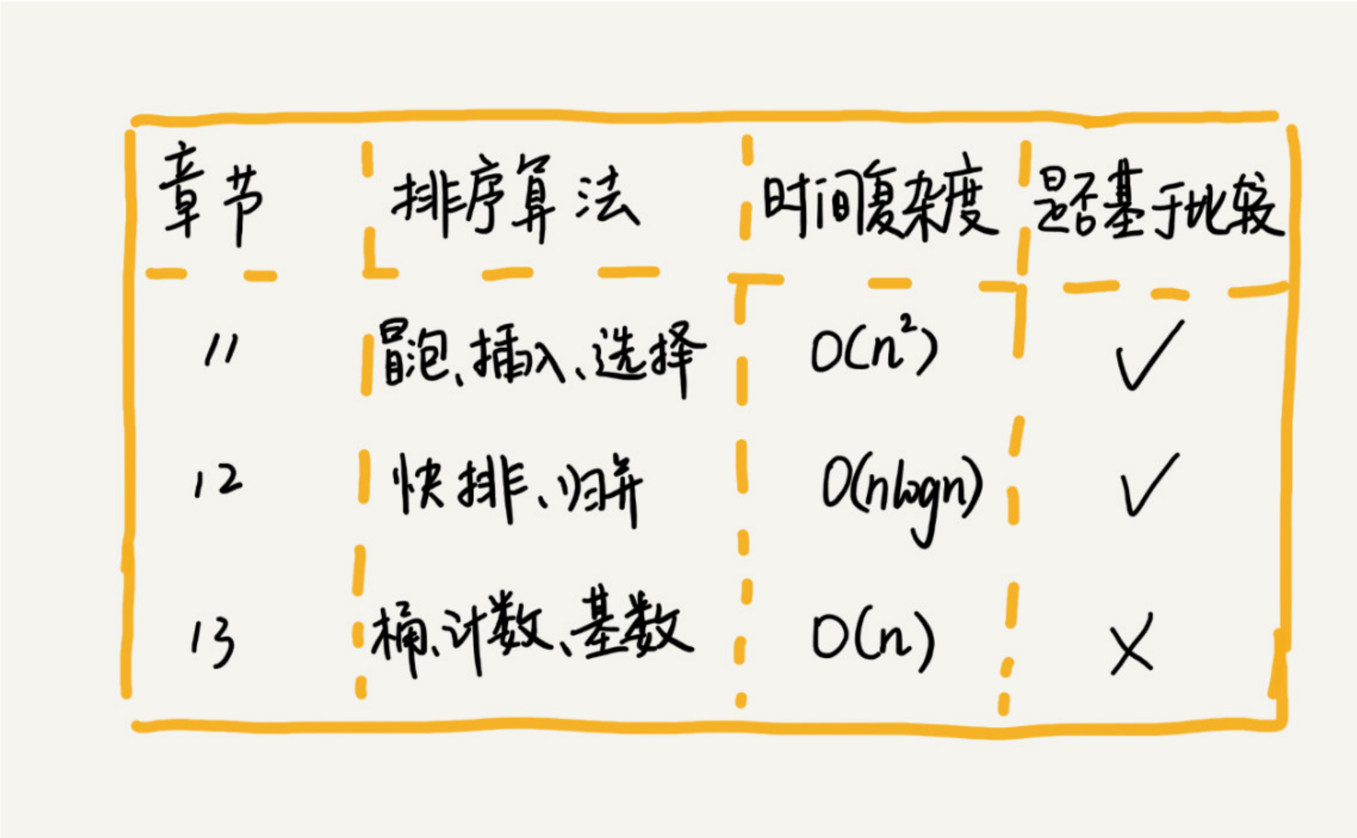
排序

上篇：开篇题目，为啥们插入排序比冒泡排序更受喜欢?

最经典、最常用的排序：冒泡排序，插入排序、选择排序、归并排序、快速排序、计数排序、基数排序、桶排序。



章节	排序算法	时间复杂度	是否基于比较
11	冒泡、插入、选择	$O(n^2)$	√
12	快排、归并	$O(nlogn)$	√
13	桶、计数、基数	$O(n)$	×

思考题：插入排序和冒泡排序的时间复杂度相同，都是 O(n2)，在实际的软件开发里，为什么我们更倾向于使用插入排序算法而不是冒泡排序算法呢?

如何分析一个排序算法

算法的执行效率 1、最好情况、最坏情况。平均情况时间复杂度 处理上述复杂度，还有说出具体什么情况的原始数据会导致这种情况 我们要知道排序算法在不同数据中的性能表现

2、时间复杂度的系数、常阶、低阶 对统一阶时间复杂度的排序算法性能对比的时候，我们吧系数、常数、低阶也要考虑进来

3、比较次数和交换(或交换)次数 基于比较的排序算法的执行过程，会设计两种操作，一种是元素比较大小，另一种是元素交换或移动，在分析算法的时候应该将比较和移动次数考虑进去

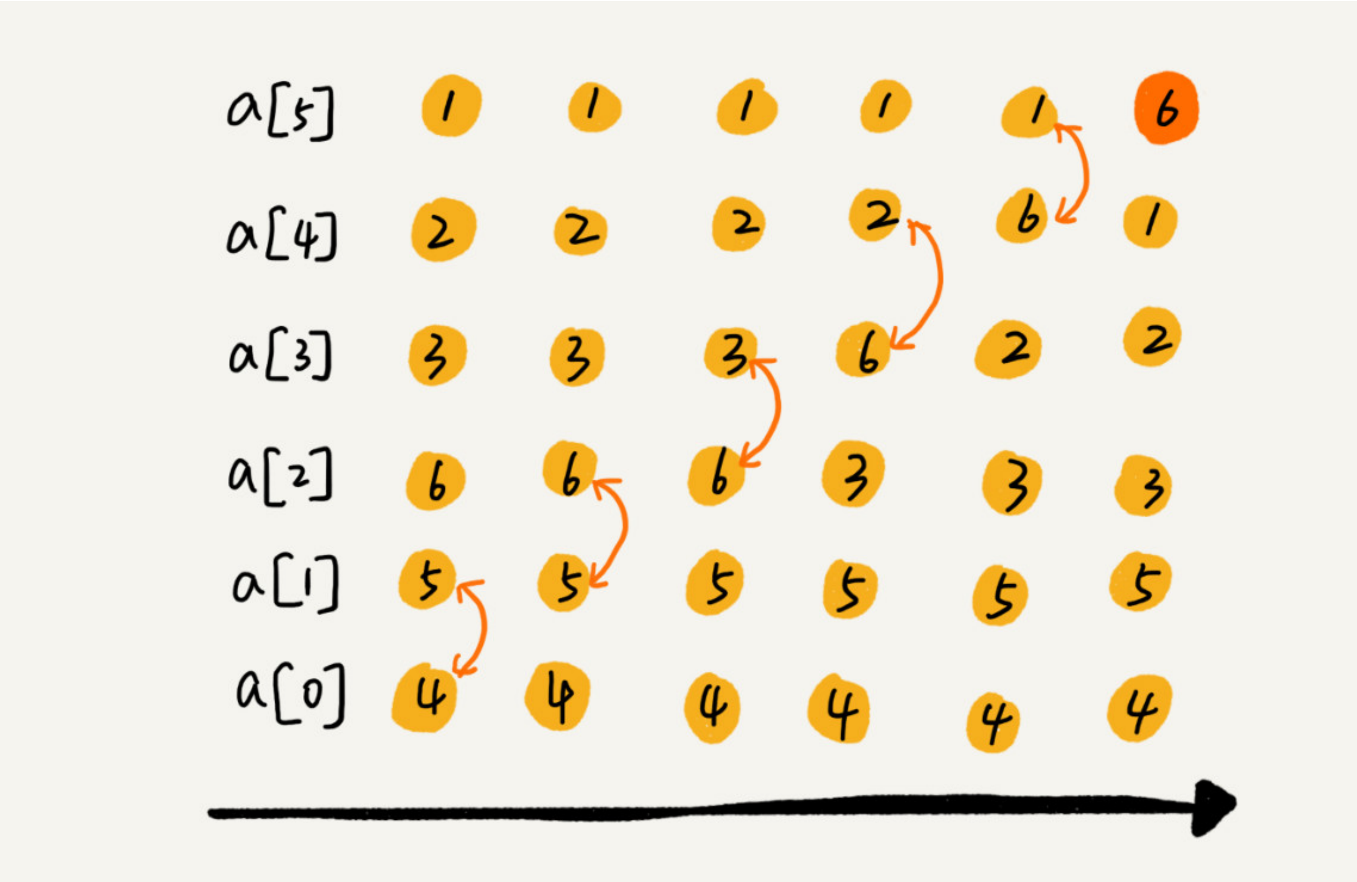
排序算法的内存消耗 算法的内存消耗可以使用空间复杂度来衡量。 原地排序，特指空间复杂度是O(1)的排序算法。

排序算法的稳定性 稳定性。这个概念是说，如果待排序的序列中存在值相等的元素，经过排序之后，相等元素之间原有的先后顺序不变。

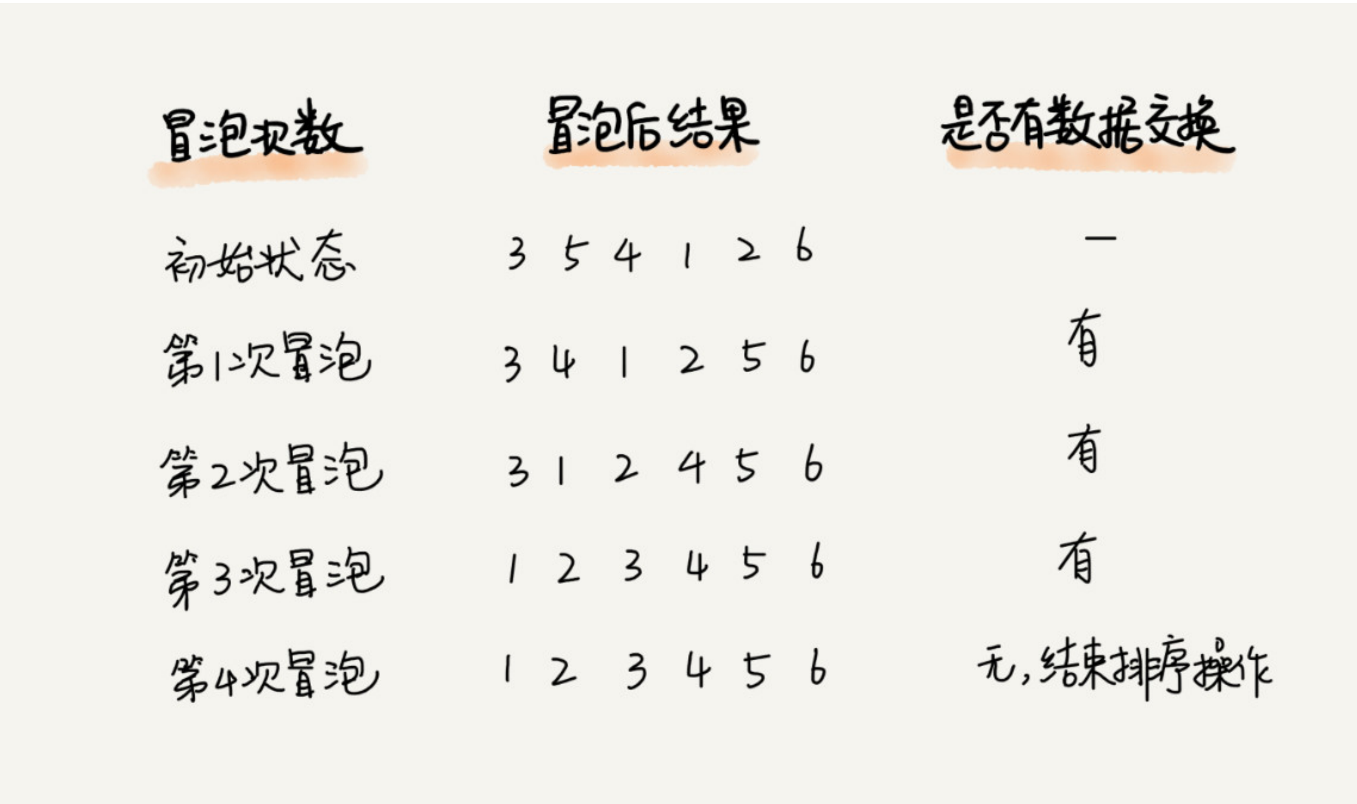
冒泡排序

冒泡排序只会操作相邻的两个数据。 每次对相邻两个元素进行比较，之后跟前判断进行交换。 一次冒泡会让至少一个元素移动到它应该在的位置，重复 n 次，就完成了 n 个数据的排序工作。

我们要对一组数据 4，5，6，3，2，1，从小到大进行排序。第一次冒泡操作的详细过程就是这样：



优化的冒泡排序，增加一个flag，判断是否存在数据交换，如果存在则没有排序完成，反之完成



具体的冒泡排序：

```
/**
 * 冒泡排序，第一层控制冒泡次数，第二层控制相邻元素之间的交换。
 * @param a
 */
public static void bubbleSort(int[] a){
    if(a.length <= 1){
        return;
    }
    for(int i = 0; i < a.length;i++){ //控制冒泡次数
        boolean flag = false; //增加判断条件，如果已经排序好了及时停止。
        for(int j = 0; j < a.length - i -1; j++){
            if(a[j] > a[j+1]){ //只有a[j] > a[j+1]才会交换位置，等于是不加交换的，保证稳定性
                int temp = a[j];
                a[j] = a[j+1];
                a[j+1] = temp;
                flag = true;
            }
        }
        if(!flag) break;
    }
}
```

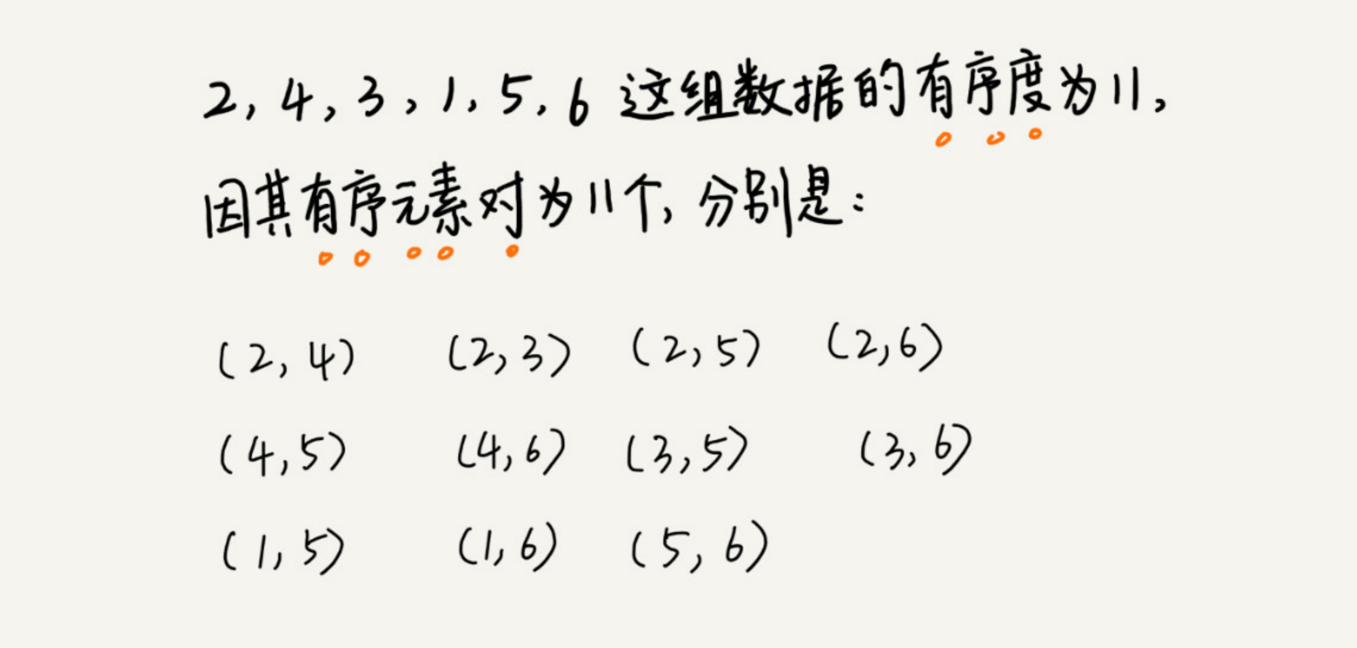
三个问题： **冒泡排序是否是原地排序算法？** 只需要常量级的临时空间，所以它的空间复杂度为 O(1)，是一个原地排序算法。

冒泡排序是稳定的排序算法吗？ 当有相邻的两个元素大小相等的时候，我们不做交换，相同大小的数据在排序前后不会改变顺序，所以冒泡排序是稳定的排序算法。

冒泡排序的时间复杂度是多少？ 最好情况是O(n),此时数据已经有序，只需做一次冒泡操作； 最坏情况是O(n^2),此时数据刚好事倒序，需要做n次冒泡操作；

平均时间复杂度，通过"有序度"和"逆序度"两个概念来进行分析： **有序度**是数组中具有有序关系的元素对的个数。

```
有序元素对：a[i] <= a[j]，如果i < j。
```



对于一个倒序排列的数组，比如 6，5，4，3，2，1，有序度是 0；对于一个完全有序的数组，比如 1，2，3，4，5，6，有序度就是 n*(n-1)/2，也就是 15。我们把这种完全有序的数组的有序度叫作满有序度。

逆序度的定义正好跟有序度相反（默认从小到大为有序）

```
逆序元素对：a[i] > a[j], 如果i < j。
```

逆序度 = 满有序度 - 有序度

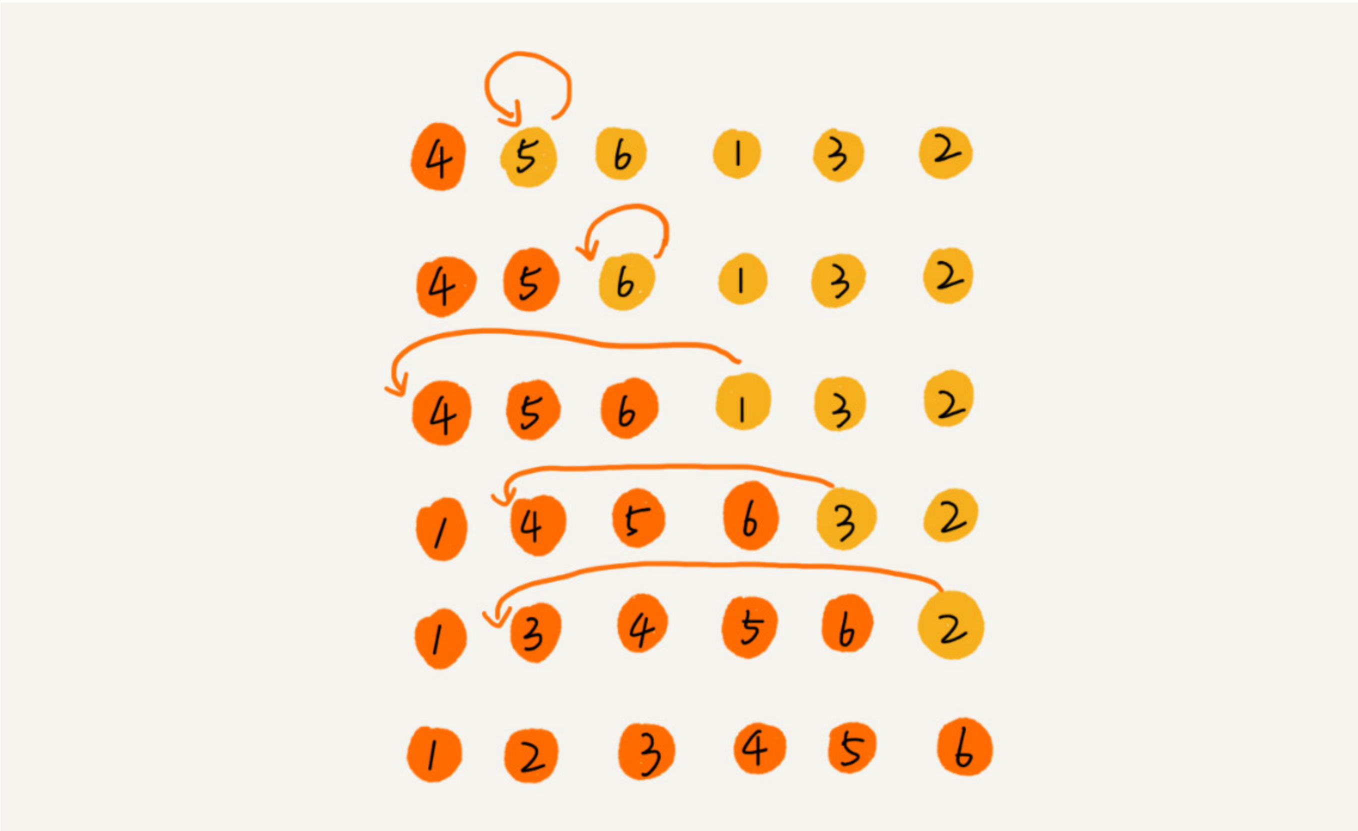
如果那一开始那个冒泡例子来看

冒泡排序包含两个操作原子，比较和交换。每交换一次，有序度就加 1 交换次数总是确定的，即为逆序度，也就是n*(n-1)/2–初始有序度。 最坏情况下，初始状态的有序度是 0，所以要进行 n*(n-1)/2 次交换。最好情况下，初始状态的有序度是 n*(n-1)/2，就不需要进行交换。我们可以取个中间值 n*(n-1)/4。

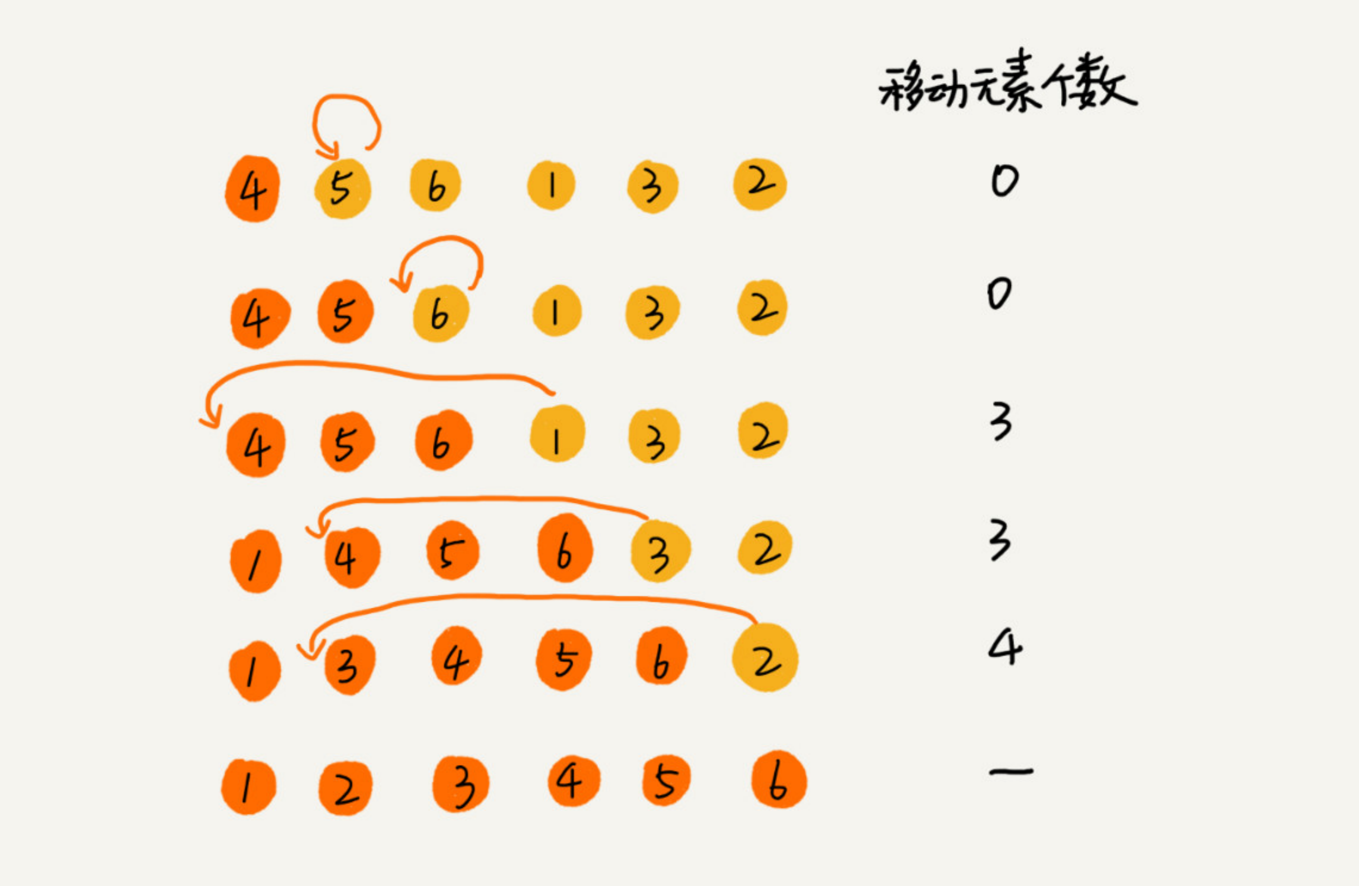
换句话说，平均情况下，需要 n*(n-1)/4 次交换操作，比较操作肯定要比交换操作多，而复杂度的上限是 O(n^2)，所以平均情况下的时间复杂度就是 O(n^2)。

插入排序

将数组中的数据分为两个区间，已排序区间和未排序区间。初始已排序区间只有一个元素，就是数组的第一个元素。插入算法的核心思想是取未排序区间中的元素，在已排序区间中找到合适的插入位置将其插入，并保证已排序区间数据一直有序。重复这个过程，直到未排序区间中元素为空，算法结束。



一种是元素的比较，一种是元素的移动； 移动操作的次数总是固定的等于逆序度；



具体代码：

```
/**
 * 插入排序
 * @param a
 */
public static void insertionSort(int[] a){
    for(int i = 1; i < a.length; i++){
        int value = a[i]; //记录数据，防止在数据移动过程中被覆盖
        int j = i - 1; //记录空位，即插入位置，在最后可以直接插入
        for(; j >= 0; j--){
            if(a[j] > value){
                a[j+1] = a[j];
            }else {
                break;
            }
        }
        a[j+1] = value;
    }
}
```

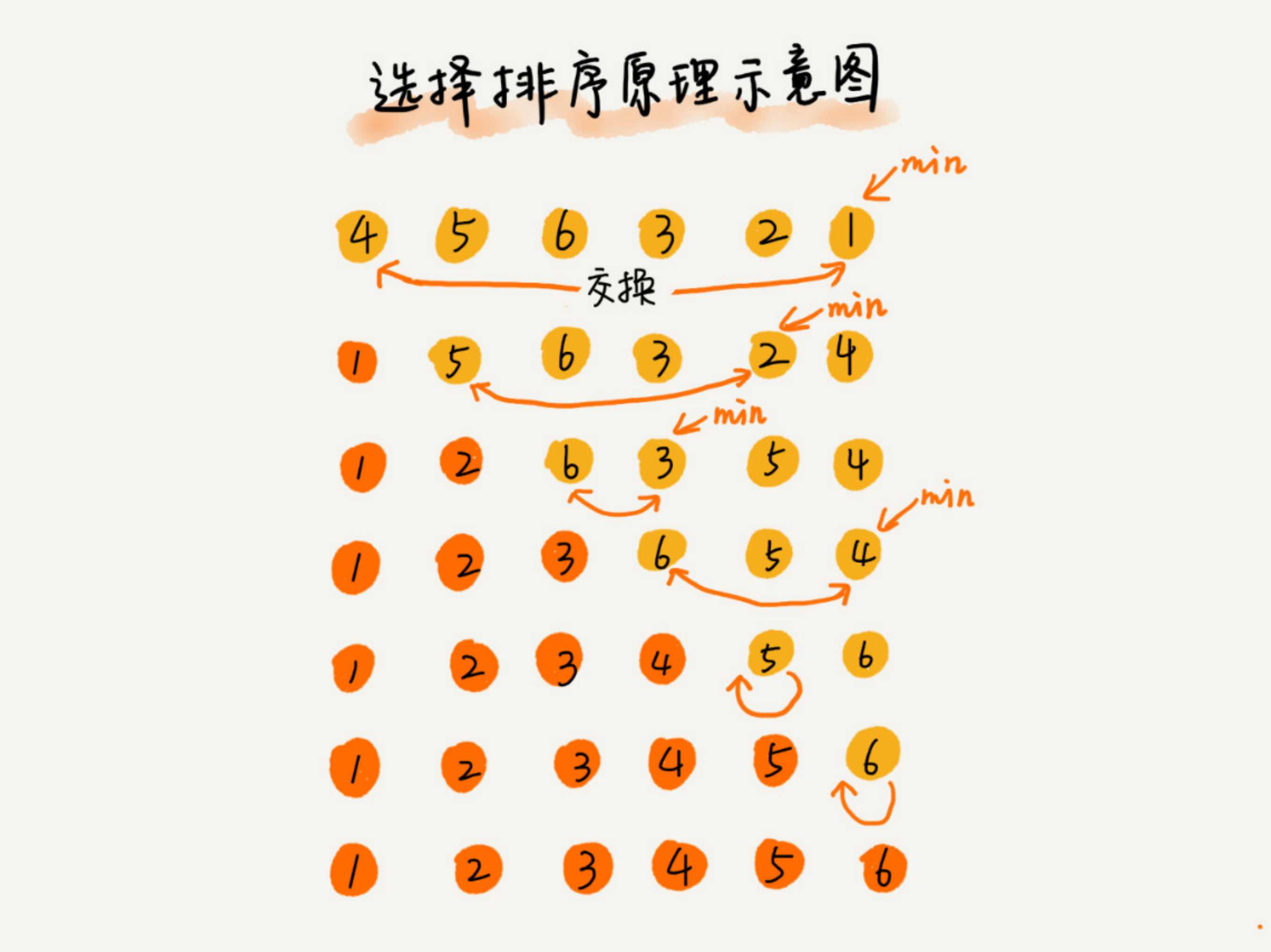
三个问题： **冒泡排序是否是原地排序算法？** 不需要额外的临时空间，所以它的空间复杂度为 O(1)，是一个原地排序算法。

冒泡排序是稳定的排序算法吗? 我们可以选择将后面出现的元素，插入到前面出现元素的后面，这样就可以保持原有的前后顺序不变，所以插入排序是稳定的排序算法。

冒泡排序的时间复杂度是多少? 最好情况是O(n),此时数据已经有序 最坏情况是O(n^2),此时数据刚好是倒序，每次插入都相当于在数组的第一个位置插入新的数据； 平均复杂度：数组插入的复杂度是O(n),但是插入排序相等于执行了n次插入操作，平均复杂度是O(n^2)

选择排序

但是选择排序每次会从未排序区间中找到最小的元素，将其放到已排序区间的末尾。



选择排序空间复杂度为 O(1)，是一种原地排序算法。 选择排序是一种不稳定的排序算法。选择排序每次都要找剩余未排序元素中的最小值，并和前面的元素交换位置，这样破坏了稳定性。

代码：

```
/**
 * 选择排序
 * @param a
 */
public static void selectionSort(int [] a){
    for(int i = 0; i < a.length; i++){//遍历数组中所有的位置
        int min = i;//默认该位置上的现有的数就是未排序区最小的
        for(int j = i + 1; j < a.length;j++){//向后遍历
```

```
            if(a[j] < a[min]){ //找出更小的数
                min = j; //记录下标
            }
        }
        int temp = a[i];  //交换位置
        a[i] = a[min];
        a[min] = temp;
    }
}
```

解答开篇

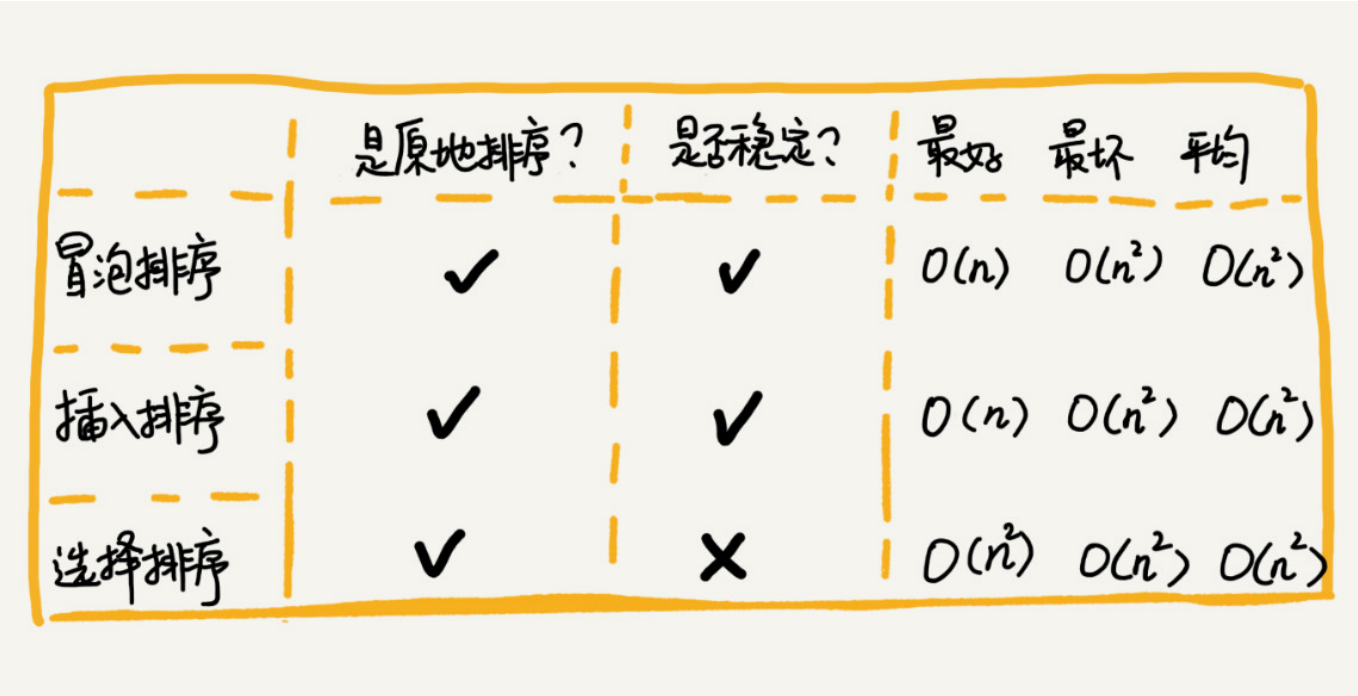
冒泡排序不管怎么优化，元素交换的次数是一个固定值，是原始数据的逆序度。 插入排序是同样的，不管怎么优化，元素移动的次数也等于原始数据的逆序度。

从代码实现上来看，冒泡排序的数据交换要比插入排序的数据移动要复杂，冒泡排序需要 3 个赋值操作，而插入排序只需要 1 个。我们来看这段操作：

```
冒泡排序中数据的交换操作：
if (a[j] > a[j+1]) { // 交换
    int tmp = a[j];
    a[j] = a[j+1];
    a[j+1] = tmp;
    flag = true;
}

插入排序中数据的移动操作：
    if (a[j] > value) {
    a[j+1] = a[j];  // 数据移动
    } else {
    break;
}
```

我们把执行一个赋值语句的时间粗略地计为单位时间（unit_time），然后分别用冒泡排序和插入排序对同一个逆序度是 K 的数组进行排序。用冒泡排序，需要 K 次交换操作，每次需要 3 个赋值语句，所以交换操作总耗时就是 3*K 单位时间。而插入排序中数据移动操作只需要 K 个单位时间。

冒泡，插入，选择排序小结



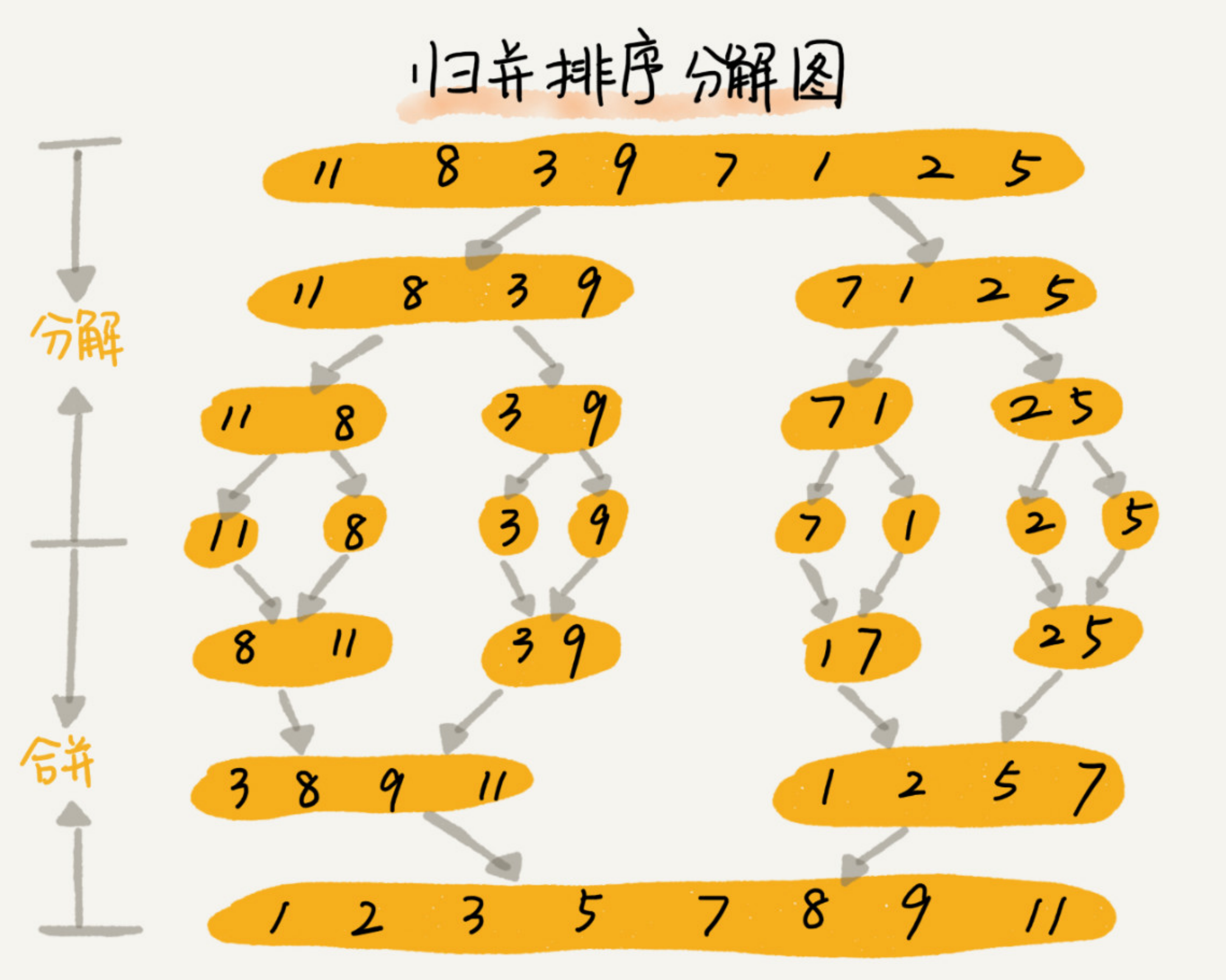
排序下篇：如何使用快排思想在O(n)内查找第K大元素

今天讲述两种时间复杂度是O(nlogn)的排序算法，归并排序和快速排序。两者都用到了分治思想 **开篇题目：如何在O(n)的时间复杂度中查找一个无序数组中的第K大的元素?**

归并排序的原理

如果要排序一个数组，我们先把数组从中间分成前后两部分，然后对前后两部分分别排序，再将排好序的两部分合并在一起，这样整个数组就都有序了。

具体的原理图如下：



分治思想，分而治之，将大问题化解为小问题来解决

分治算法一般都是用递归来实现的，分治是一种思想，递归时编程技巧。

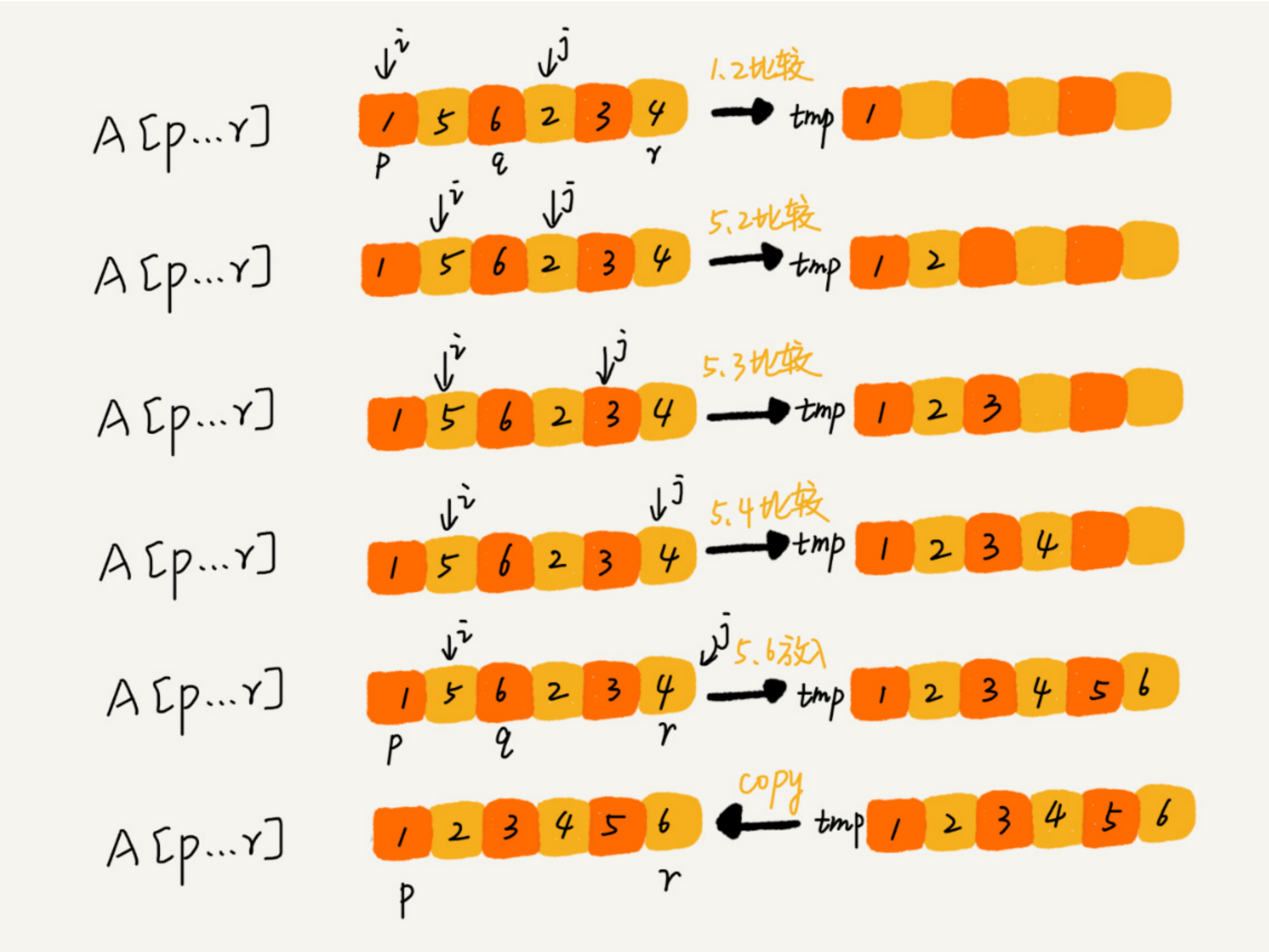
如何使用递归代码来实现归并排序

```
//归并排序算法，A是数组，n表示数组大小
merge_sort(A,n){
    merge_sort_c(A,0,n-1)
}

//递归调用函数
merge_sort_c(A,p,r){
    //递归终止条件
    if p >= r then return

    //取p到r之间中间位置q
    q = (p+r)/2
    //分治递归
    merge_sort_c(A,p,q)
    merge_sort_c(A,q+1,r)
```

```
    //将A[p...q]和A[q+1...r]合并为A[p...r]
    merge(A[p...r],A[p...q],A[q+1...r])
}
```

merge(A[p...r],A[p...q],A[q+1...r])函数的作用是，将已经有序的A[p...q]和A[q+1...r]合并成一个有序的数组，并且放入A[p...r]

我们用两个游标 i 和 j，分别指向 A[p...q] 和 A[q+1...r] 的第一个元素。比较这两个元素 A[i] 和 A[j]，如果 A[i]<=A[j]，我们就把 A[i] 放入到临时数组 tmp，并且 i 后移一位，否则将 A[j] 放入到数组 tmp，j 后移一位。 继续上述比较过程，直到其中一个子数组中的所有数据都放入临时数组中，再把另一个数组中的数据依次加入到临时数组的末尾，这个时候，临时数组中存储的就是两个子数组合并之后的结果了。最后再把临时数组 tmp 中的数据拷贝到原数组 A[p...r] 中。



代码实现：

```
/**
 * 归并排序
 * @param a
 */
public static void mergeSort(int[] a){
    //调用
    mergeSortC(a,0,a.length-1);
```

```
}

/**
 * 递归函数实现分而治之这个过程
 * @param a
 * @param start
 * @param end
 */
public static void mergeSortC(int[] a,int start,int end){
    //判断是否此时只有一个元素，递归终止条件
    if(start >= end){
        return;
    }
    //找到数组中间元素下标
    int mid = (start + end)/2;
    //递归左分
    mergeSortC(a,start,mid);
    //递归右分
    mergeSortC(a,mid+1,end);
    //两者合并
    merge(a,start,mid,end);
}

/**
 * 将两个分开的数组合并成一个数组，放回原数组位置
 * @param a
 * @param left
 * @param mid
 * @param right
 */
public static void merge(int[] a,int left,int mid,int right){
    int[] tmp = new int[a.length];
    int p1 = left;
    int p2 = mid + 1;
    int k = left;
    //数据比较，放入新的暂存空间
    while(p1 <= mid && p2 <= right){
        if(a[p1] <= a[p2]){
            tmp[k++] = a[p1++];
        }else{
            tmp[k++] = a[p2++];
        }
    }
    //将有剩余的数组全部放到暂存数组最后
    while (p1 <= mid){
        tmp[k++] = a[p1++];
    }
    while (p2 <= right){
        tmp[k++] = a[p2++];
    }
    //数组拷贝回原来数组
```

```
    for(int i = left;i <= right; i++){
        a[i] = tmp[i];
    }
}
```

归并排序的性能分析

第一、归并排序是稳定的排序算法，只要在merge函数中保证稳定就可以实现稳定 第二、归并排序的时间复杂度是多少？ 问题a可以分解为问题b，c求解a就变成了求解b,c,最后得出这样的递推关系 T(a) = T(b) + T(c) + K K等于问题b，c的结果合并成问题a的结果所消耗的时间 不仅递归求解的问题可以写成递推公式，递归代码的时间复杂度也可以写成递推公式。

```
T(1) = C;   n=1时，只需要常量级的执行时间，所以表示为C。
T(n) = 2*T(n/2) + n;  n>1

T(n) = 2*T(n/2) + n
     = 2*(2*T(n/4) + n/2) + n
     = 4*T(n/4) + 2*n
     = 4*(2*T(n/8) + n/4) + 2*n
     = 8*T(n/8) + 3*n
     = 8*(2*T(n/16) + n/8) + 3*n
     = 16*T(n/16) + 4*n
     ......
     = 2^k * T(n/2^k) + k * n
```

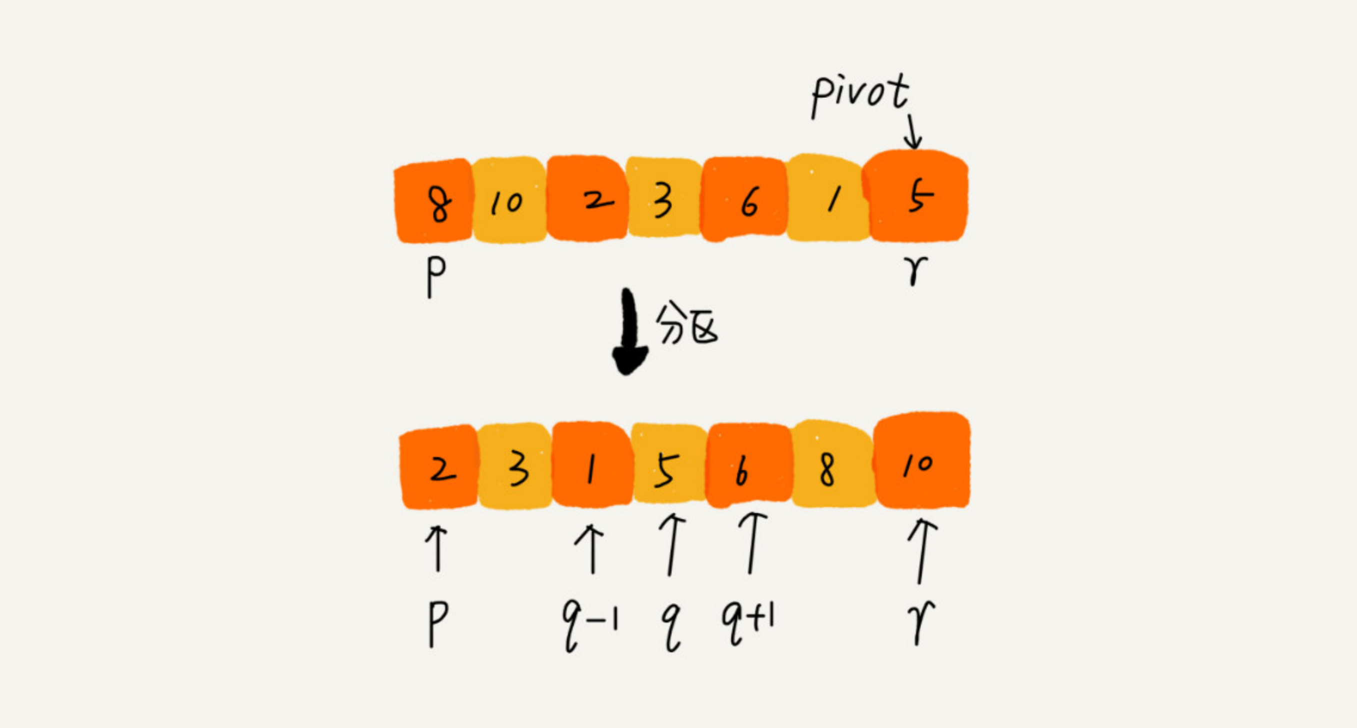
我们可以得到 T(n) = 2^kT(n/2^k)+kn。当 T(n/2^k)=T(1) 时，也就是 n/2^k=1，我们得到 k=log2n 。 我们将 k 值代入上面的公式，得到 T(n)=Cn+nlog2n 。如果我们用大 O 标记法来表示的话，T(n) 就等于 O(nlogn)。

第三、归并排序的空间复杂度是多少 一个致命的“弱点”，那就是归并排序不是原地排序算法。临时内存空间最大也不会超过 n 个数据的大小，所以空间复杂度是 O(n)。

快速排序的原理

快排的思想是这样的：如果在排序数组中下标从p到r之间的一组数据，我们选择p到r之间任意一个数据作为pivot（分区点）

我们遍历p到r之间的数据，将小于pivot的放到左边，将大于pivot的放在右边，将pivot放到中间。 经过这个步骤之后，数组p到r之间的数据就被分成三个部分。



根据分治、递归的处理思想，我们将递归小标从p到q-1之间的数据和下标从q+1到r之间的数据，直至区间缩小到1，就说明所有的数据都有序了

```
递推公式:
quick_sort(p...r) = quick_sort(p...q-1) + quick_sort(q+1...r)

终止条件:
p >= r
```

递归伪代码：

```
//快速排序，A是数组，n是数组大小
quick_sort(A,n){
    quick_sort_c(A, 0, n-1)
}
//快速排序递归函数，p，r为下标
quick_sort_c(A,p,r) {
    if p >= r then return

    q = partition(A,p,r)
    quick_sort_c(A,p,q-1)
    quick_sort_c(A,q+1,r)
}
```

partition() 分区函数。随机选择一个元素作为 pivot（一般情况下，可以选择 p 到 r 区间的最后一个元素），然后对 A[p...r] 分区，函数返回 pivot 的下标。

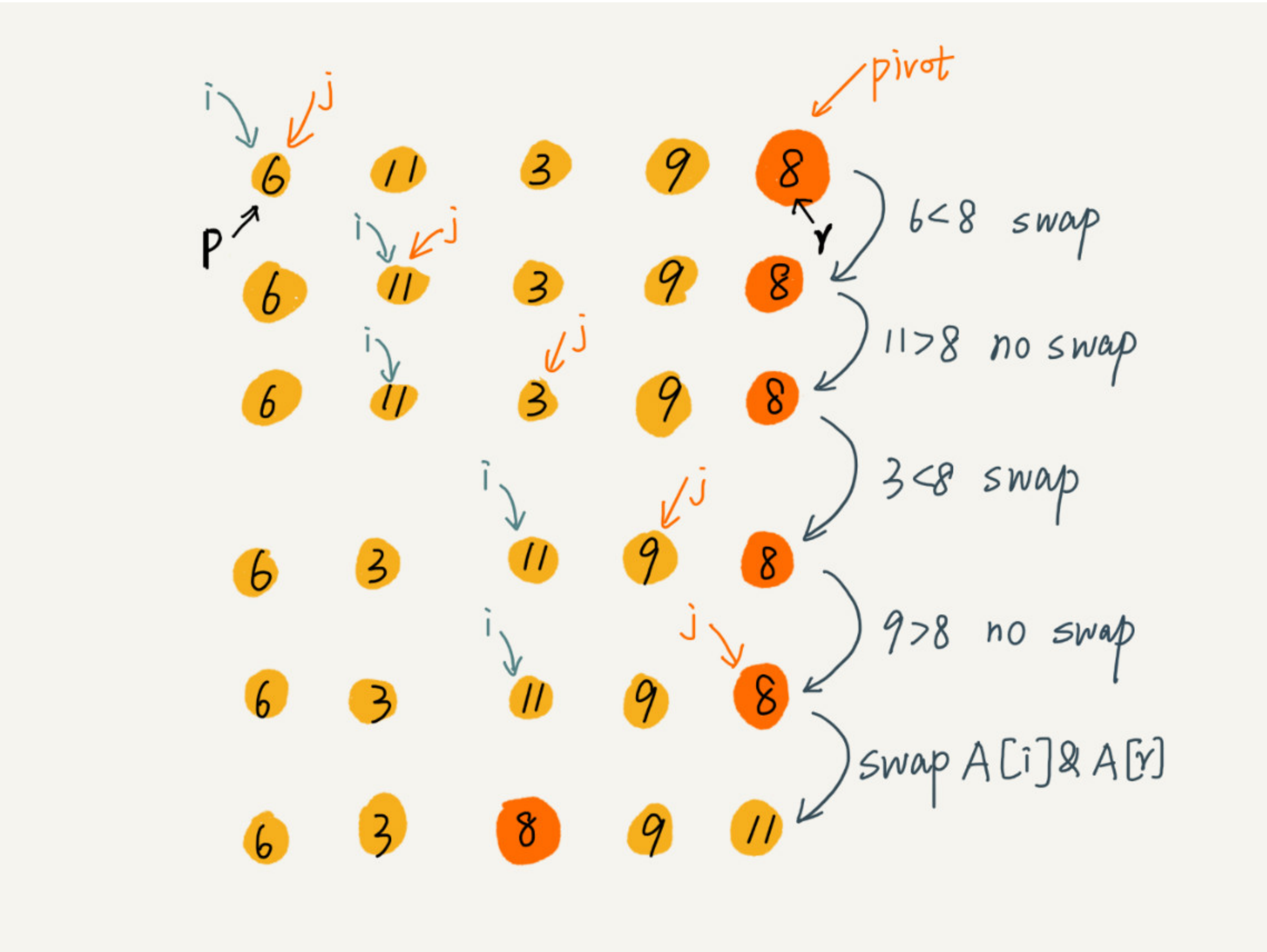
```
partition(A, p, r) {
    pivot := A[r]
    i := p
    for j := p to r-1 do {
        if A[j] < pivot {
        swap A[i] with A[j]
        i := i+1
        }
    }
    swap A[i] with A[r]
return i
```

实现代码：

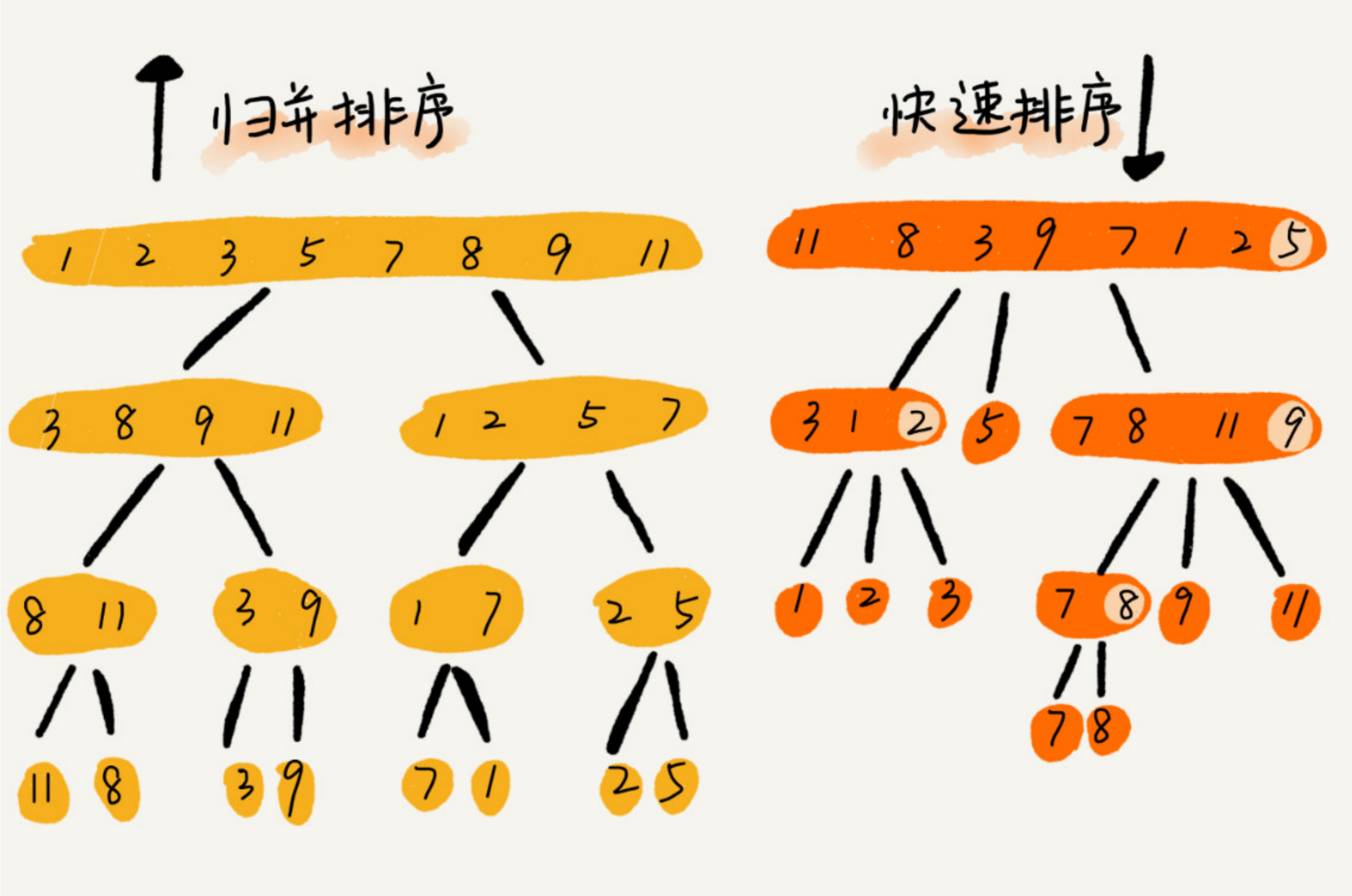
```
/**
 * 快速排序
 * @param a
 */
public static void quickSort(int[] a){
    //调用递归函数
    quickSortC(a,0,a.length-1);
}

/**
 * 递归实现分区过程
 * @param a
 * @param start
 * @param end
 */
public static void quickSortC(int[] a,int start,int end){
    if(start >= end){
        return;
    }
    int mid = partition(a,start,end);
    quickSortC(a,start,mid-1);
    quickSortC(a,mid+1,end);
}

/**
 * 将排序分区
 * @param a
 * @param start
 * @param end
 * @return
 */
public static int partition(int[] a,int start,int end){
    //默认最后一个元素是分界区元素
    int pivot = a[end];
```

```
        //i为开始的元素
        int i = start;
        //j从开始一直到倒数第二个元素
        for(int j = start; j < end;j++){
            //如果a[j]小于分界点元素，将a[j]放到已处理区间末尾，也就是a[i]处，i往后挪
            if(a[j] < pivot){
                int tmp = a[i];
                a[i] = a[j];
                a[j] = tmp;
                i = i+1;
            }
        }
        //循环完毕后，将默认的最后一个元素也就是分界点与a[i]交换
        int tmp = a[i];
        a[i] = a[end];
        a[end] = tmp;
        return i;
    }
```

快排是原地排序算法，具体就是通过游标i将A[p...r-1]分为两部分。A[p...i-1]的元素都是pivot的，我们暂且叫已处理区间，A[i...r-1]是未处理区间。我们每次都从未处理区间中取一个元素A[j]，与pivot对比，如果小于pivot将其加入到已处理区间的尾部，也就是A[i]的位置。只需要将 A[i] 与 A[j] 交换，就可以在 O(1) 时间复杂度内将 A[j] 放到下标为 i 的位置。



分区过程中涉及到交换操作，所以快速排序并不是一个稳定的排序算法。



归并排序的处理过程是由下到上的，先处理子问题，然后再合并。而快排正好相反，它的处理过程是由上到下的，先分区，然后再处理子问题。

归并排序虽然是稳定的，非原地排序算法。 快速排序通过设计巧妙的原地分区函数，可以实现原地排序，解决了归并排序占用太多内存的问题。

快排的性能分析

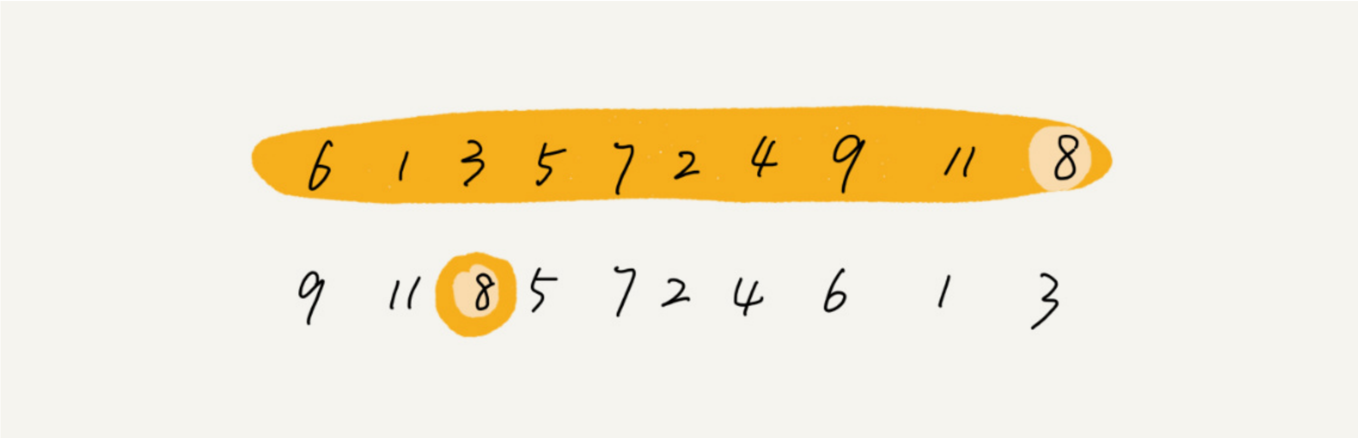
如果每次分区操作，都能正好把数组分成大小接近相等的两个小区间，那快排的时间复杂度递推求解公式跟归并是相同的。所以，快排的时间复杂度也是 O(nlogn)

一个是分区极其均衡，一个是分区极其不均 T(n) 在大部分情况下的时间复杂度都可以做到 O(nlogn)，只有在极端情况下，才会退化到 O(n2)。

解答开篇

求第K大元素 我们选择数组区间 A[0...n-1] 的最后一个元素 A[n-1] 作为 pivot，对数组 A[0...n-1] 原地分区，这样数组就分成了三部分，A[0...p-1]、A[p]、A[p+1...n-1]。

如果 p+1=K，那 A[p] 就是要求解的元素；如果 K>p+1, 说明第 K 大元素出现在 A[p+1...n-1] 区间，我们再按照上面的思路递归地在 A[p+1...n-1] 这个区间内查找。同理，如果 K



第一次分区查找，我们需要对大小为 n 的数组执行分区操作，需要遍历 n 个元素。第二次分区查找，我们只需要对大小为 n/2 的数组执行分区操作，需要遍历 n/2 个元素。依次类推，分区遍历元素的个数分别为、n/2、n/4、n/8、n/16.......直到区间缩小为 1。如果我们把每次分区遍历的元素个数加起来，就是：n+n/2+n/4+n/8+...+1。这是一个等比数列求和，最后的和等于 2n-1。所以，上述解决思路的时间复杂度就为 O(n)。

课后思考

现在你有 10 个接口访问日志文件，每个日志文件大小约 300MB，每个文件里的日志都是按照时间戳从小到大排序的。你希望将这 10 个较小的日志文件，合并为 1 个日志文件，合并之后的日志仍然按照时间戳从小到大排列。如果处理上述排序任务的机器内存只有 1GB，你有什么好的解决思路，能"快速"地将这 10 个日志文件合并吗?

先构建十条io流，分别指向十个文件，每条io流读取对应文件的第一条数据，然后比较时间戳，选择出时间戳最小的那条数据，将其写入一个新的文件，然后指向该时间戳的io流读取下一行数据，然后继续刚才的操作，比较选出最小的时间戳数据，写入新文件，io流读取下一行数据，以此类推，完成文件的合并， 这种处理方式，日志文件有n个数据就要比较n次，每次比较选出一条数据来写入，时间复杂度是O（n），空间复杂度是O（1）,几乎不占用内存，这是我想出的认为最好的操作了。

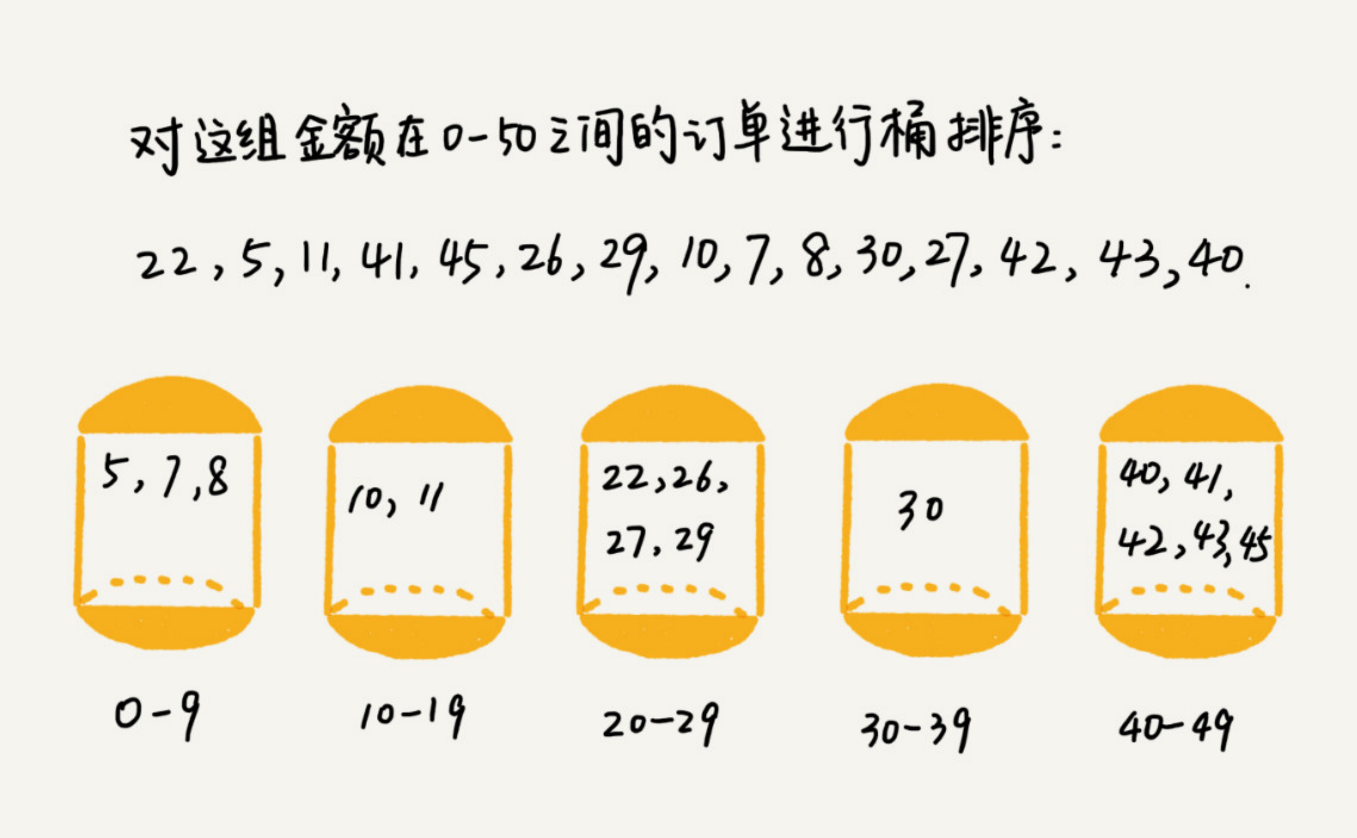
线性排序

三种时间复杂度是O(n)的排序算法：桶排序、计数排序、基数排序，这些排序算法复杂度是线性的，所以称之为线性排序。之所以可以做到线性排序，主要原因是非基于比较的排序算法。

重点是掌握这些排序算法的适用场景； 如何根据年龄给100万用户排序?

桶排序（Bucket sort)

核心思想是将要排序的数据分到几个有序的桶里，每个桶里的数据单独排序，桶内排完之后，再把每个桶里的数据按照顺序依次取出，组成的序列就是有序的了。



桶排序的时间复杂度是O(n)，如果排序的数据有n个，我们把它们均匀的划分到m个桶内，每个桶里就有k=n/m个元素。每个桶内使用快速排序，时间复杂度为O(k * logk) 。m个桶内时间表复杂度就是O(m * k * logk)，又因为k=n/m，所以整个桶排序的时间就变成了O(n*log(n/m)),当桶的个数m接近个数n时，log(n/m)就是一个非常小的常量，这个时候桶排序的时间复杂度接近O(n)。

桶排序看起来很优秀，那它是不是可以替代我们之前讲的排序算法呢？ 不可以，桶排序对要排序数据的要求是非常苛刻的 首先数据要很容易划分为m个桶，并且桶与桶之间有着天然的大小顺序，其次数据在各个桶之间分布式比较均匀的。 桶排序比较适合在外部排序中，所谓外部排序就是数据存储在外部磁盘中，数据量比较大，内存有限，无法将数据全部加载到内存中。

代码实现

```
/**
 * 桶排序
 * @param a
 * @param bucketSize
 */
public static void bucketSort(int[] a, int bucketSize){
    if(a.length <= 1){
        return;
    }
    //寻找最大最小值
    int min = a[0];
    int max = a[0];
    for (int i = 0; i < a.length;i++){
        if(min > a[i]){
            min = a[i];
```

```
        }else if(max < a[i]){
            max = a[i];
        }
    }
    //计算桶间隔
    int bucketCount = (max - min) / bucketSize + 1;
    //创建二维数组做桶
    int[][] buckets = new int[bucketCount][bucketSize];
    //存储每个桶中元素个数
    int[] indexArr = new int[bucketCount];

    //遍历所有数据将数据放入不同的桶中
    for(int i = 0; i < a.length;i++){
        int bucketIndex = (a[i] - min) / bucketSize;
        //扩容
        if(indexArr[bucketIndex] == buckets[bucketIndex].length){
            ensureCapacity(buckets,bucketIndex);
        }
        buckets[bucketIndex][indexArr[bucketIndex]++] = a[i];
    }

    int k = 0;
    for(int i = 0; i < buckets.length;i++) {
        if (indexArr[i] == 0) {
            continue;
        }
        //桶内元素快排
        quickSortC(buckets[i], 0, indexArr[i] - 1);
        //元素复制到原数组中
        for (int j = 0; j < indexArr[i]; j++) {
            a[k++] = buckets[i][j];
        }
    }
}

/**
 * 数组扩容
 * @param buckets
 * @param bucketIndex
 */
public static void ensureCapacity(int[][] buckets,int bucketIndex){
    int[] tempArr = buckets[bucketIndex];
    int[] newArr = new int[tempArr.length * 2];
    for (int j = 0; j < tempArr.length; j++) {
        newArr[j] = tempArr[j];
    }
    buckets[bucketIndex] = newArr;
}
```

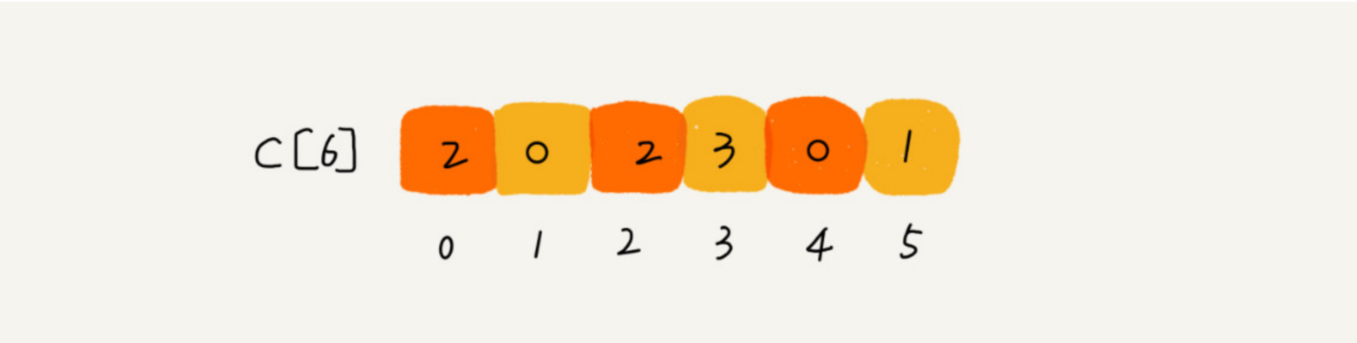
计数排序

其实计数排序应该是桶排序的一种特殊情况。 当要排序的 n 个数据，所处的范围并不大的时候，比如最大值是 k，我们就可以把数据划分成 k 个桶。每个桶内的数据值都是相同的，省掉了桶内排序的时间。

考生的满分是 900 分，最小是 0 分，这个数据的范围很小，所以我们可以分成 901 个桶，对应分数从 0 分到 900 分。根据考生的成绩，我们将这 50 万考生划分到这 901 个桶里。桶内的数据都是分数相同的考生，所以并不需要再进行排序。我们只需要依次扫描每个桶，将桶内的考生依次输出到一个数组中，就实现了 50 万考生的排序。因为只涉及扫描遍历操作，所以时间复杂度是 O(n)。

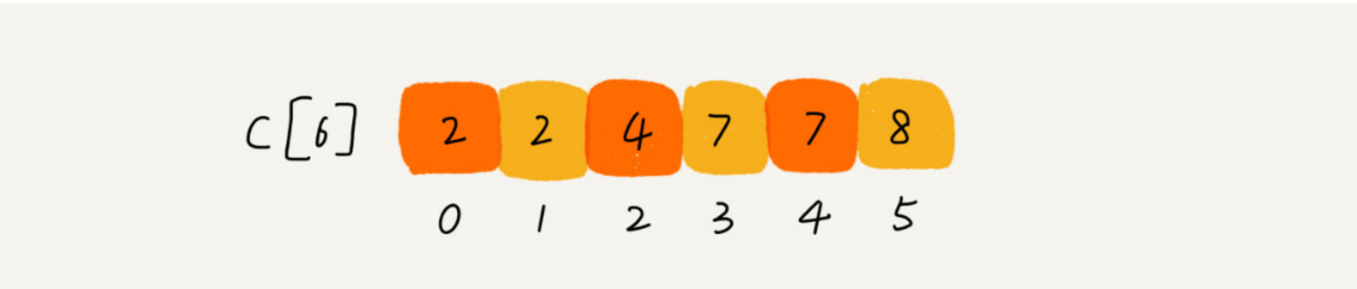
为什么这个排序算法叫“计数”排序呢？“计数”的含义来自哪里呢？

假设8个考生，分数在0到5之间，这8个考生成绩放在一个数组A[8]中，分别是：2,5,3,0,2,3,0,3。 我们使用C[6]来表示桶，其中下标表示对应分数，C[6]内存储的并不是考生，而是对应的考生个数。 则C[6]的值为：



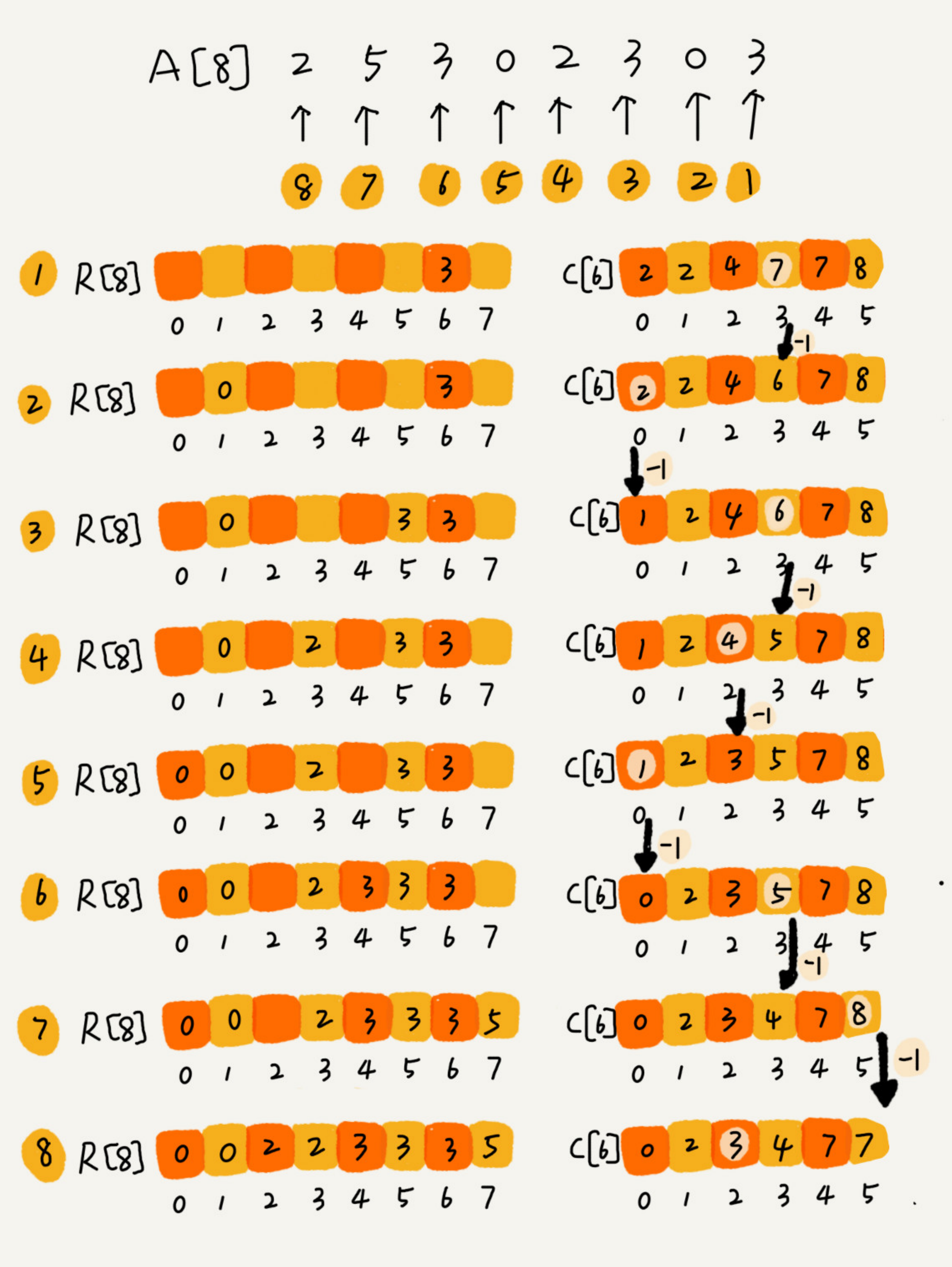
从图中可以看出，分数为 3 分的考生有 3 个，小于 3 分的考生有 4 个，所以，成绩为 3 分的考生在排序之后的有序数组 R[8] 中，会保存下标 4，5，6 的位置。

如何快速计算出，每个分数考生对应的存储位置呢？ 我们对C[6]顺序求和，C[6] 存储的数据就变成了下面这样子。C[k] 里存储小于等于分数 k 的考生个数。



我们从后往前（可以保证稳定性）扫描数组A，当扫描到3的时候，我们可以从数组C中去除下标是3 的值7，到目前为止，分数小于等于3的考生有7个，也就是说3是数组R中第7个元素，当3放入数组R中，小于等于3的元素

就剩6个了，所以C[3]的值减一，变成6。以此类推。



代码实现：

```
/**
 * 计数排序
 * 可以对数据进行变换，数据只能是非负整数
 * @param a
 */
public static void countingSort(int[] a){
    if(a.length <= 1){
        return;
    }
    //寻找数据范围
    int max = a[0];
    for(int i = 0; i < a.length; i++){
        if(max < a[i]){
            max = a[i];
        }
    }
    //申请一个数组c，小标是0~max
    int[] c = new int[max + 1];
    //计算每个元素的个数放入到c中
    for(int i = 0; i < a.length;i++){
        c[a[i]]++;
    }
    //依次累加
    for(int i = 1; i < c.length;i++){
        c[i] = c[i-1] + c[i];
    }
    //申请临时数组
    int[] r = new int[a.length];
    //计数排序，从后往前可以保证稳定性
    for(int i = a.length-1; i >= 0;i--){
        r[c[a[i]]-1] = a[i];
        c[a[i]]--;
    }
    //结果拷贝给数组a
    for(int i = 0; i < a.length; i++){
        a[i] = r[i];
    }
}
```

总结

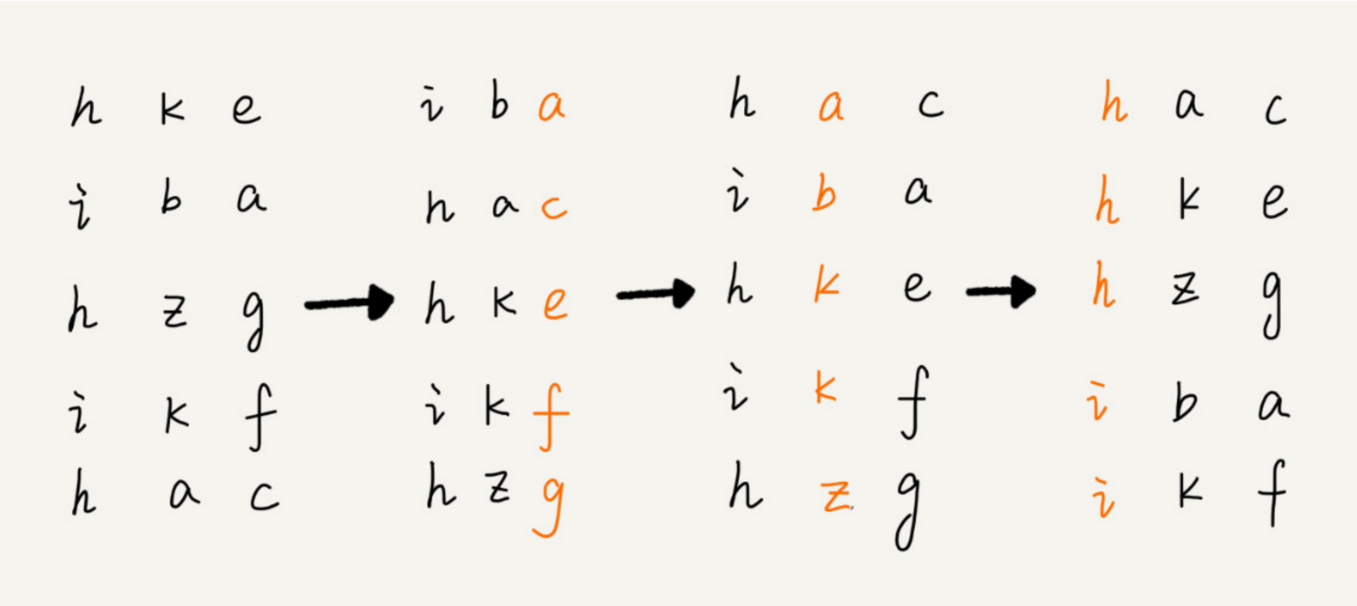
计数排序只能用在数据范围不大的场景中，如果数据范围k比要排序的数据n大很多，就不适合计数排序了。而且计数排序只能给非负整数排序，如果要排序的数据是其他类型的要将不改变其相对大小的情况下，转化为非负整数。

基数排序

假设我们有 10 万个手机号码，希望将这 10 万个手机号码从小到大排序，你有什么比较快速的排序方法呢？

假设要比较两个手机号码 a，b 的大小，如果在前面几位中，a 手机号码已经比 b 手机号码大了，那后面的几位就不用看了

借助稳定排序算法，先按照最后一位来排序手机号码，然后再按照倒数第二位重新排序，以此类推最后按照第一位重新排序，经过11次排序之后，手机号码就都有序了。



根据每一位来排序，我们可以用刚讲过的桶排序或者计数排序，它们的时间复杂度可以做到 O(n)。如果要排序的数据有 k 位，那我们就需要 k 次桶排序或者计数排序，总的时间复杂度是 O(k*n)。当 k 不大的时候，比如手机号码排序的例子，k 最大就是 11，所以基数排序的时间复杂度就近似于 O(n)。

单词不等长，就将所有的单词补齐到相同长度，位数不够补0。

基数排序对要排序的数据是有要求的，需要可以分割出独立的“位”来比较，而且位之间有递进的关系，如果 a 数据的高位比 b 数据大，那剩下的低位就不用比较了。除此之外，每一位的数据范围不能太大，要可以用线性排序算法来排序，否则，基数排序的时间复杂度就无法做到 O(n) 了。

代码实现

```
/**
 * 基数排序
 * @param arr
 */
public static void radixSort(int[] arr) {
    int max = arr[0];
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }

    // 从个位开始，对数组arr按"指数"进行排序
    for (int exp = 1; max / exp > 0; exp *= 10) {
        countingSort(arr, exp);
    }
```

```
    }
    /**
     * 变化的计数排序
     * @param arr
     * @param exp
     */
    public static void countingSort(int[] arr, int exp) {
        if (arr.length <= 1) {
            return;
        }

        // 计算每个元素的个数
        int[] c = new int[10];
        for (int i = 0; i < arr.length; i++) {
            c[(arr[i] / exp) % 10]++;
        }

        // 计算排序后的位置
        for (int i = 1; i < c.length; i++) {
            c[i] += c[i - 1];
        }

        // 临时数组r，存储排序之后的结果
        int[] r = new int[arr.length];
        for (int i = arr.length - 1; i >= 0; i--) {
            r[c[(arr[i] / exp) % 10] - 1] = arr[i];
            c[(arr[i] / exp) % 10]--;
        }

        for (int i = 0; i < arr.length; i++) {
            arr[i] = r[i];
        }
    }
```

解答开篇

根据年龄给100万用户排序，类似50万考生排序，假设年龄范围是1到120岁。遍历120万用户，将年龄划分到这120个桶内，一次遍历120个桶内的元素，就得到了按照年龄排序的120万数据。

如果数据特征比较符合这些桶排序、计数排序、基数排序算法的要求，应用这些算法，会非常高效，线性时间复杂度可以达到 O(n)。

课后思考

假设我们现在需要对 D，a，F，B，c，A，z 这个字符串进行排序，要求将其中所有小写字母都排在大写字母的前面，但小写字母内部和大写字母内部不要求有序。比如经过排序之后为 a，c，z，D，F，B，A，这个如何来实现呢？如果字符串中存储的不仅有大小写字母，还有数字。要将小写字母的放到前面，大写字母放在最后，数字放在中间，不用排序算法，又该怎么解决呢？

利用桶排序思想，弄小写，大写，数字三个桶，遍历一遍，都放进去，然后再从桶中取出来就行了。复杂度O(n)

排序优化：如何实现一个通用的高性能的排序函数

排序函数如何实现的，底层是什么排序算法?

如何实现一个通用的、高性能的排序函数

如何选择合适的排序算法

	时间复杂度	是稳定排序?	是原地排序?
冒泡排序	$O(n^2)$	✓	✓
插入排序	$O(n^2)$	✓	✓
选择排序	$O(n^2)$	×	✓
快速排序	$O(nlogn)$	×	✓
归并排序	$O(nlogn)$	✓	×
计数排序	$O(n+k)$ k是数据范围	✓	×
桶排序	$O(n)$	✓	×
基数排序	$O(dn)$ d是维度	✓	×

线性排序算法时间复杂度低，但使用场景比较特殊，写一个通用的排序函数，不能选择线性排序算法。

小规模数据排序，可以选择时间复杂度是O(n^2)的算法，如果是大规模数据进行排序，时间复杂度是O(nlogn)的算法更有效。为了兼顾任意规模，一般都会首选时间复杂度是O(nlogn)的排序算法。

堆排序和快速排序都有比较多的应用，java语言采用堆排序实现排序函数，C语言使用快速排序实现排序函数。

如何优化快速排序

快速排序的糟糕主要是因为分区点的选择不合理造成的。

最理想的分区点是：被分区点分开的两个分区中，数据的数量差不多。

1、三数取中法 从区间的首、尾、中间，分别取出一个数，然后对比大小，取这三个数的中间值作为分区点。如果排序数组比较大，那就五数取中或者十数取中。

2、随机法 随机法就是每次从要排序的区间中，随机选择一个元素作为分区点。不能保证每次分区都是好的，但概率是不大可能每次都是很差的。

举例说明排序函数

拿 Glibc 中的 qsort() 函数举例说明一下； qsort() 会优先使用归并排序来排序输入数据，因为归并排序的空间复杂度是 O(n)，所以对于小数据量的排序，比如 1KB、2KB 等，归并排序额外需要 1KB、2KB 的内存空间，这个问题不大。

要排序的数据量比较大的时候，qsort() 会改为用快速排序算法来排序。

qsort() 选择分区点的方法就是“三数取中法”

递归太深会导致堆栈溢出的问题，qsort() 是通过自己实现一个堆上的栈，手动模拟递归来解决的

在快速排序的过程中，当要排序的区间中，元素的个数小于等于 4 时，qsort() 就退化为插入排序，不再继续用递归来做快速排序，因为我们前面也讲过，在小规模数据面前，O(n2) 时间复杂度的算法并不一定比 O(nlogn) 的算法执行时间长。

时间复杂度代表的是一个增长趋势，在小规模数据时候，低阶，系数，常数之间对时间的影响还是很大的。