三层负责遍历当前 h 分组中所有的元素，把顺序不对的相邻元素交换过来。以折半增量序列为代表的交换式希尔算法代码实现如下：

```
/**希尔排序（交换法）**/
void shellSort(vector<int> &nums){
    int n = nums.size();
    for(int h = n/2;h>0;h/= 2){
        // 遍历各个组（从h开始）
        for (int i = h; i < n; i++) {
            //遍历各个组中的所有的元素（从右到左，每个组元素索引间隔h）
            for (int j = i - h; j>=0; j -= h){
                //使用从小到大进行排序
                if (nums[j] > nums[j + h]) {
                    swap(nums[j],nums[j + h]);
                }
            }
        }
    }
}
```

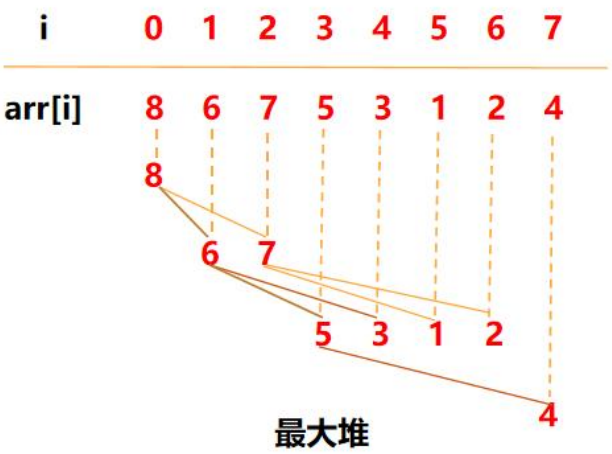
有时候我们简单换一下增量 h 的取值序列，就可以显著提升希尔排序的算法效率，如我们将 h 的取值切换为三分之一来看一下：

```
/**希尔排序（缩小增量为三分之一）**/
void shellSort2(vector<int> &nums) {
    int n = nums.size();
    int h = 1;
    //初始增量h应该比n/3稍大，不能直接为n/3（因为必须保证h的最后一个值为1）
    while (h < n/3) h = 3*h + 1;
    for(;h >= 1;h /= 3) {
        // 遍历各个组（从h开始）
        for (int i = h; i < n; i++) {
            //遍历各个组中的所有的元素（从右到左，每个组元素索引间隔h）
            for (int j = i-h; j >= 0; j -= h) {
                if(nums[j] > nums[j+h]){
                    swap(nums[j],nums[j+h]);// 交换两个元素
                }
            }
        }
    }
}
```

该 h 的取值比折半 Shell 增量的效率要高（力扣 912 题该 h 取值的希尔算法可以通过测试。）

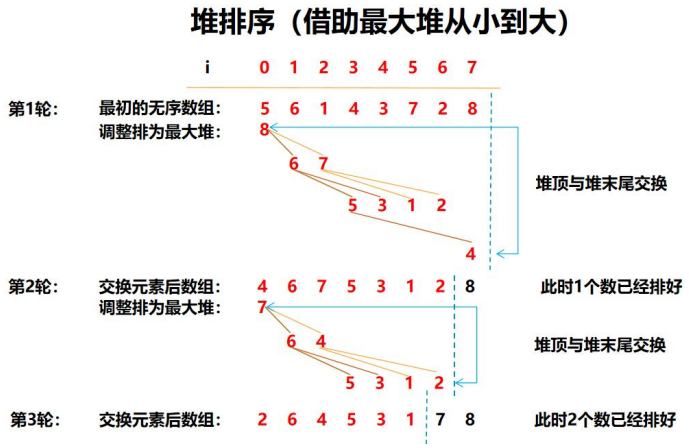
6、堆排序

堆排序（heap-sort）是利用一种叫做堆（heap）的数据结构所设计的排序算法。堆本质上就是用数组实现的二叉树，所以也叫做二叉堆。堆有一个性质，叫做堆有序。根据这种堆有序的性质，堆分为 2 种，最大堆（大顶堆）和最小堆（小顶堆）。在最大堆中，父节点的值都比每一个子节点的值要大；在最小堆中，父节点的值都比每一个子节点的值要小。（注意和二叉搜索树的区别，二叉搜索树是左子树的值比父节点的值小，右子树的值比父节点的值大）。下图是一个最大堆的内部数据的逻辑结构，根据这种特点，给定任意一个数所在的索引 index，我们就可以知道它的父节点索引或者左右子节点的索引。




```
// 父节点的索引
int parent(int index) {
    return (index-1) / 2;
}
// 左孩子的索引
int left(int root) {
    return root * 2+1;
}
// 右孩子的索引
int right(int root) {
    return root * 2 + 2;
}
```

【算法思路】：假设从小到大排序，堆排序的基本思路是先将整个无序数组构造成一个最大堆，那么最大堆的根节点一定就是全局最大的数，将其取出来放入已排序序列；删除掉根节点的最大堆在重新构建一个最大堆，从而又找到了一个局部最大值，接着将其放入已排序序列，当无序序列中构建最大堆的数只剩下一个时，排序也就完成。过程如下：



【编码思路】：在上述思路分析中，可知将一个无序序列转换为堆是最关键的步骤。最大堆是一个父节点都大于子节点的二叉树，在编码上可以采取递归的方式来实现。我们首先借助 C++的优先级队列来实现排序。

在大多数编程语言中，优先级队列（Priority Queue）就可以看做是一个堆结构（其底层

可能就是借助堆实现的），优先级队列的特点就是其队首元素始终是整个队列中最大或者最小的数，这和最大堆或最小堆的特点一样。所以，我们给出如下代码即可完成排序：

```
#include<queue> //包含priority_queue
void heapSort_withPQ(vector<int> &nums){
    int len=nums.size();
    priority_queue<int> max_heap;//初始化一个优先级队列（默认就是最大堆的特点）
    // 将整个数组元素加入最大堆
    for(int i = 0; i <len; i++){
        max_heap.push(nums[i]);
    }
    // 进行堆排序
    for(int i = len - 1; i >= 0; i--){
        nums[i] = max_heap.top(); // 把栈顶元素赋给末尾元素，剩下的元素max_heap会自动调整为最大堆
        max_heap.pop;
    }
}
```

上述编码其实就是堆排序的思路，借助一个最大堆或最小堆，把无序序列始终调整为堆有序，其堆顶元素一定是全局最大或最小。现在假设面试官不让我们继续使用 Priority Queue，让我们手写堆的调整结构，不要怕，难度也不会增加很多。

假设我们有一个函数 adjust()，其功能就是负责将 i 为根节点的子树调整为堆有序，即满足最大堆或最小堆，那么我们借助这个 adjust 就可以写出如下堆排序的代码：

```
// 堆排序
void heapSort(vector<int> &nums){
    int len=nums.size();
    // 构建最大堆（从右向左依次传入父节点的索引i，最右侧的父节点索引一定是（len-1）/2）
    for(int i = (len-1) / 2 ; i >= 0; i--){
        adjust(nums, len, i);// 负责将i为根节点的子树调整为堆
    }
    // 进行堆排序
    for(int i = len - 1; i >= 1; i--){
        // 把最大堆的堆顶元素与最后一个元素交换
        swap(nums[0], nums[i]);
        // 调整剩余的打乱的数为最大堆，其根节点索引始终是0，长度为i（i=1时表示排序完成）
        adjust(nums, i, 0);
    }
}
```

现在我们的任务就是写出 adjust 函数，其也利用了二叉树递归的思路，只要保证每个小子树中父节点比孩子节点都大或者都小即可，如下：

```
//调整为堆有序，len是调整的序列长度，parent是堆对应的二叉树的根节点索引
void adjust(vector<int> &nums, int len, int parent){
    int left = 2 * parent + 1; // parent的左子节点
    int right = 2 * parent + 2;// parent的右子节点

    int maxIdx = parent;//假设父节点为最大值
    // 如果左子节点大于最大值，则最大值设为左子节点
    if (left < len && nums[left] > nums[maxIdx])     maxIdx = left;
    // 如果右子节点大于最大值，则最大值设为右子节点
    if (right < len && nums[right] > nums[maxIdx])     maxIdx = right;
    // 如果此时父节点已经不是最大值
    if (maxIdx != parent) {
        swap(nums[maxIdx], nums[parent]);//交换父节点和最大值，
        adjust(nums, len, maxIdx);//交换后，原来最大值的位置可能已不是其对应子树的最大值，所以要重新调整下
    }
}
```

【性能分析】：

堆排序	O(n log *n*)	O(n log *n*)	O(n log *n*)	O(1)	不稳定

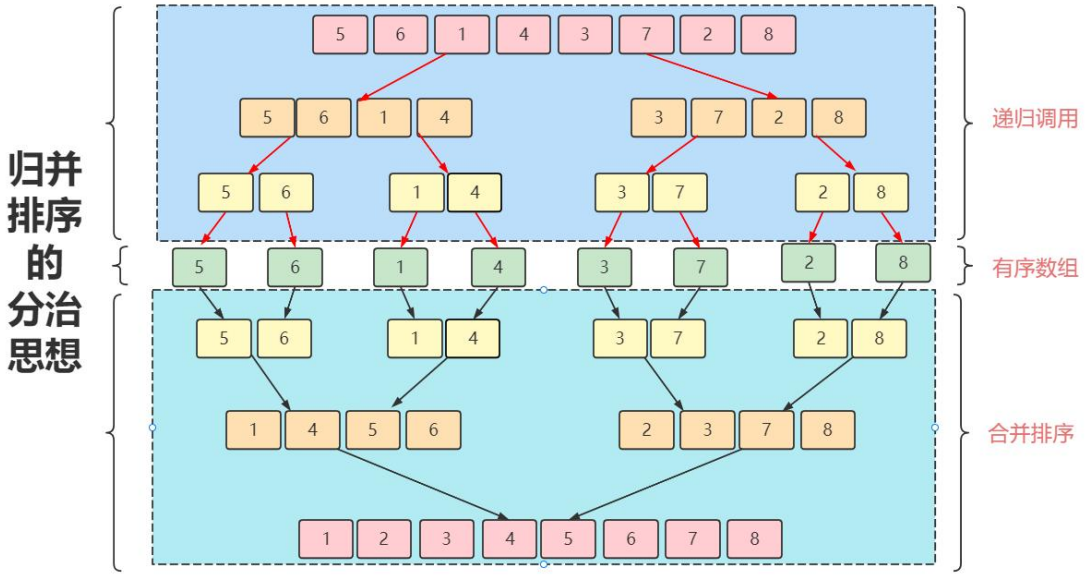
此时我们已经完成了整个堆排序的任务，最重要的是 adjust 的实现，读者要多默写几遍。学习堆排序后，我们实际也同时掌握了堆、优先级队列的核心知识，一石三鸟。

二、外排序

1、归并排序

归并排序（merge-sort）采用了分治思想（即将一个大问题分解成若干个小问题来解决），分治思想在编码上一般容易使用递归法。

【算法思路】：归并排序认为如果要给一个数组排序，可以把整个大数组对等分成 2 个小数组，2 个小数组继续分割，直到分出的每个小数组大小都是 1，只有 1 个元素的数组一定是有序数组了，然后再把 2 个有序数组合并成新的更大数组，合并的时候需要重新排下序，合并后的数组就又有序了。

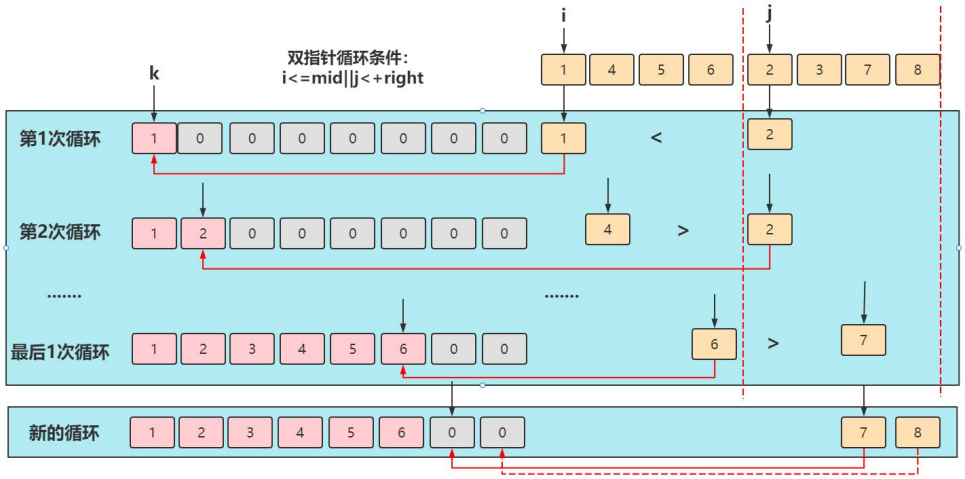


给 2 个小的有序数组（长度=N/2）重新排序通常比直接排序一个大的无序数组（长度=N）要快，这也是归并排序要先划分后合并的原因，先划分直至每个数组都是有序数组，然后合并排序成一个新的有序数组。

【编码思路】：在上述的思路分析中，首先递归调用来不断分割数组的编码很简单了，如下：

```java
/**归并排序(递归法)**/
void mergeSort(vector<int> &nums, int left, int right){
    // 如果 left == right，表示数组只有一个元素，则不用递归排序
    if (left < right) {
        // 把大的数组分隔成两个数组
        int mid = left + (right - left) / 2;
        // 对左半部分进行排序
        mergeSort(nums, left, mid);
        // 对右半部分进行排序
        mergeSort(nums, mid + 1, right);

        // 将两部分合并后需要重新排一次序
        // ToDo
    }
}
```

接下来就是考虑如何合并 2 个有序数组了，归并排序使用了申请新数组的双指针法来合并排序。首先直接申请一个足够容纳 2 个有序数组的临时数组，然后 i 指针指向数组的左半部分 nums[left,mid]，j 指针指向数组的右半部分 nums[mid+1,right]。接下来依次比较 2 个数组中的每个元素，哪个小数组的数更小就放入临时数组中，如下：



最后双指针循环结束后，原数组中还可能会有剩余的没有复制，因为双指针的循环条件是有一个指针达到其终点即可，所以循环结束时至少有一个数组的元素全部被复制完（图中是 i 指针指向的左数组），另一个数组则需要一个新的循环来复制，最后直接上代码。

```cpp
/**归并排序(递归法)**/
void mergeSort(vector<int> &nums, int left, int right){
    // 如果 left == right，表示数组只有一个元素，则不用递归排序
    if (left < right) {
        // 把大的数组分隔成两个数组
        int mid = left + (right - left) / 2;
        // 对左半部分进行排序
        mergeSort(nums, left, mid);
        // 对右半部分进行排序
        mergeSort(nums, mid + 1, right);

        // 将两部分合并后需要重新排一次序
        //合并第1步，新建一个临时数组，通过指针法将原数组左半部分和右半部分看做两个小数组
        vector<int> temp(right - left + 1);
        int i = left;
        int j = mid + 1;
        int k = 0;
        //合并第1步，分别选取左右两半部分中的较小值复制到临时数组
        while (i <= mid && j <= right) {
            // 谁小先放谁到临时数组,相等则先放左半部分的
            if (nums[i] <= nums[j]) {
                temp[k++] = nums[i++];
            } else {
                temp[k++] = nums[j++];
            }
        }
        //合并第3步，跳出循环的条件i>mid或者j>right，能来到第3步说明
        //此时两个小数组中至少有一个一定没有剩余
        //不知道哪个还有剩余，干脆两者都判断下，有剩余直接进循环复制
        while(i <= mid) temp[k++] = nums[i++];
        while(j <= right) temp[k++] = nums[j++];
        // 此时原有数组的左右部分全部复制到临时数组并已排序，把临时数组复制回原数组
        for (i = 0; i < k; i++) {
            nums[left++] = temp[i];
        }
}
```

【性能分析】：先给出归并排序的性能分析结论：

算法名称	最好时间复杂度	最坏时间复杂度	平均时间复杂度	空间复杂度	是否稳定
归并排序	O(nlogn)	O(nlogn)	O(nlogn)	O(n)	稳定

归并排序分为两部分：递归分组和合并排序。通过递归方式把整个大数组一层一层的折半分组，则由完全二叉树的深度可知，整个排序过程需要进行 logn（向上取整）次分组,最后合并排序的时候需要将每个有序小数组的数都扫描一次，整个排序过程一共扫描了整个大数组的数，所以每轮合并排序的时间复杂度为 O(n)，则总的时间复杂度为 O(nlogn)。

归并排序的执行效率与要排序的原始数组的有序程度无关，所以在最好，最坏，平均情况下时间复杂度均为 O(nlogn) 。

归并排序所创建的临时数组都会在方法结束时释放，单次归并排序的最大空间是 n,所以归并排序的空间复杂度为 O(n).

归并排序的稳定性，取决于合并排序的时候如何处理相等的两个数。在归并排序的代码中，设置了 nums[i]<=nums[j]，当两个元素相同时，先放 nums[i] 的值到大数组中，所以两个相同元素的相对位置没有发生改变，此处归并排序是稳定的排序算法。

虽然归并排序时间复杂度很稳定，但是他的应用范围却不如快速排序广泛，这是因为归

并排序不是原地排序算法，空间复杂度不为 O(1)。

二、附加资料

本章节的主要参考资料：

（四）排序【C++刷题】-Caoer199-博客园(cnblogs.com)

十大排序算法(背诵版+动图) - 力扣（LeetCode）

借助**快速排序**的算法思想可以快速解决如下 LeetCode 题目：

912. 排序数组 - 力扣（LeetCode）

75. 颜色分类 - 力扣（LeetCode）

借助**快速排序**的算法思想可以快速解决如下 LeetCode 题目：

215. 数组中的第 K 个最大元素 - 力扣（LeetCode）

剑指 Offer II 076.数组中的第 k 大的数字 - 力扣（LeetCode）

```cpp
int findKthLargest(vector<int>& nums, int k) {
    int len = nums.size();
    return quickFind(nums,k-1,0,len-1);//从大到小排序，第k大就是第k-1个索引，返回第k-1个索引的值
}
int quickFind(vector<int>& nums, int k,int left,int right){
    int first=left,last=right,key=nums[first];
    // 开始分区
    while(first<last){
        // 从右到左找第一个比key小的数
        while (first<last&&nums[last]<=key)last--;
        nums[first]=nums[last];
        // 从左到右找第一个比key大的数
        while (first<last&&nums[first]>=key )first++;
        nums[last]=nums[first];
    }
    nums[first]=key;//基准值归位
    // 结束分区
    if(first==k){
        return nums[first];//first是基准值索引，表示第first个
    }
    else if(first>k){
        return quickFind(nums, k,left,first-1);
    }
    else {
        return quickFind(nums, k,first+1,right);
    }
}
```