数据结构和算法（二）

1：今日课程内容介绍(了解)

在第一天的课程中我们学习了四个基本的数据结构：数组，链表，栈，队列。针对java 中已有的实现进行了源码分析，并且我们也手动实现栈和队列这两种数据结构。在今天的课程中我们主要学习一些算法，主要包括三大类算法：**递归，排序，二分查找**。

2：今日课程目标(了解)

1：掌握算法的时间复杂度和空间复杂度的分析方法，掌握常见算法的时间复杂度。

2：理解并掌握递归算法的思想，针对递归类型的问题能够找到递推公式并写出递归的代码。

3：理解掌握常见排序算法的原理，手动实现每一个排序算法，能够分析每一个排序算法的时间复杂度和空间复杂度

4：理解并掌握二分查找的原理，能够用代码实现二分查找，分析二分查找的复杂度，掌握二分查找的实际应用场景。

3：复杂度分析(重要)

数据结构和算法解决的是“快”和“省”的问题，即如何让代码运行的更快，如何让代码更省存储空间。因此代码的执行效率是一个非常重要的考量指标，那如何来衡量代码的执行效率呢？，我们可以用时间复杂度，空间复杂度来对代码的代码的执行效率，性能进行评估，也就是算法的复杂度分析。

算法的复杂度分析主要包含两个方面：

- 时间复杂度分析
- 空间复杂度分析

为什么要进行复杂度分析？

1：和性能测试相比，复杂度分析有不依赖执行环境、成本低、效率高、易操作、指导性强的特点。

2：掌握复杂度分析，将能编写出性能更优的代码，有利于降低系统开发和维护成本。

3.1：时间复杂度表示法(理解)

算法的执行效率，粗略地讲，就是算法代码执行的时间，那如何在不直接运行代码的前提下粗略的计算执行时间呢？

先来看一段简短的代码，求：1，2，3，4……n 累加和

```
int sum(int n) {
  int sum = 0; //执行一遍
  for ( int i = 1; i <= n; ++i) {//执行 n 遍
    sum = sum + i;//执行 n 遍
  }
  return sum;
}
```

假设每行代码执行时间都一样为：timer，那此代码的执行时间为多少呢：(2n+1)*timer，由此可以看出来，所有代码的执行时间 T(n)与代码的执行次数成正比。

按照该思路我们接着看下面一段代码

```
int sum(int n) {
  int sum = 0;//执行一遍
  for (int i=1; i <= n; ++i) { //执行 n 遍
    for (int j=1; j <= n; ++j) { //执行 n*n 遍
      sum = sum +  i * j;//执行 n*n 遍
    }
  }
}
```

同理，此代码的执行时间为：(2n*n+n+1) * timer

因此有一个重要结论：**代码的执行时间 T(n)与每行代码的执行次数 n 成正比**，我们可以把这个规律总结成一个公式。

```
T(n) = O(f(n))
```

解释一下：T(n)表示代码的执行时间，n 表示数据规模的大小，f(n)表示了代码执行次数的总和，它是一个公式因此用 f(n)表示，O 表示了代码执行时间与 f(n)成正比

因此第一个例子中的 T(n)=O(2n+1)，第二个例子中的 T(n)=O(2n*n+n+1)，这就是**大 O 时间复杂度表示法**

大 O 时间复杂度实际上并不具体表示代码真正的执行时间，而是**表示代码执行时间随数据规模增长的变化趋势**，所以，也叫作渐进时间复杂度，**简称时间复杂度**。

当 n 很大时，公式中的低阶，常量，系数三部分并不左右其增长趋势，因此可以忽略，我们只需要记录一个最大的量级就可以了，因此如果用大 O 表示刚刚的时间复杂度可以记录为：T(n)=O(n),T(n)=O(n*n)

3.2：时间复杂度分析方法(理解并掌握)

如何计算程序（算法）时间复杂度的问题：

1：代码循环次数最多原则

我们刚刚讲到,大 O 复杂度表示法只代表一种变化趋势，公式中的低阶，常量，系数三部分并不左右其增长趋势，因此可以忽略，我们只需要记录一个最大的量级就可以了。因此我们在***分析一个算法或者一个代码的时间复杂度时，只需关注循环执行次数最多的那一段代码即可***。

```
int test(int n){
      int sum=0;
      int i=0;
      for (;i<n;i++){
            sum + = i;  // 循环内执行次数最多，n 次 因此 这段程序的时间复杂度记为O(n)
      }
      return sum;
}
```

2：加法原则

比如有如下代码：

```
int sum(int n) {
    // 常量级：忽略
   int sum_1 = 0;
   int p = 1;
    // 因为值循环了100 次，属于常量级：忽略
   for (; p <= 100; ++p) {
     sum_1 = sum_1 + p;
   }
  // 常量级：忽略
   int sum_2 = 0;
   int q = 1;

    //循环n 次，时间复杂度为：O(n)
   for (; q < n; ++q) {
     sum_2 = sum_2 + q;
   }
// 常量级：忽略
   int sum_3 = 0;
```

```
    int i = 1;
    int j = 1;
     //嵌套循环，时间复杂度为：O(n*n)
    for (; i <= n; ++i) {
      j = 1;
      for (; j <= n; ++j) {
        sum_3 = sum_3 +  i * j;
      }
    }
    return sum_1 + sum_2 + sum_3;
 }
```

其中两段最大量级的复杂度分别为 O(n)和 O(n*n)，其结果本应该是：
T(n)=O(n)+O(n * n)，我们取其中最大的量级，因此整段代码的复杂度为：O(n * n)

也就是说：***总的时间复杂度就等于量级最大的那段代码的时间复杂度***

3：乘法原则

嵌套代码的复杂度等于嵌套内外代码复杂度的乘积，举个例子

```
int sum(int n) {
    int ret = 0;
    int i = 1;
     //单独看是:O(n),由于func(i)是O(n)因此整体是:O(n) * O(n) = O(n*n) = O
(n*n)
    for (; i < n; ++i) {
      ret = ret + func(i);//f(i)是O(n)
    }
 }
// O(n)
 int func(int n) {
  int sum = 0;
  int i = 1;
  for (; i < n; ++i) {
    sum = sum + i;
  }
  return sum;
 }
```

因此可以看出：***嵌套代码的复杂度等于嵌套内外代码复杂度的乘积***

3.3：常见的时间复杂度(理解并掌握)

虽然代码写法千差万别，但是我们平常所见的复杂度量级并不多，列举如下

3.3.1：O(1)

首先要明确，O(1)并不是指代码只有一行，它是一种常量级复杂度的表示方法，比如说有一段代码如下：

```
public void test01(){
    int i=0;
    int j = 1;
    return i+j;
}
```

代码只有三行，它的复杂度也是 O(1)，而不是 O(3),再看如下代码：

```
public void test02(){
    int i=0;
    int sum=0;
    for(;i<100;i++){
        sum = sum+i;
    }
    System.out.println(sum);
}
```

整个代码中因为循环次数是固定的就是 100 次，这样的代码复杂度我们认为也是 O(1),

因此总结下来就是：只要代码的执行时间不随着 n 的增大而增大，这样的代码复杂度都是 O(1)，或者说：只要在算法中不存在递归语句，随 n 变化的循环语句等，即使有千万行代码，复杂度也是 O(1)

3.3.2：O(n)

这种复杂度量级随处可见，比如我们分析如下代码：

```
public void test03(int n){
    int i=0;
    int sum=0;
    for(;i<n;i++){
        sum = sum+i;
    }
    System.out.println(sum);
}
```

所以我们不在赘述。

3.3.3：O(logn)，O(nlogn)

对数阶的复杂度非常的常见，但同时也是很难分析的一种复杂度，如下代码：

```
public void test04(int n){
    int i=1;
    while(i<=n){
        i = i * 2;
    }
}
```

根据之前的经验，第四行代码：i = i*2 循环次数最多，我们也知道时间复杂度就是代码的执行次数(时间)随着数据量的变化，因此我们要分析出第四行代码执行了多少次我们就得出了此代码的时间复杂度。

那第四行代码执行了多少次呢？代码执行第一次 i=2，执行第二次 i=4，执行第三次 i=8.....，i 的取值其实是一个等比数列的形式，如图

由图中分析可知，代码的时间复杂度表示为

那如果我将代码进行修改，改为如下：

```
public void test05(int n){
    int i=1;
    while(i<=n){
        i = i * 3;
    }
}
```

通过刚刚的思路我们马上就能分析出来：这段代码的时间复杂度为：但是实际上不管是以 2 为底还是以 3 为底，或者以 10 为底，我们把所有的对数阶的时间复杂度都记为: 为什么呢？

我们知道对数有一个换底公式：，因此,而以 3 为底，2 的对数是一个常量系数，基于我们前面的讨论，使用大 O 标记时间复杂度时不考虑低阶，系数，常量，所以在对数阶时间复杂度中我们忽略对数的底统一表示为：

那分析完我们讲解的 O(log n)，那 O(n * log n)就很容易理解了，比如下列代码：

```
public void test06(int n){
    int i=0;
    for(;i<=n;i++){
        test04(n);
    }
}
public void test07(int n){
    int i=1;
    while(i<=n){
        i = i * 2;
    }
}
```

这是一种非常常见的算法时间复杂度，我们马上要学习的归并排序，快速排序的时间复杂度都是 O(nlogn)

3.4：最好/最坏/平均时间复杂度分析(理解并掌握)

3.4.1：最好/最坏复杂度

有一个需求：在数组 array 中查找变量 x 的位置，实现如下：

```
//其中n 表示数组 array 的长度
public int getX(int[] array, int n, int x) {
  int i = 0;
  int pos = -1;
  for (; i < n; ++i) {
      if (array[i] == x) {
            pos = i;
      }
  }
  return pos;
}
```

通过之前学习的复杂度分析方式，我们可以分析出来这段代码的复杂度为 O(n)，但是这段代码实现的并不是特别好，我们稍作优化，因为在数组中查找某一元素并不需要把整个数组都遍历一遍，因为可能在遍历的途中就已经找到了，找到了就直接返回，提前结束循环，优化后的结果如下：

```
//其中n 表示数组 array 的长度
public int getX(int[] array, int n, int x) {
  int i = 0;
  int pos = -1;
  for (; i < n; ++i) {
      if (array[i] == x) {
            pos = i;
        break;
      }
  }
  return pos;
}
```

优化完成之后我们再看这段代码的复杂度还是 O(n)吗？

要查找的变量 x 在可能在数组中的任意位置，如果数组中的第一个元素就是我们要找的变量 x，那就不需要继续变量余下的 n-1 个元素了，那复杂度为 O(1),如果数组中不存在变量 x 那需要完整的遍历一遍数组，那复杂度就是 O(n)。所以：在不同的情况下，同一段代码的复杂度并不一样

因此我们需要引入三个概念：**最好情况复杂度，最坏情况复杂度，和平均情况复杂度**

最好情况复杂度：在最理想的情况下代码的时间复杂度

最坏情况复杂度：在最糟糕的情况下代码的时间复杂度

3.4.2：平均情况时间复杂度

我们知道最好或者最坏情况复杂度分别对应了两种极端的情况，发生的概率并不大，为了更好的表示平均情况下的复杂度，我们引入平均情况复杂度。

那刚刚的代码如何分析评价情况复杂度呢？

```
//其中n 表示数组 array 的长度
public int getX(int[] array, int n, int x) {
  int i = 0;
  int pos = -1;
  for (; i < n; ++i) {
      if (array[i] == x) {
            pos = i;
        break;
      }
  }
  return pos;
}
```

查找变量 x 在数组中的位置有 n+1 中情况，在数组中 0~n-1 的位置上以及不在数组中，我们把每一种情况下代码需要遍历的次数加起来，然后除以 n+1 就得到了代码需要遍历执行的平均值。如图：

我们知道，在大 O 复杂度标记法中，可以省略系数，低阶，常量，因此把该结果简化之后得到的复杂度仍然为 O(n)

此时，结论虽然正确，但是计算过程稍微有点儿问题，问题出在哪里？就是我们刚刚讲到的这 n+1 种情况出现的概率并不是一样的，

通过之前的讲解我们知道查找变量 x 在数组中的位置要么在数组中，要么不在数组中，意味着在数组中 0~n-1 位置上出现的概率为 1/2，不再数组中出现的概率为 1/2,

此外在数组中 0~n-1 位置上每一种情况下的概率为 1/n，因此根据概率法则，要查找的数据出现在 0 ~ n-1 中任意位置的概率为 1/2n

所以前面的结论推导过程中我们需要将概率考虑进去，那计算平均时间复杂度的计算过程就变成了这样：

这个值在概率论中叫**加权平均值，也叫做期望值，所以平均时间复杂度也叫期望时间复杂度**，同样的道理如果用大 O 表示法来表示，加权时间复杂度仍然为：O(n)

实际上：大多数情况下，我们不需要去区分最好/最坏/平均 时间复杂度，只有在同一段代码块在不同情况下的复杂度有量级的差距时才需要用这三种复杂度来加以区分

3.5：空间复杂度(理解)

前面我们讲过，**时间复杂度的全称是渐进时间复杂度，表示算法的执行时间与数据规模之间的增长关系**，类比一下，**空间复杂度全称是渐进空间复杂度，表示算法占用的存储空间与数据规模之间的增长关系**

用一段小的代码来说明一下：

```
void print(int n) {
  int i = 0;
  int[] a = new int[n];
  for (i; i <n; ++i) {
    a[i] = i * i;
  }
  for (i = n-1; i >= 0; --i) {
    System.out.println(a[i]);
  }
}
```

代码第二行，申请一个空间存储变量 i，但是是常量阶的，跟数据规模 n 没有关系，第三行申请了一个大小为 n 的 int 数组，此外后面的代码几乎没有占用更多的空间，因此整段代码的空间复杂度就是 O(n)

我们常见的空间复杂度就是 O(1),O(n),O(n * n)，其他像对数阶的复杂度几乎用不到，因此空间复杂度比时间复杂度分析要简单的多。因此掌握这些足够。

3.5：小结

这一小节我们主要讲的是复杂度的分析，学完这一节内容我们掌握了复杂度分析的三个原则，掌握了常见的几个复杂度的量级，并且我们能够针对某一个具体的算法进行复杂度的分析，以此来判断该算法执行的性能和效率。

4：递归算法(重要)

4.1：递归的概念(理解)

递归(Recursion)是一种非常广泛的算法，与其说递归是一种算法不如说递归是一种编程技巧，我个人更倾向于将递归理解为一种编程技巧。在后续数据结构和算法的编码实现过程中我们都要用到递归，比如 DFS 深度优先搜索，前中后序二叉树

的遍历等都需要用到递归的知识，因此搞懂递归非常的重要，否则后续的学习会非常的吃力。

那递归到底是什么？怎么理解递归呢？为了便于解释这里我分别举三个例子来帮助大家进行理解

例 1：我们平时去查词典，这个过程本身就是一种递归，为了解释一个词，需要使用更多的词。当你查一个词，发现这个词的解释中某个词仍然不懂，于是你开始查这第二个词，可惜，第二个词里仍然有不懂的词，于是查第三个词，这样查下去，直到有一个词的解释是你完全能看懂的，那么递归走到了尽头，然后你开始后退，逐个明白之前查过的每一个词，最终，你明白了最开始那个词的意思。

例 2：假设你在一个电影院，你想知道自己坐在哪一排，但是前面人很多，你懒得去数了，于是你问前一排的人"你坐在哪一排?"，这样前面的人 (代号 A) 回答你以后，你就知道自己在哪一排了,你只要把 A 的答案加一，就是自己所在的排了。不料 A 比你还懒，他也不想数，于是他也问他前面的人 B “你坐在哪一排?"，这样 A 可以用和你一模一样的步骤知道自己所在的排。然后 B 也如法炮制。直到他们这一串人问到了最前面的一排，第一排的人告诉问问题的人"我在第一排"。最后大家就都知道自己在哪一排了。

例 3：天下有奇族人姓计，长生不老。一日其孙问其父：吾之 18 代祖名何？

其父不明，父问其父
其父不明，父问其父
其父不明，父问其父
其父不明，父问其父
...
晌后，其 18 代祖回其子：你猜
然其回其子：你猜
然其回其子：你猜
然其回其子：你猜
然其回其子：你猜
.....
终，计姓末代孙知其 18 代祖名“你猜”

通过以上这三个小例子，我想大家大概明白什么叫递归了，递归在维基百科的官方解释为：**递归(Recursion),又名递回，**在数学与计算机科学中，是指在函数的定义中使用函数自身的方法。英文 Recursion 也就是重复发生，再次重现的意思。 而对应的中文翻译 ”递归“ 却表达了两个意思：”递“+”归“。 这两个意思，正是递归思想的精华所在，去的过程叫做递，回来的过程叫做归。在编程语言中对递归可以简单理解为：方法自己调用自己，只不过每次调用时参数不同而已。

基本上所有的递归问题都可以使用**递推公式**来表示，而所谓的递推公式就是将一个递归问题的规律用数学公式的形式表示出来。比如说对于刚刚例 2 电影院的例子，我们可以使用递推公式表示出来是这个样子的：

```
f(n) = f(n-1)+1,其中 f(1)=1
```

f(n) 表示你想知道自己在哪一排，f(n-1) 表示前面一排所在的排数，f(1)=1 表示第一排的人知道自己在第一排。有了这个递推公式，我们就可以很轻松地将例 2 的问题使用递归的代码进行实现，如下：

```
int f(int n) {
    if (n == 1){
        return 1;
    }
      return f(n-1) + 1;
}
```

4.2：满足递归的条件(掌握)

我们之前列举的这三个例子都是非常典型的递归，那究竟什么样的问题我们可以使用递归来解决呢？为此我总结出来两个条件，需要同时满足这两个条件，就可以使用递归来解决。

条件 1：递归表达式(规律)

如果一个问题的解能够拆分成多个子问题的解，拆分之后，子问题和该问题在求解上除了数据规模不一样之外，求解的思路和该问题的求解思路完全相同，也就是说能够找到一种规律，这种规律就是我们说的递推公式，那么这个问题就可以使用递归来求解。

比如之前的例 2 关于电影院座位的例子，你要知道“自己在哪一排”的问题，可以分解为“前一排的人在哪一排”这样一个子问题，你求解“自己在哪一排”的思路，和前面一排人求解“自己在哪一排”的思路，是一模一样的。

条件 2：终止递归的条件(递归出口)

把一个问题的解分解为多个子问题的节，把子问题再分解为子子问题，一层一层分解下去，不能存在无限循环，这就需要有终止条件。就比如电影院的例子，第一排的人不需要再继续询问任何人，就知道自己在哪一排，也就是 f(1)=1，这就是递归的终止条件。

综上所述：写递归代码的关键就是找到如何将一个问题拆分成多个小问题的规律，并且基于此写出递推公式，然后再找到递归终止条件，最后将递推公式和终止条件翻译成代码即可

刚讲的电影院的例子，我们的递归调用只有一个分支，也就是说“一个问题只需要分解为一个子问题”，我们很容易能够想清楚“递“和”归”的每一个步骤，所以写起

来、理解起来都不难。但是，当我们面对的是一个问题要分解为多个子问题的情况，递归代码就没那么好理解了，在有些情况下人脑几乎没办法把整个“递”和“归”的过程一步一步都想清楚。

但是对于计算机而言，它处理起来递归就非常的容易了，它非常喜欢做这种重复的事情。而我们人脑更喜欢平铺直叙的思维方式。当我们看到递归时，我们总想把递归平铺展开，脑子里就会循环，一层一层往下调，然后再一层一层返回，试图想搞清楚计算机每一步都是怎么执行的，这样就很容易被绕进去。对于递归代码，这种试图想清楚整个递和归过程的做法，实际上是进入了一个思维误区。很多时候，我们理解起来比较吃力，主要原因就是自己给自己制造了这种理解障碍。那正确的思维方式应该是怎样的呢？

如果一个问题 A 可以分解为若干子问题 B、C、D，你可以假设子问题 B、C、D 已经解决，在此基础上思考如何解决问题 A。所以你只需要思考问题 A 与子问题 B、C、D 两层之间的关系即可，不需要一层一层往下思考子问题与子子问题，子子问题与子子子问题之间的关系。屏蔽掉递归细节，这样子理解起来就简单多了。因此，编写递归代码的关键是，只要遇到递归，我们就把它抽象成一个递推公式，不用想一层层的调用关系，不要试图用人脑去分解递归的每个步骤。

4.3：递归的问题(理解)

在我们实际编写递归的代码时会遇到很多的问题，那具体都有哪些问题呢？我们来一一分析。

问题 1：堆栈溢出

递归最常见的问题就是堆栈溢出，从而造成应用崩溃，那为什么递归代码会很容易造成堆栈溢出呢？

接下来我们看一段代码：

```
int main() {
   int a = 1;
   int b = 0;
   int c = 0;
   b = add(3, 5);
   c = a +b;
   return 0;
}
/*
* 求和
*/
int add(int x, int y) {
   int sum = 0;
   sum = x + y;
   return sum;
}
```

我们知道，操作系统给每个线程分配了一块独立的内存空间，这块内存被组织成“栈”这种结构,称为函数调用栈。 用来存储函数调用时的临时变量。每进入一个函数，就会将临时变量作为一个栈帧入栈，当被调用函数执行完成，返回之后，将这个函数对应的栈帧出栈。从代码中我们可以看出，main() 函数调用了 add() 函数，获取计算结果，并且与临时变量 a 相加。为了让大家清晰地看到这个过程对应的函数栈里出栈、入栈的操作，我画了一张图。图中显示的是，在执行到 add() 函数时，函数调用栈的情况。

通过图我们发现递归代码在执行的过程中，每一次递归的调用都会向函数调用栈中压入临时变量，直到满足递归终止条件在回归的过程中才会依次的将临时变量压出栈，如果递归调用的层次很深，一直在压入栈，我们知道系统栈或者虚拟机栈的空间一般都不大，所以如果一直入栈就会出现堆栈溢出的风险。

那我们在编码的过程中如何避免堆栈溢出呢？

我们可以在代码中限制递归调用的最大深度，比如递归调用超过 10000 次后就不在继续递归调用了，直接返回或者抛出异常。但是其实这样去做并不能完全的解决问题，因为当前线程能允许的最大递归深度其实跟当前剩余的栈空间大小有关系，事先是不知道有多大的，如果想知道就得实时的去计算剩余的栈空间大小，这样也会影响我们的性能，所以说如果递归的深度不是特别大的话是可以使用这种方式来防止堆栈溢出的，否则的话这种办法也不合适。

问题 2：重复计算

除了刚刚所讲到的堆栈溢出的问题外，递归还会出现重复计算的问题，比如一个非常典型的问题：斐波那契数列（Fibonacci sequence）

斐波那契数列指的是这样一个数列：1、1、2、3、5、8、13、21、34.....现在需要求出第 n 个数是多少？

通过分析查看我们可以知道，该数列从第三项开始，每一项的数都是前两项数的和，也就是说我们可以找到递推公式：当 n>=3 时 f(n)= f(n-1)+f(n-2);其中 f(2)=1,f(1)=1

如果我们使用递归来实现它则应该是：

```
public static int fibonacci(int n) {
        if (n == 1) {
            return 1;
        } else if (n == 2) {
            return 1;
        } else {
            return (fibonacci(n - 1) + fibonacci(n - 2));
```

```
        }
    }
```

那这里面如何涉及到了重复计算问题呢？假设我们现在要求解 f(5)，现将整个递归的过程分解如下：

从图中我们可以发现：f(3)被重复计算了多次，这就是重复计算的问题，那如何避免重复计算呢？

为了避免重复计算，我们可以通过一个数据结构（比如散列表）来保存已经求解过的 f(k)。当递归调用到 f(k) 时，先看下是否已经求解过了。如果是，则直接从散列表中取值返回，不需要重复计算，这样就能避免重复计算的问题了。

4.4：递归和循环

我们先来对比一下递归和循环的一个特点：递归是静中有动，有去有回；循环是动静如一，有去无回。从编程的角度来看递归有利有弊，利是递归代码的表达力很强，写起来非常简洁；而弊就是空间复杂度高、有堆栈溢出的风险、存在重复计算、过多的函数调用会耗时较多等问题，那**是否可以将递归代码改写为非递归的代码呢**？答案是肯定的。

我们可以**使用迭代循环的方式将递归代码改写为非递归代码**，如何改造呢？我们还是借用本节开篇的一个递推公式：f(n)=f(n-1)+1,其中 f(1)=1，

如果使用递归来写：

```
int f(int n) {
    if (n == 1){
        return 1;
    }
      return f(n-1) + 1;
}
```

如果改写成非递归的形式：

```
int f(int n) {
  int ret = 1;
  for (int i = 2; i <= n; ++i) {
    ret = ret + 1;
  }
  return ret;
}
```

递归其实和循环是非常像的，**循环都可以改写成递归，递归未必能改写成循环**。那么，有了循环，为什么还要用递归呢？？在某些情况下(费波纳切数列，汉诺塔)，使用递归会比循环简单很多很多。

4.5：递归的应用

前面所讲的斐波那契数列的求解就是递归的一种应用场景，接下来我们在来讲解几个使用递归的应用场景：

1：阶乘问题

阶乘问题的数学表达式为：**n! = n * (n-1) * (n-2) * ...* 1 (n>0)**，比如我们要求解 5 的阶乘

其实该问题的递推公式我们已经知晓，剩余就是将递推公式转换为 java 代码的实现，接下来我们实现如下：

```
/**
 * 阶乘问题
 * 阶乘的递推公式：n! = n*(n-1)*(n-2)*.....*1, 其中 n>0
 */
public class Factorial {

    /**
     * 求一个数 n 的阶乘
     * @param n
     * @return
     */
    public int factorial(int n){
        if(n==0){
            return 1;
        }else {
            return n * factorial(n-1);
        }
    }
    @Test
   public void test1(){
        //输出 5 的阶乘
        System.out.println(factorial(5));
   }
}
```

测试输出：发现 5 的阶乘结果为：120

那如果我们求 100 的阶乘呢？我们将 5 换成 100，查看输出结果为：0

这个时候我们的第一想法就是不可能，为什么 5 的阶乘能算出来，而 100 的阶乘反而是 0 呢？其实问题的根本原因在于我们使用 java 中的 int 类型进行乘法，100

的阶乘其结果最终超出了 int 类型的存储范围，所以我们得到的结果为 0，那如何来解决这个问题呢？

当然了要解决这种大数据运算我们有很多种解决方案，我们现在介绍一种最简单的方法，java 中提供了一些数学类可以帮助我们实现大数据运算，特别是一些商业计算，他们分别是：BigInteger 或者 BigDecimal

BigInteger 实现了任意精度的整数运算，BigDecimal 实现了任意精度的浮点数运算，接下来我们使用 BigInteger 来对我们的阶乘算法进行改造。

```
/**
 * 阶乘问题
 * 阶乘的递推公式：n! = n*(n-1)*(n-2)*.....*1, 其中 n>0
 */
public class Factorial {

    /**
     * 求一个数 n 的阶乘
     * @param n
     * @return
     */
    /*public int factorial(int n){
        if(n==0){
            return 1;
        }else {
            return n * factorial(n-1);
        }
    }*/

    public BigInteger factorial(int n){
        if(n==0){
            return BigInteger.ONE;
        }else {
            return BigInteger.valueOf(n).multiply(factorial(n-1));
        }
    }
    @Test
   public void test1(){
        //输出 100 的阶乘
        System.out.println(factorial(100));
    }
}
```

然后测试输出结果！

在这里给大家留下一个**课后思考题**：大数据运算除了使用 jdk 中提供的数学类来完成之外，还可以使用别的方式来完成吗？

2：目录拷贝

现在有这样一个应用场景：通过 java 程序拷贝一个目录下的所有文件及目录到另一个目录下。

通过分析这个场景我们发现，拷贝一个目录可以拆分成拷贝该目录下的所有文件数据及其子目录下的所有文件数据，拷贝子目录又可以拆分成拷贝子目录下的所有文件数据及其子子目录下的所有文件数据，直到某一目录下再也没有子目录为止，每层目录的拷贝操作都是一样的。因此完全符合了递归的使用条件，那接下来我们就来代码实现，在 java 中我们可以使用 BufferedInputStream、BufferedOutputStream 高效的拷贝文件夹。

代码实现如下：

```
/**
 * 高效拷贝目录
 */
public class CopyDir {

    /**
     * 实现目录拷贝
     * @param source
     * @param targer
     */
    public void copyDir(File source,File targer){
        if(source.isFile()|| !source.exists()){
            return;
        }
        //在目标目录下创建原目录
        File newDir = new File(targer,source.getName());
        newDir.mkdir();

        //获取原目录下的所有文件和目录
        File[] listFiles = source.listFiles();
        //遍历判断是文件还是目录，是文件则直接进行拷贝工作,是目录则递归调用自己实现拷贝
        for (File listFile : listFiles) {

            if(listFile.isFile()){
             //如果是文件则直接拷贝
                try {
                    BufferedInputStream bis = new BufferedInputStream(new FileInputStream(listFile));
                    BufferedOutputStream bos = new BufferedOutputStream(new FileOutputStream(new File(newDir,listFile.getName())));
                    int len = 0;
                    byte[] bytes = new byte[1024*8];
                    while ((len= bis.read(bytes))!=-1){
```

```
                    bos.write(bytes,0,len);
                }
                bos.close();
                bis.close();
            } catch (FileNotFoundException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }else {
            //如果是目录就递归拷贝
            copyDir(listFile,newDir);
        }

    }
}

@Test
public void testCopyDir(){
    //创建原目录
    File source = new File("D:\\io\\source");
    //创建目标目录
    File target = new File("D:\\io\\target");
    //实现拷贝
    copyDir(source,target);
}
}
```

测试时请先在原目录下准备一些文件以及目录等资源以待拷贝！

在这里同样给大家留下一个**课后思考题**：File 类下的 boolean delete()方法:删除此抽象路径名表示的文件或目录。如果此路径名表示一个目录，则该目录必须为空才能删除，那如何通过该方法删除一个包含多级子目录的目录呢？

4.6：小结

本节内容我们主要讲解了递归算法的概念，以及什么样的问题可以使用递归来解决，也分析了递归可能会出现的问题及对应的解决方案，最后使用递归解决了一个阶乘问题和一个目录拷贝的问题，在实际的企业开发中，在某些情况我们可以使用递归的形式去替代循环的写法，在某些适合递归的业务场景下我们能够使用递归的思路去解决该场景下的问题。

5：排序算法(重要)

在这一章节中我们要来学习排序算法，这对绝大部分从事软件行业的人来说都不陌生，可能你接触的第一个跟算法相关的东西就是排序，在很多的编程语言中都提供了排序的 API 可直接使用，在日常的软件研发过程中我们也经常使用到排序，当

然了排序的算法有很多，甚至有些排序算法我们都没听过，在我们本章节的课程中只讲其中最经典最常用的排序算法：冒泡排序，选择排序，插入排序，希尔排序，归并排序，快速排序，计数排序，桶排序，基数排序；在这些排序算法中如果按照时间复杂度来分类大致可以分为三类：：冒泡排序，选择排序，插入排序；：归并排序，快速排序，希尔排序；：计数排序，基数排序，桶排序。

我们要学习这么多的排序算法，除了学习其原理，实现其代码外还要评判出各种排序算法之间性能，效率。那我们应该从哪几个方面来分析一个排序算法是好是坏呢？所以在正式进入排序算法之前，我们先来说说排序算法的评判标准。

5.1：评判排序算法好坏的标准

对于众多的排序算法我们要将它们做一个对比需要从如下三个方面着手：

1：时间复杂度

时间复杂度其实就代表了一个算法执行的效率，我们在**分析排序算法的时间复杂度时要分别给出最好情况、最坏情况、平均情况下的时间复杂度**。为什么要区分这三种时间复杂度呢？第一，有些排序算法会区分，为了好对比，所以我们最好都做一下区分。第二，对于要排序的数据，有的接近有序，有的完全无序。有序度不同的数据，对于排序的执行时间肯定是有影响的，我们要知道排序算法在不同数据下的性能表现。

在之前的章节中学习复杂度的分析时我们说过，复杂度反映的是一个算法随着 n 的变化的一个增长趋势，在表示的时候往往会忽略表达式中的系数，低阶，常量，但是实际的软件开发中，我们排序的可能是 50 个、100 个、1000 个这样规模很小的数据，所以，**在对同一阶时间复杂度的排序算法性能对比的时候，我们就要把系数、常数、低阶也考虑进来**。

在本章节中讲的都是排序算法中的不同实现，其中有些排序算法是基于数据比较的排序算法，这些算法在执行过程中会涉及到比较元素大小，然后元素的交换或者移动，所以**在分析基于比较的排序算法时要将元素比较/交换/移动的次数也考虑进来**。

2：空间复杂度

空间复杂度在一个层面代表了算法对存储空间的消耗程度，我们可以简单的理解为算法的内存消耗，在这里我们还引入另外一个概念：**in-place** 和 **out-place**；其中 in-place 可以称为原地排序就是特指空间复杂度为 O(1)的排序算法，算法只占用常数内存，不占用额外内存，而 out-place 的算法需要占用额外内存。

3：算法稳定性

如果我们只用上面提到的时间复杂度和空间复杂度来度量一个排序算法其实是不够的，针对排序算法，还有一个指标就是：**稳定性**。

所谓排序算法的稳定性指的是：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。

举个例子，有一组数据：3 7 2 7 5 8 9，我们按照大小排序之后的数据为：2 3 5 7 7 8 9，在这组数据中有两个 7，如果经过某种排序算法后两个 7 的前后顺序没有发生改变则称该算法是稳定的排序算法，否则称为该算法是不稳定的排序算法。

在我们后续每学习一个排序算法我们都应该使用刚刚所讲的这个三个评判规则去分析该算法，好，接下来我们就依次的对每一个排序算法进行学习。

5.2：冒泡排序

原理

冒泡排序(Bubble Sort)是一种简单的排序算法，它通过依次比较两个相邻的的元素，看两个元素是否满足大小关系要求，如果不满足则交换两个元素。每一次冒泡会让至少一个元素移动到它应该在的位置上，这样 n 次冒泡就完成了 n 个数据的排序工作。这个算法的名字由来是因为越小的元素会经由交换慢慢“浮”到数列的顶端。

接下来对整个算法的过程进行描述：

- 比较相邻的元素。如果第一个比第二个大，就交换它们两个；
- 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对，这样在最后的元素应该会是最大的数；
- 针对所有的元素重复以上两个步骤，除了最后一个；
- 重复前三步，直到排序完成。

  为了便于大家理解，我们以一副动图的形式展现给大家(注：以下动图来自网络，下同)

链接：
https://mmbiz.qpic.cn/mmbiz*gif/QCu849YTaIOOdfiakqsTRHKk9icjqQZJYuffv5BticjiaK3BNNtdH6dRFglibdwgA9w2oR6QZTadJeZHdOsicqyjasPg/640?tp=webp&wxfrom=5&wx*lazy=1

实现

理解了冒泡排序的原理后代码实现如下：

```
/**
 * 冒泡排序算法
 * 冒泡排序是一种简单的排序算法。它重复地走访过要排序的数列，一次比较两个元
```

```
素，如果它们的顺序错误就把它们交换过来。
 * 走访数列的工作是重复地进行直到没有再需要交换，也就是说该数列已经排序完成。
 * 这个算法的名字由来是因为越小的元素会经由交换慢慢“浮”到数列的顶端
 *
 * 步骤：
 *  1：比较相邻的元素。如果第一个比第二个大，就交换它们两个；
    2：对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对，这样在最后的元素应该会是最大的数；
    3：针对所有的元素重复以上的步骤，除了最后一个；
    4：重复步骤 1~3，直到排序完成。
 */
public class BubbleSort {

    public void bubbleSort1(int[] array){
        int len = array.length;
        if(len<=1){
            return;
        }
        //开始冒泡
        for(int i=0;i<len;i++){

            for(int j=0;j< len - i -1;j++){
                //判断前后数据是否需要交换 如果前一个数据大于后一个数据则进行交换否则不交换
                if(array[j] > array[j+1]){
                    int temp = array[j];
                    array[j] = array[j+1];
                    array[j+1] = temp;
                }
            }
        }
    }

    /**
     * 测试冒泡排序
     */
    @Test
    public void testBubbleSort1(){
        //准备一个int 数组
        int[] array = new int[6];
        array[0] = 5;
        array[1] = 2;
        array[2] = 6;
        array[3] = 9;
        array[4] = 0;
        array[5] = 3;
        //进行排序
        System.out.println(Arrays.toString(array));
```

```
        bubbleSort1(array);
        //输出排序结果
        System.out.println(Arrays.toString(array));

    }

}
```

实际上，这里的冒泡排序算法还可以继续优化：因为当某次冒泡时发现已经没有数据需要进行交换时，说明所有元素都已经达到有序状态了，此时就不用再执行后续的冒泡操作了，接下来对之前的冒泡进行优化的代码实现如下：

```
    /**
     * 冒泡排序优化
     * @param array
     */
    public void bubbleSort2(int[] array){
        int len = array.length;
        if(len<=1){
            return;
        }
        //开始冒泡
        for(int i=0;i<len;i++){
            //是否需要提前结束冒泡的标识
            boolean flag = true;

            for(int j=0;j< len - i -1;j++){
                //判断前后数据是否需要交换 如果前一个数据大于后一个数据则进行交换否则不交换
                if(array[j] > array[j+1]){
                    int temp = array[j];
                    array[j] = array[j+1];
                    array[j+1] = temp;

                    flag = false;
                }
            }
            //在当前这次冒泡中如果所有元素都不需要进行交换则证明所有元素都已有序,则无需进行后续的冒泡操作了
            if(flag){
                break;
            }
        }
    }
    @Test
    public void testBubbleSort2(){
        //准备一个int 数组
```

```
        int[] array = new int[6];
        array[0] = 5;
        array[1] = 2;
        array[2] = 6;
        array[3] = 9;
        array[4] = 0;
        array[5] = 3;
        //进行排序
        System.out.println(Arrays.toString(array));
        bubbleSort2(array);
        //输出排序结果
        System.out.println(Arrays.toString(array));

    }
```

总结

对于冒泡排序我们要使用之前学习的三个标准来进行评判：

1：冒泡排序的时间复杂度是多少？

最好情况下，要排序的数据已经是有序的了，我们只需要进行一次冒泡操作，就可以结束了，所以
最好情况时间复杂度是 O(n)。而最坏的情况是，要排序的数据刚好是倒序排列的，我们需要进行 n
次冒泡操作，所以**最坏情况时间复杂度为 O(n2)**。

2：冒泡排序的空间复杂度是多少？

冒泡的过程只涉及相邻数据的交换操作，只需要常量级的临时空间，所以它的空间复杂度为 O(1)，是一种 in-place 排序算法。

3：冒泡排序是稳定的排序算法吗？

在冒泡排序中，只有交换才可以改变两个元素的前后顺序。为了保证冒泡排序算法的稳定性，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。

5.3：插入排序

原理

插入排序(Insertion Sort)的原理是：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。

算法描述如下：

- 从第一个元素开始，该元素可以认为已经被排序；
- 取出下一个元素，在已经排序的元素序列中从后向前扫描；
- 如果该元素（已排序）大于新元素，将该元素移到下一位置；
- 重复步骤 3，直到找到已排序的元素小于或者等于新元素的位置；
- 将新元素插入到该位置后；
- 重复步骤 2~5。

动图效果如下：

链接：
https://mmbiz.qpic.cn/mmbiz*gif/QCu849YTaIOOdfiakqsTRHKk9icjqQZJYusIFPUq7PlJn7maGNCmlhzTnLCkRcNjulAZk34Elic3oeVka2u4icXWDA/640?tp=webp&wxfrom=5&wx*lazy=1

实现

插入排序的算法实现如下：

```
/**
 * 插入排序算法
 * 插入排序（Insertion-Sort）的算法描述是一种简单直观的排序算法。
 * 我们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。
 *
 * 步骤：
 1：从第一个元素开始，该元素可以认为已经被排序；
 2：取出下一个元素，在已经排序的元素序列中从后向前扫描；
 3：如果该元素（已排序）大于新元素，将该元素移到下一位置；
 4：重复步骤 3，直到找到已排序的元素小于或者等于新元素的位置；
 5：将新元素插入到该位置后；
 6：重复步骤 2~5。
 */
```

```
public class InsertionSort {

    public void insertionSort1(int[] arr){
        int len = arr.length;
        if(len <=1) {
            return;
        }
        //开始排序
        for(int i= 1;i<len;i++){
            //取出未排序的下一个元素，及当前参与比较的元素
            int current = arr[i];
            //在已经排序的元素序列中从后向前扫描,定义前置索引
            int preIndex = i-1;
            // 从后向前依次和当前元素进行比较,
            while(preIndex >= 0 && arr[preIndex] > current ){
                //比较过程中如果元素大于当前的元素则将元素后移一位
                arr[preIndex+1] = arr[preIndex];
                preIndex --;
            }
            //比较过程中如果该元素小于等于当前元素,则将当前元素放在该元素后面
            arr[preIndex+1] = current;
        }
    }

    /**
     * 测试插入排序
     */
    @Test
    public void testInsertionSort1(){
        //准备一个int 数组
        int[] array = new int[6];
        array[0] = 5;
        array[1] = 2;
        array[2] = 6;
        array[3] = 9;
        array[4] = 0;
        array[5] = 3;
        //进行排序
        System.out.println(Arrays.toString(array));
        insertionSort1(array);
        //输出排序结果
        System.out.println(Arrays.toString(array));
    }
}
```

总结

1：插入排序的时间复杂度是多少？

如果要排序的数据已经是有序的，我们并不需要搬移任何数据。如果我们从尾到头在有序数据组里面查找插入位置，每次只需要比较一个数据就能确定插入的位置。所以这种情况下，**最好时间复杂度为 O(n)**。注意，这里是从尾到头遍历已经有序的数据。

如果数组是倒序的，每次插入都相当于在数组的第一个位置插入新的数据，所以需要移动大量的数
据，所以**最坏情况时间复杂度为 O(n2)**。还记得我们在数组中插入一个数据的平均时间复杂度是多少吗？没错，是 O(n2)。所以，对于插入排序来说，每次插入操作都相当于在数组中插入一个数据，循环执行 n 次插入操作，所以**平均时间复杂度为 O(n2)**。

2：插入排序的空间复杂度是多少？

从实现过程可以很明显地看出，插入排序算法的运行并不需要额外的存储空间，所以**空间复杂度是 O(1)**，也就是说，这是一个 in-place【原地排序】排序算法。

3：插入排序是稳定的排序算法吗？

在插入排序中，对于值相同的元素，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**。

5.4：选择排序

原理

选择排序(Selection Sort)的原理有点类似插入排序，也分已排序区间和未排序区间。但是选择排序每次会从排序区间中找到最小的元素，将其放到已排序区间的末尾。

算法描述如下：

- 初始状态：无序区间为 R[1..n]，有序区为空；
- 第 i 趟排序(i=1,2,3...n-1)开始时，当前有序区和无序区分别为 R[1..i-1]和 R(i..n）。该趟排序从当前无序区中-选出关键字最小的记录 R[k]，将它与无序区的第 1 个记录交换，使 R[1..i]和 R[i+1..n)分别变为记录个数增加 1 个的新有序区和记录个数减少 1 个的新无序区；
- n-1 趟结束，数组有序化了。

动图效果如下：

链接：
https://mmbiz.qpic.cn/mmbiz*gif/QCu849YTaIOOdfiakqsTRHKk9icjqQZJYuROpQscX9*

*fen1nqP1nia2lUADm29QpKHn7IqPn2Aiaic4DoPQ72GYKak6w/640?tp=webp&wxfrom=5&wx*lazy=1

实现

代码实现如下:

```
/**
 * 选择排序算法
 * 选择排序算法的实现思路有点类似插入排序，也分已排序区间和未排序区间。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
 */
public class SelectionSort {

    /**
     * 算法实现
     * @param arr
     */
    public void selectionSort(int[] arr){
        int len = arr.length;
        if(len<=1){
            return;
        }
        for(int i=0;i<len;i++){

            //接下来找到未排序区间的最小值的下标
            int minIndex = i;
            for(int j=i;j<len;j++){
                if(arr[j] < arr[minIndex]){
                    minIndex = j;
                }
            }
            //交换未排序区间最小元素和当前元素的位置
            int current = arr[i];
            arr[i] = arr[minIndex];
            arr[minIndex] = current;
        }
    }
    /**
     * 测试选择排序
     */
    @Test
    public void testInsertionSort1(){
```

```
        //准备一个int 数组
        int[] array = new int[6];
        array[0] = 5;
        array[1] = 2;
        array[2] = 6;
        array[3] = 9;
        array[4] = 0;
        array[5] = 3;
        //进行排序
        System.out.println(Arrays.toString(array));
        selectionSort(array);
        //输出排序结果
        System.out.println(Arrays.toString(array));
    }
}
```

总结

1：选择排序的时间复杂度是多少？

结合之前的分析方式分析可知选择排序的**最好情况时间复杂度为 O(n2)，最坏情况时间复杂度为:O(n2)，平均情况下的时间复杂度为：O(n2)**。

2：选择排序的空间复杂度是多少？

通过算法的实现我们可以发现，选择排序的**空间复杂度为 O(1)，是一个 in-place 排序算法**

3：选择排序是一个稳定的排序算法吗？

注意：**选择排序不是一个稳定的排序算法**，为什么呢？选择排序每次都要找剩余未排序元素中的最小值，并和未排序区间的第一个元素进行交换位置，这样破坏了稳定性，比如 5，8，5，2，9 这样一组数据，使用选择排序算法来排序的话，第一次找到最小元素 2，与第一个 5 交换位置，那第一个 5 和中间的 5 顺序就变了，所以就不稳定了。正是因此，从稳定性上来说选择排序相对于冒泡排序和插入排序就稍微逊色了。

5.5：归并排序

原理

归并排序(Merge Sort)的核心思想还是蛮简单的。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。

归并排序使用的是**分治思想**。分治，顾名思义，就是分而治之，将一个大问题分解成小的子问题来解决。小的子问题解决了，大问题也就解决了。从我刚才的描述，你有没有感觉到，分治思想跟我们前面讲的递归思想很像。是的，分治算法一

般都是用递归来实现的。分治是一种解决问题的处理思想，递归是一种编程技巧，这两者并不冲突。而对于递归就是要找到递推公式及终止条件，所以我们可以先写出归并排序的递推公式

`mergeSort(m->n) = merge(mergeSort(m->k),mergeSort(k+1->n));`当 m=n 时终止

我们来解释一下这个公式：我们要对 m->n 之间的数列进行排序，其实可以拆分成对 m->k 之间的数列进行排序，以及对 k+1->n 之间的数列排序，然后将连个拍好序的数列进行合并就称为了最终的数列，同样的道理，每一段数列的排序又可以继续往下拆分，形成递归。

算法描述：

- 把长度为 n 的输入序列分成两个长度为 n/2 的子序列；
- 对这两个子序列分别采用归并排序；
- 将两个排序好的子序列合并成一个最终的排序序列。

动图效果如下：

链接：

https://mmbiz.qpic.cn/mmbiz*gif/XaklVibwUKn4hqBOkhUQouzu2FDmmen4XHM8pExUK8sHf29rRqpwal4xeqhtJCxt7VDCA0ia5ZveB5rsdKngf6Iw/640?tp=webp&wxfrom=5&wx*lazy=1

https://mmbiz.qpic.cn/mmbiz*gif/QCu849YTaIOOdfiakqsTRHKk9icjqQZJYuEib77DsqHVhsUlM8iayfn5sV1ou3LKGjtDqzhhNPeibLDgyKw6S3Diam4g/640?tp=webp&wxfrom=5&wx*lazy=1

实现

代码实现如下：

```
/**
 * 归并排序算法
 *  归并排序的核心思想还是蛮简单的。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
```

```
 */
public class MergeSort {

    public int[] mergeSort(int[] arr){
        if(arr.length < 2){
            return arr;
        }
        //将数组从中间拆分成左右两部分
        int mid = arr.length/2;
        int[] left = Arrays.copyOfRange(arr,0,mid);
        int[] right = Arrays.copyOfRange(arr,mid,arr.length);
        return merge(mergeSort(left),mergeSort(right));
    }
    /**
     * 合并两个有序数组并返回新的数组
     * @param left
     * @param right
     */
    public int[] merge(int[] left,int[] right){
        //创建一个新数组,长度为两个有序数组的长度之和
        int[] newArray = new int[left.length+right.length];

        //定义两个指针,分别代表两个数组的下标
        int lindex=0;
        int rindex=0;
        for(int i=0;i<newArray.length;i++){

            if(lindex >= left.length){
                newArray[i] = right[rindex++];
            }else if(rindex >= right.length){
                newArray[i] = left[lindex++];
            }else if(left[lindex] < right[rindex] ){
                newArray[i] = left[lindex++];
            }else{
                newArray[i] = right[rindex++];
            }
        }
        return newArray;
    }


    @Test
    public void testInsertionSort1(){
        //准备一个int 数组
        int[] array = new int[6];
        array[0] = 5;
        array[1] = 2;
        array[2] = 6;
        array[3] = 9;
```

```
        array[4] = 0;
        array[5] = 3;
        //进行排序
        System.out.println(Arrays.toString(array));
        array = mergeSort(array);
        //输出排序结果
        System.out.println(Arrays.toString(array));
    }
}
```

总结

1：归并排序的时间复杂度是多少？

归并排序涉及递归，借此机会正好可以学习一下如何分析递归代码的时间复杂度，在学习递归时我们说递归适用的场景是一个问题 A 可以拆分成多个问题 B,C，问题 A 的解就等于问题 B 的解+问题 C 的解，同理，问题 B/C 可按照相同方式进行拆分，所以问题 A 的时间复杂度可以表示为：

```
O(A) = O(B)+O(C)+O(k)
```

其中 O(A)代表 A 的时间复杂度，O(B)代表求解问题 B 的时间复杂度，O(C)代表求解问题 C 的时间复杂度,O(k)代表将问题 B 的解和问题 C 的解合并称为最终的解的时间复杂度。

从刚刚的分析，我们可以得到一个重要的结论：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。

我们假设对 n 个元素进行归并排序需要的时间是 O(n)，那分解成两个子数组排序的时间都是 O(n/2)。我们知道，merge() 函数合并两个有序子数组的时间复杂度是 O(n)。所以，套用前面的公式，归并排序的时间复杂度的计算公式就是：

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

可能这个公式看起来仍然不直观，那我们继续往下分解几步，

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

所以最终：得到**归并排序的时间复杂度为：**

从我们的原理分析和伪代码可以看出，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，所以其时间复杂度是非常稳定的，不管是最好情况、最坏情况，还是平均情况，时间复杂度都是 O(nlogn)

2：归并排序的空间复杂度是多少？

归并排序的空间复杂度是多少呢？是：，因为归并排序的合并函数，在合并两个有序数组为一个有序数组时，需要借助额外的存储空间。这一点你应该很容易理解，但是如果我们继续按照分析递归时间复杂度的方法，通过递推公式来求解，那整个归并过程需要的空间复杂度就是 O(nlogn)。不过，类似分析时间复杂度那样来分析空间复杂度，这个思路对吗？

实际上，递归代码的空间复杂度并不能像时间复杂度那样累加。刚刚我们忘记了最重要的一点，那就是，尽管每次合并操作都需要申请额外的内存空间，但在合并完成之后，临时开辟的内存空间就被释放掉了。在任意时刻，CPU 只会有一个函数在执行，也就只会有一个临时的内存空间在使用。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)**，因此归并排序并不是一种 in-place 排序算法而是一种 out-place 排序算法。

3：归并排序是稳定的排序算法吗？

归并排序算法稳定还是不文档取决于合并函数 merge()。也就是两个有序子数组合并成一个有序数组的那部分代码，通过分析 merge 函数我们发现，**归并排序也是一个稳定的排序算法**。

5.6：快速排序

原理

快速排序(Quick Sort)算法，简称快排，利用的也是分治的思想，初步看起来有点像归并排序，但是其实思路完全不一样，快排的思路是：如果要对 m->n 之间的数列进行排序，我们选择 m->n 之间的任意一个元素数据作为分区点(Pivot)，然后我们遍历 m->n 之间的所有元素，将小于 pivot 的元素放到左边，大于 pivot 的元素放到右边，pivot 放到中间，这样整个数列就被分成三部分了，m->k-1 之间的元素是小于 pivot 的，中间是 pivot，k+1->n 之间的元素是大于 pivot 的。然后再根据分治递归的思想处理两边区间的的元素数列，直到区间缩小为 1，就说明整个数列都已有序了。

算法描述如下：

- 从数列中挑出一个元素，称为“基准”（**pivot**）；
- 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（partition）操作；

- 递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序。

动图效果如下：

连接：
https://mmbiz.qpic.cn/mmbiz*gif/XaklVibwUKn4hqBOkhUQouzu2FDmmen4Xn9ornBuYJeibehOnHAh3DibRoXHbfdpT6hhtkr4yFGViaWpb7rfvxfpiaw/640?tp=webp&wxfrom=5&wx*lazy=1

https://mmbiz.qpic.cn/mmbiz*gif/QCu849YTaIOOdfiakqsTRHKk9icjqQZJYuGKg8QiaSGDcDuicwJPIPniceyQYUz4uch8XvGsHOB2MUdhDsrvhwibibrKA/640?tp=webp&wxfrom=5&wx*lazy=1

如果我们用递推公式来将上面的过程写出来的话，就是这样：

递推公式：
```
quickSort(p…r) = quickSort(p…q-1) + quickSort(q+1, r)
```

终止条件：
```
p >= r
```

接下来将递推公式翻译成伪代码如下：

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

  q = partition(A, p, r) // 获取分区点
  quickSort(A, p, q-1)
  quickSort(A, q+1, r)
}
```

我们这里有一个 partition() 分区函数。partition() 分区函数实际上我们前面已经讲过了，就是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。如果我们不考虑空间消耗的话，partition() 分区函数可以写得非常简单。我们申请两个临时数组 X 和

Y，遍历 A[p...r]，将小于 pivot 的元素都拷贝到临时数组 X，将大于 pivot 的元素都拷贝到临时数组 Y，最后再将数组 X 和数组 Y 中数据顺序拷贝到 A[p...r]，如下图所示

但是，如果按照这种思路实现的话，partition() 函数就需要很多额外的内存空间，快排就不是一种 in-place 排序算法了。如果我们希望快排的空间复杂度得是 O(1)，那 partition()分区函数就不能占用太多额外的内存空间，我们就需要在 A[p...r] 的原地完成分区操作,那如何去完成呢？其实这里的实现思路非常的巧妙，下面是实现分区的伪代码，一起来分析一下：

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

实现

代码实现如下：

```
/**
 * https://visualgo.net
 * 快速排序算法
 *  1：从数列中挑出一个元素，称为 “基准”（pivot）；
    2：重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分区退出之后，该基准就处于数列的中间位置。这个称为分区（partition）操作；
    3：递归地（recursive）把小于基准值元素的子数列和大于基准值元素的子数列排序。
 *
 */
public class QuickSort {
    /**
     * 快排
     * 借助递归和分区的思想来实现
     * @param arr
     * @param begin
     * @param end
```

```
     */
    public void quickSort(int[] arr,int begin,int end){
        //判断递归截止条件
        if(arr.length <=1 || begin >=end){
            return;
        }
        //进行分区 得到分区下标
        int pivotIndex = partition(arr,begin,end);
        //对分区左侧进行快排
        quickSort(arr,begin,pivotIndex-1);
        //对分区右侧进行快排
        quickSort(arr,pivotIndex+1,end);
    }

    /**
     * 分区函数
     * @param arr
     * @param begin
     * @param end
     * @return
     */
    private int partition(int[] arr, int begin, int end) {
        //默认数组中待分区区间的最后一个是pivot 元素  当然也可以随机指定piv
ot 元素
        int  pivot = arr[end];
        //定义分区后pivot 元素的下标
        int pivotIndex = begin;
        for(int i= begin;i< end;i++){
            //判断如果该区间内如果有元素小于pivot 则将该元素从区间头开始一直
向后填充 有点类似选择排序
            if(arr[i] < pivot){

                if(i>pivotIndex){
                    swap(arr,i,pivotIndex);
                }
                pivotIndex++;
            }
        }
        swap(arr,pivotIndex,end);
        return pivotIndex;
    }

    /**
     * 交换数组内下标为i j 的两个元素
     * @param arr
     * @param i
     * @param j
     */
```

```
    private void swap(int[] arr,int i,int j){
        int temp = arr[j];
        arr[j] = arr[i];
        arr[i] = temp;
    }
    /**
     * 测试快排
     */
    @Test
    public void testQuickSort(){
        //准备一个int 数组
        int[] array = new int[7];
        array[0] = 5;
        array[1] = 2;
        array[2] = 6;
        array[3] = 9;
        array[4] = 0;
        array[5] = 3;
        array[6] = 4;
        //进行排序
        System.out.println(Arrays.toString(array));
        quickSort(array,0,array.length-1);
        //输出排序结果
        System.out.println(Arrays.toString(array));
    }
}
```

总结

1：快速排序的时间复杂度是多少？

快排的时间复杂度最好以及平均情况下的复杂度都是 O(nlogn)，只有在极端情况下会变成 O(n2)

2：快速排序的空间复杂度是多少？

通过快排的代码实现我们发现，快排不需要额外的存储空间，所有的操作都能在既定的空间内完成，因此快排的**空间复杂度为 O(1)**，也就是说快排是一种 in-place 的排序算法。

3：快速排序是稳定的排序算法吗？

因为分区的过程涉及交换操作，如果数组中有两个相同的元素，比如序列 6，8，7，6，3，5，9，4，在经过第一次分区操作之后，两个 6 的相对先后顺序就会改变。所以，**快速排序并不是一个稳定的排序算法**。

4：快排和归并的异同

首先快排和归并都用到了分治递归的思想，在快排中对应的叫分区操作，递推公式和递归代码也非常相似，但是归并排序的处理过程是由下到上的由局部到整体，先处理子问题，然后再合并。而快排正好相反，它的处理过程
是由上到下由整体到局部，先分区，然后再处理子问题。归并排序虽然是稳定的、时间复杂度为 O(nlogn) 的排序算法，但是它是一种 out-place 排序算法。主要原因是合并函数无法在原地(数组内)执行。快速排序通过设计巧妙的原地(数组内)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。

截至目前我们已经学习了：冒泡，插入，选择，归并，快排；并且分析了每种排序算法的原理、时间复杂度、空间复杂度、稳定性等。就复杂度而言：冒泡，插入，选择都是 O(n2)，归并和快排是：O(n logn)；接下来我们要来学习三种时间复杂度是 O(n) 的排序算法：桶排序、计数排序、基数排序。因为这些排序算法的时间复杂度是线性的，所以我们把这类排序算法叫作**线性排序（Linear sort）**。之所以能做到线性的时间复杂度，主要原因是这三个算法是非基于比较的排序算法，都不涉及元素之间的比较操作。

5.7：桶排序

原理

桶排序(Bucket Sort)顾名思义，会用到“桶”，桶我们可以将其想象成一个容器，核心思想是将要排序的数据分到几个有序的桶里，每个桶里的数据再单独进行排序。桶内排完序之后，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，换句话说：桶排序是将待排序集合中处于同一个值域的元素存入同一个桶中，也就是根据元素值特性将集合拆分为多个区域，则拆分后形成的多个桶，从值域上看是处于有序状态的。对每个桶中元素进行排序，则所有桶中元素构成的集合是已排序的。

桶排序过程中存在两个关键环节：

- 元素值域的划分，也就是元素到桶的映射规则。映射规则需要根据待排序集合的元素分布特性进行选择，若规则设计的过于模糊、宽泛，则可能导致待排序集合中所有元素全部映射到一个桶上，若映射规则设计的过于具体、严苛，则可能导致待排序集合中每一个元素值映射到一个桶上。
- 从待排序集合中元素映射到各个桶上的过程，并不存在元素的比较和交换操作，在对各个桶中元素进行排序时，可以自主选择合适的排序算法，每个桶内的排序算法的复杂度和稳定性，决定了最终的算法的复杂度和稳定性

那么桶排序的时间复杂度是多少呢？我们可以建议分析一下：

如果要排序的数据有 n 个，我们把它们均匀地划分到 m 个桶内，每个桶里就有 k=n/m 个元素。假设每个桶内部使用快速排序，时间复杂度为 O(k * logk)。m 个桶排序的时间复杂度就是 O(m * k * logk)，因为 k=n/m，所以整个桶排序的时间复杂度就是 O(n*log(n/m))。当桶的个数 m 接近数据个数 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)。

桶排序看起来是如此的优秀，那是不是可以替代我们之前讲到的排序算法呢？答案是否定的。首先，要排序的数据需要很容易就能划分成 m 个桶，并且，桶与桶之间有着天然的大小顺序。这样每个桶内的数据都排序完之后，桶与桶之间的数据不需要再进行排序。其次，数据在各个桶之间的分布是比较均匀的。如果数据经过桶的划分之后，有些桶里的数据非常多，有些非常少，很不平均，那桶内数据排序的时间复杂度就不是常量级了。在极端情况下，如果数据都被划分到一个桶里，那就退化为 O(nlogn) 的排序算法了。

桶排序比较适合用在非内存排序中。所谓的非内存排序就是说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。此外由桶排序的过程可知，当待排序集合中存在元素值相差较大时，对映射规则的选择是一个挑战，有时可能导致元素集中分布在某一个桶中或者绝大多数桶是空桶的现象，对算法的时间复杂度或空间复杂度有较大影响，所以桶排序适用于元素值分布较为集中的序列，或者说待排序的元素能够均匀分布在某一个范围[MIN, MAX]之间。

接下来笔者以一个亲身经历过的面试题为例来说明：比如说我们有 10GB 的订单数据，我们希望按订单金额（假设金额都是正整数）进行排序，但是我们的内存有限，只有几百 MB，没办法一次性把 10GB 的数据都加载到内存中。这个时候该怎么办呢？

现在我来讲一下，如何借助桶排序的处理思想来解决这个问题。我们可以先扫描一遍文件，看订单金额所处的数据范围。假设经过扫描之后我们得到，订单金额最小是 1 元，最大是 10 万元。我们将所有订单根据金额划分到 100 个桶里，第一个桶我们存储金额在 1 元到 1000 元之内的订单，第二桶存储金额在 1001 元到 2000 元之内的订单，以此类推。每一个桶对应一个文件，并且按照金额范围的大小顺序编号命名（00，01，02...99）。

理想的情况下，如果订单金额在 1 到 10 万之间均匀分布，那订单会被均匀划分到 100 个文件中，每个小文件中存储大约 100MB 的订单数据，我们就可以将这 100 个小文件依次放到内存中，用快排来排序。等所有文件都排好序之后，我们只需要按照文件编号，从小到大依次读取每个小文件中的订单数据，并将其写入到一个文件中，那这个文件中存储的就是按照金额从小到大排序的订单数据了。

不过，你可能也发现了，订单按照金额在 1 元到 10 万元之间并不一定是均匀分布的 ，所以 10GB 订单数据是无法均匀地被划分到 100 个文件中的。有可能某个金额区间的数据特别多，划分之后对应的文件就会很大，没法一次性读入内存。这又该怎么办呢？针对这些划分之后还是比较大的文件，我们可以继续划分，比如，订

单金额在 1 元到 1000 元之间的比较多，我们就将这个区间继续划分为 10 个小区间，1 元到 100 元，101 元到 200 元，201 元到 300 元...901 元到 1000 元。如果划分之后，101 元到 200 元之间的订单还是太多，无法一次性读入内存，那就继续再划分，直到所有的文件都能读入内存为止。

接下来为了能对该算法做具体的实现，我们对该算法进一步做具体的描述：

- 人为设置一个 BucketSize，作为每个桶所能放置多少个不同数值（例如当 BucketSize==5 时，该桶可以存放｛1,2,3,4,5｝这几种数字，但是容量不限，即可以存放 100 个 3）；
- 遍历输入数据，并且把数据一个一个放到对应的桶里去；
- 对每个不是空的桶进行排序，可以使用其它排序方法，也可以递归使用桶排序；
- 从不是空的桶里把排好序的数据拼接起来。

注意，如果递归使用桶排序为各个桶排序，则当桶数量为 1 时要手动减小 BucketSize 增加下一循环桶的数量，否则会陷入死循环，导致内存溢出。

图片演示：

实现

对该算法具体实现如下：

```
/**
 * 桶排序
 */
public class BucketSort {
    /**
     * 桶排序
     * @param array        待排序集合
     * @param bucketSize   桶中元素类型的个数即每个桶所能放置多少个不同数值（例如当 BucketSize==5 时，该桶可以存放｛1,2,3,4,5｝这几种数字，但是容量不限，即可以存放 100 个 3）
     * @return             排好序后的集合
     */
    public static List<Integer> bucketSort(List<Integer> array,int bucketSize){
        if(array == null || array.size() < 2 || bucketSize < 1){
            return array;
        }
```

```
        //找出集合中元素的最大值,最小值
        int max = array.get(0);
        int min = array.get(0);
        for(int i=0 ;i < array.size();i++){
            if(array.get(i) > max){
                max = array.get(i);
            }
            if(array.get(i) < min){
                min = array.get(i);
            }
        }
        //计算桶的个数    最大值-最小值代表了集合中元素取值范围区间
        int bucketCount = (max - min )/bucketSize +1;
        //按序创建桶,创建一个List,List 带下标是有序的,List 中的每一个元素是
一个桶,也用List 表示
        List<List<Integer>> bucketList = new ArrayList<>();
        for(int i=0;i< bucketCount;i++){
            bucketList.add(new ArrayList<Integer>());
        }
        //将待排序的集合依次添加到对应的桶中
        for(int j=0; j< array.size();j++){
            int bucketIndex = (array.get(j)-min)/bucketSize;
            bucketList.get(bucketIndex).add(array.get(j));
        }
        //对每一个桶中的数据进行排序(可以使用别的排序方式),然后再将桶中的数据
依次取出存放到一个最终的集合中
        //创建最终的集合
        List<Integer> resultList = new ArrayList<>();
        for(int j=0;j < bucketList.size(); j++){

            List<Integer> everyBucket = bucketList.get(j);
            //如果桶内有元素
            if(everyBucket.size()>0){

                //递归的使用桶排序为每一个桶进行排序
                //当某次桶排序待排序集合都分配到一个桶中时,缩小桶的范围以获
得更多的桶
                if(bucketCount ==1){
                    bucketSize--;
                }
                List<Integer> temp = bucketSort(everyBucket, bucketSiz
e);
                for(int i=0;i<temp.size();i++){
                    resultList.add(temp.get(i));
                }
            }
        }
        return resultList;
```

```
    }

    /**
     * 测试桶排序
     */
    @Test
    public void testBucketSort(){
        List<Integer> list = new ArrayList<>();
        list.add(5);
        list.add(2);
        list.add(2);
        list.add(6);
        list.add(9);
        list.add(0);
        list.add(3);
        list.add(4);
        System.out.println(list);
        List<Integer> bucketSort = bucketSort(list, 2);
        System.out.println(bucketSort);
    }
}
```

总结

1：桶排序的时间复杂度是多少？

桶排序的时间复杂度，取决与对各个桶之间数据进行排序的时间复杂度，如果我们将待排序元素映射到某一个桶的映射规则做的很好的话，很显然，桶划分的越小，各个桶之间的数据越少，排序所用的时间也会越少。但相应的空间消耗就会增大。我们一般对每个桶内的元素进行排序时采用快排也可以采用递归桶排序，通过我们刚开始的分析，当我们对每个桶采用快排时如果桶的个数接近数据规模 n 时，**复杂度为 O(n)**，如果在极端情况下复杂度退化为 O(n* log n)。

2：桶排序的空间复杂度是多少？

由于需要申请额外的空间来保存元素，并申请额外的空间来存储每个桶，所以**空间复杂度为 O(N+M)**，其中 M 代表桶的个数。所以桶排序虽然快，但是它是采用了用空间换时间的做法。

3：桶排序是稳定的排序算法吗？

桶排序是否稳定取决于对每一个桶内元素排序的算法的稳定性，如果我们对桶内元素使用快排时桶排序就是一个不稳定的排序算法。

5.8：计数排序

原理

计数排序(Counting Sort) 使用了一个额外的数组 C，其中第 i 个元素是待排序数组 A 中值等于 i 的元素的个数。然后根据数组 C 来将 A 中的元素排到正确的位置。其实计数排序其实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 m，我们就可以把数据划分成 m 个桶(其实是个数组)。每个桶内的数据值都是相同的，省掉了桶内排序的时间。每个桶内存储的也不是待排序的数据而是待排序数组 A 中值等于某个值的元素个数，接下来我们以一个例子来说明

我们都经历过高考，我们查分数的时候，系统会显示我们的成绩以及所在省的排名。如果你所在的省有 100 万考生，如何通过成绩快速排序得出名次呢？

我们都知道高考的满分是 750 分，最小是 0 分，这个数据的范围很小，所以我们可以分成 751 个桶，对应分
数从 0 分到 750 分。根据考生的成绩，我们将这 100 万考生划分到这 751 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，那具体如何来做呢？

在这里为了方便理解和说明，我们假设有 10 考生，大家的分数在 0-7 分之间，这 10 个考生的成绩我们存放在一个数组 A[10]中，分别为：1，4，5，1，0，3，4，2，6，3；因为成绩的分布是在 0-7 之间，我们使用一个大小为 8 的数组 C[8]代表 8 个桶，数组的下标对应的是考生的分数，数组 C 中存储的并不是考生信息，而是对应下标分数的考生个数，我们只需要编译一遍 A[10]这样就可以得到 C[8]的值如下：

从图中我们可以看出：分数为 3 的考生有 2 个，小于 3 分的考生有 4 个，所以，成绩为 3 分的考生在最终排序好的有序数组 R[10]中会保存在下标为：4，5 的位置上

那接下来就是计算出每个分数的考生在最终的有序数组中的存储位置，这个处理方案很是巧妙，下面是处理思路：对数组 C[8]顺序求和，就变成了下面这个样子，C[i]里存储的就是分数小于等于 i 的考生个数

下面就是计数排序中稍微复杂度一点的地方了，我们从后到前依次扫描待排序数组A。比如，当扫描到元素 3 时，我们可以从数组 C 中取出下标为 3 的值 6，也就是说，到目前为止，包括自己在内，分数小于等于 3 的考生有 6 个，也就是说 3 是最终有序数组 R 中的第 6 个元素（也就是数组 R 中下标为 5 的位置）。当 3 放入到数组 R 中后，小于等于 3 的元素就只剩下了 5 个了，所以相应的 C[3] 要减 1，变成 5。

以此类推，当我们扫描到第 2 个分数为 3 的考生的时候，就会把它放入数组 R 中的第 5 个元素的位置（也就是下标为 4 的位置）。当我们扫描完整个数组 A 后，数组 R 内的数据就是按照分数从小到大有序排列的了。

当然了这只是大致的思路，在代码的具体实现中我们还需要根据实际的情况来做出一些调整

动图连接：

https://mmbiz.qpic.cn/mmbiz*gif/QCu849YTaIOOdfiakqsTRHKk9icjqQZJYu0t2QuZRYMJqzEAUiaiagwpngltGHrJyegsZCwr7GpxQoRcSpTmypS3ag/640?tp=webp&wx*from=5&*wxlazy=1

实现

计数排序实现如下：

```
/**
 * 计数排序
 *  1:找出待排序的数组中最大和最小的元素；
    2:统计数组中每个值为 i 的元素出现的次数，存入数组 C 的第 i 项；
    3:对所有的计数累加（从 C 中的第一个元素开始，每一项和前一项相加）；
    4:反向填充目标数组：将每个元素 i 放在新数组的第 C(i)项，每放一个元素就将 C(i)减去 1。
 */
public class CountingSort {

    public void countingSort(int[] array){
        //求出待排序数组的最大值,最小值,找出取值区间
        int max = array[0];
        int min = array[0];
        for(int i=0;i<array.length;i++){
            if( array[i] > max){
```

```
                max = array[i];
            }
            if(array[i] < min){
                min = array[i];
            }
        }
        //定义一个额外的数组C
        int bucketSize = max - min +1;
        int[] bucket = new int[bucketSize];
        //统计对应元素的个数,数组的下标不是单纯的值
        for(int i=0;i< array.length;i++){
            int bucketIndex = array[i] - min;
            bucket[bucketIndex]+=1;
        }
        //对数组C 内元素进行累加
        for(int i=1;i<bucket.length;i++){
            bucket[i] = bucket[i] + bucket[i-1];
        }
        //创建临时数组R 存储最终有序的数据列表
        int[] temp = new int[array.length];
        //逆序扫描待排序数组 可保证元素的稳定性
        for(int i= array.length -1;i >=0;i--){
            int bucketIndex = array[i] - min;
            temp[bucket[bucketIndex]-1]  = array[i];
            bucket[bucketIndex] -=1;
        }
        /*for(int i= 0;i < array.length;i++){
            int bucketIndex = array[i] - min;
            temp[bucket[bucketIndex]-1]  = array[i];
            bucket[bucketIndex] -=1;
        }*/
        //将临时数据列表依次放入原始数组
        for(int i=0;i<temp.length;i++){
            array[i] = temp[i];
        }
    }

    /**
     * 测试快排
     */
    @Test
    public void testCountingSort(){
        //准备一个int 数组
        int[] array = new int[8];
        array[0] = 5;
        array[1] = 2;
        array[2] = 6;
        array[3] = 9;
        array[4] = 0;
```

```
        array[5] = 3;
        array[6] = 3;
        array[7] = 4;
        //进行排序
        System.out.println(Arrays.toString(array));
        countingSort(array);
        //输出排序结果
        System.out.println(Arrays.toString(array));
    }
}
```

总结

1：计数排序的时间复杂度是多少？

通过代码的实现过程我们发现计数排序不涉及元素的比较，不涉及桶内元素(数组C)的排序，只有对待排序数组和用于计数数组的遍历操作，因此**计数排序的时间复杂度是 O(n+k)，其中 k 是桶的个数即待排序的数据范围**，是一种线性排序算法。计数排序不是比较排序，排序的速度快于任何比较排序算法。由于用来计数的数组 C 的长度 k 取决于待排序数组中数据的范围（等于待排序数组的最大值与最小值的差加上 1），这使得计数排序对于数据范围很大的数组，需要大量时间和内存。

2：计数排序的空间复杂度是多少？

在计数排序的过程中需要创建额外的桶空间(数组 C)来计数，因此我们可以得知 **计数排序的口空间复杂度为：O(n+K)，其中 n 是数据规模大小，K 是计数排序中需要的桶的个数**，其实也就是用来计数的数组 C 的长度，之前我们提到过它取决于待排序数组中数据的范围。

3：计数排序是稳定的排序算法吗？

在计数排序中核心操作中我们是逆序的去扫描待排序数组，这样仍然可以使待排序数组中值相同但是位置靠后的元素在最终的已排序数组中保持着相同的位置关系，因此**计数排序是一个稳定的排序算法**。

4：计数排序的适用场景？

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，就不适合用计数排序了。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，要将其在不改变相对大小的情况下，转化为非负整数。

比如，还是拿分数这个例子。如果分数精确到小数后一位，我们就需要将所有的分数都先乘以 10，转化成整数，然后再放到桶内。再比如，如果要排序的数据中有负数，数据的范围是[-100, 100]，那我们就需要先对每个数据都加 100，转化成非负整数。

5.10：小结

接下来我们以一幅图的形式来总结一下各种排序算法。

排序算法	平均情况复杂度	最好情况复杂度	最坏情况复杂度	空间复杂度	排序方式	稳定性	备注
冒泡排序	O(n2)	O(n)	O(n2)	O(1)	in-place	稳定	
选择排序	O(n2)	O(n2)	O(n2)	O(1)	in-place	不稳定	
插入排序	O(n2)	O(n)	O(n2)	O(1)	in-place	稳定	
归并排序	O(n*log n)	O(n*log n)	O(n*log n)	O(n)	out-place	稳定	
快速排序	O(n*log n)	O(n*log n)	O(n2)	O(1)	in-place	不稳定	
桶排序	O(n)	O(n)	O(n*log n)	O(n+k)	out-place	不稳定	桶排序的复杂度和稳定性取决于用何种排序算法为每一个桶进行排序，在此以快排为例为每个桶进行排序，k 为桶的个数
计数排序	O(n+k)	O(n+k)	O(n+k)	O(n+k)	out-place	稳定	其中 n 为数据规模，k 为计数排序中需要的桶的个数，其实也就是用来计数的数组 C 的长

度，它取决于待排序数组中数据的范围

6：二分查找(重要)

6.1：二分查找的原理

在上一章节中我们学习了各种排序算法，接下来我们讲一种针对有序集合的查找算法：二分查找(Binary Search)算法，也叫折半查找算法。二分查找是一种非常简单易懂的快速查找算法，其思想在生活中随处可见，比如朋友聚会的时候爱玩的一个猜数游戏，我随机写一个 0-100 之间的数字，然后大家依次来猜，猜的过程中大家每猜一次我都会告诉大家猜大了还是猜小了，直到有人猜中为止，猜中的人会有一些惩罚措施。这个过程其实就是二分查找思想的一种体现。

这只是生活中的一个例子，我们现在回到实际的开发场景中，假设有 10 个订单，其金额分别是：6，12，15，19，24，26，29，35，46，67。请从中找出订单金额为 15 的订单，利用二分查找的思想我们每次都与区间中间的数据进行大小的比较以缩小查找的范围，下面这幅图代表了查找的过程，其中 low，high 代表了待查找区间的下标，mid 表示待查找区间中间元素的下标(如果范围区间是偶数个导致中间数有两个就选择较小的那个)

通过这个查找过程我们可以对二分查找的思想做一个汇总：**二分查找针对的是一个有序的数据集合，查找思想有点类似分治思想。每次都通过跟区间的中间元素对比，将待查找的区间缩小为之前的一半，直到找到要查找的元素，或者区间被缩小为 0**

6.2：二分查找的时间复杂度

理解了二分查找的思想后我们来分析二分查找的时间复杂度，首先我们要明确二分查找是一种非常高效的查找算法，通过分析其时间复杂度我们就可以发现，我们假设数据大小为 n，每次查找完后数据的大小缩减为原来的一半，直到最后数据大小被缩减为 1 此时停止，如果我们用数据来描述其变化的规律那就是：

n，n/2，n/4，n/8，n/16，n/32，……，1；

可以看出来，这是一个等比数列，当数据大小变为 1 时：

其中 k 的值就是总共缩小的次数。而每一次缩小操作只涉及两个数据的大小比较，所以，经过了 k 次区间缩小操作，通过计算 k 的值我们可以得出**二分查找的时间复杂度就是 O(logn)**

这是一种非常高效的时间复杂度，有时候甚至比 O(1)复杂度更高效，为什么这么说呢？因为对于 log n 来说即使 n 非常的大对应的 log n 的值也会很小，之前在学习 O(1)复杂度时我们讲过 O(1)代表的是一种常量级复杂度并不是说代码只需要执行一次，有时候可能要执行 100 次，1000 次这种常数级次数的复杂度都可以用 O(1)表示，所以，常量级时间复杂度的算法有时候可能还没有 O(logn) 的算法执行效率高。

6.3：二分查找的实现

二分查找的实现我们可以分为两大类情况：1，有序数列中不存在重复元素的简单实现；2：有序数列中存在重复元素的变形实现，当然了二分查找的实现方式可以采用循环或者递归实现。接下来我们就依次的来进行实现并分析实现过程中的一些问题

6.3.1：有序数列中不存在重复元素

就以前面提到的 10 个订单为例，我们用非递归及递归的方式均实现一下，给定一个有序的订单列表数据{6，12，15，19，24，26，29，35，46，67}，从中找出订单金额为 15 的订单在订单列表中的下标即可。

```
/**
 * 二分查找的简单实现
 */
public class SimpleBinarySearch {

    /**
     * 使用非递归的形式查找
     * @param array  待查找的有序数组
     * @param value  要查找的数值
     * @return
     */
    public int cicleBinarySearch(int[] array,int value){
        int low = 0;
        int high = array.length-1;

        while(low <= high){
            //int mid = (high+low)/2;
            //计算中间元素的下标
            int mid = low +((high-low) >> 1);
            if(array[mid] == value){
                return mid;
            }else if( array[mid] < value){
                low = mid+1;
            }else {
                high = mid-1;
            }
        }
```

```
        //如果至此还未找到则返回-1 代表未找到匹配的元素
        return -1;
    }

    /**
     * 使用递归的形式查找
     * @param array
     * @param value
     * @return
     */
     public int recursionBinarySearch(int[] array,int value,int low,int
high){
        if(low > high){
            return -1;
        }

        //int mid = (high+low)/2;
        //计算中间元素的下标
        int mid = low +((high-low) >> 1);
        if(array[mid] == value){
            return mid;
        }
        if( array[mid] < value){
            return recursionBinarySearch(array,value,mid+1,high);
        }else {
            return recursionBinarySearch(array,value,low,mid-1);
        }
    }

    @Test
    public void testBinarySearch(){
        //准备一个int 数组 6，12，15，19，24，26，29，35，46，67
        int[] array = new int[10];
        array[0] = 6;
        array[1] = 12;
        array[2] = 15;
        array[3] = 19;
        array[4] = 24;
        array[5] = 26;
        array[6] = 29;
        array[7] = 35;
        array[8] = 46;
        array[9] = 67;
        //进行排序
        System.out.println(Arrays.toString(array));
       // int i = cicleBinarySearch(array, 15);
        int i = recursionBinarySearch(array, 15,0,array.length-1);
        //输出排序结果
        System.out.println("找到了目标元素的下标:"+i+"---对应的值为:"+arra
```

```
y[i]);
    }
}
```

6.3.2：有序数列中存在重复元素

前面我们实现了简单二分查找，为什么说简单那是因为数据序列中不存在重复的元素，如果数据序列中存在重复元素在实际的应用场景中可能会出现如下几种情况：

1：从数据序列中查找第一个值等于给定值的元素

比如从如下给定的数据序列中找出第一个等于 29 的元素，按照我们之前的代码逻辑实现过程如下：

直接找到了下标为 7 的元素，但其实第一个等于 29 的元素下标为 6，因此这种简单的二分查找实现并不正确，解析来我们来写一下这个问题的代码实现思路和逻辑

```
/**
 * 数据序列中存在重复元素的二分查找的变形写法
 */
public class BinarySearch {
    /**
     * 查找数据序列中第一个等于给定值的元素的下标
     * @param array
     * @param value
     * @return
     */
    public int binarySearch1(int[] array,int value){
        //定义查找边界 low high
        int low = 0;
        int high = array.length-1;

        //循环的去判断
        while (low <= high){
            //计算 mid
            int mid = low + ((high-low)>>1);

            if(array[mid] == value){
                //之前是直接返回 mid,现在要找的是第一个等于值的元素。
                //由于数据序列是有序的,因此我们向前一位查看一下是否也等于该
值,如果相等则继续二分查找,否则直接返回 mid
                if(mid==0 || array[mid-1] != value){
                    return mid;
                }else {
                    high = mid-1;
```

```
                }
            }else if(array[mid] > value){
                high = mid -1;
            }else {
                low = mid +1;
            }
        }
        return -1;
    }

    /**
     * 测试查找数据序列中第一个等于给定值的元素的下标
     */
    @Test
    public void test1(){
        //准备一个int 数组 6，12，15，19，24，26，29，29，29，67
        int[] array = new int[10];
        array[0] = 6;
        array[1] = 12;
        array[2] = 15;
        array[3] = 19;
        array[4] = 24;
        array[5] = 26;
        array[6] = 29;
        array[7] = 29;
        array[8] = 29;
        array[9] = 67;
        //进行排序
        System.out.println(Arrays.toString(array));
        // int i = cicleBinarySearch(array, 15);
        int i = binarySearch1(array, 29);
        //输出排序结果
        System.out.println("找到了目标元素的下标:"+i+"---对应的值为:"+arra
y[i]);
    }
}
```

2：从数据序列中查找最后一个值等于给定值的元素

这个问题其实和刚刚第一个问题是相同的解决思路，现代码实现如下

```
/**
     * 从数据序列中查找最后一个值等于给定值的元素
     * @param array
     * @param value
     * @return
     */
    public int binarySearch2(int[] array,int value){
        //定义查找边界 low high
```

```
        int low = 0;
        int high = array.length-1;

        //循环的去判断
        while (low <= high){
            //计算mid
            int mid = low + ((high-low)>>1);

            if(array[mid] == value){
                //之前是直接返回mid,现在要找的是最后一个值等于给定值的元素
                //由于数据序列是有序的,因此我们向前后位查看一下是否也等于该值,如果相等则继续二分查找,否则直接返回mid
                if(mid==array.length-1 || array[mid+1] != value){
                    return mid;
                }else {
                    low = mid +1;
                }
            }else if(array[mid] > value){
                high = mid -1;
            }else {
                low = mid +1;
            }
        }
        return -1;
    }

    /**
     * 测试从数据序列中查找最后一个值等于给定值的元素
     */
    @Test
    public void test2(){
        //准备一个int 数组 6，12，15，19，24，26，29，29，29，67
        int[] array = new int[10];
        array[0] = 6;
        array[1] = 12;
        array[2] = 15;
        array[3] = 19;
        array[4] = 24;
        array[5] = 26;
        array[6] = 29;
        array[7] = 29;
        array[8] = 29;
        array[9] = 67;
        //进行排序
        System.out.println(Arrays.toString(array));
        // int i = cicleBinarySearch(array, 15);
        int i = binarySearch2(array, 29);
        //输出排序结果
```

```
        System.out.println("找到了目标元素的下标:"+i+"---对应的值为:"+arra
y[i]);
    }
```

3：从数据序列中查找第一个大于等于给定值的元素

有了前面两个问题解决思路的铺垫，这个问题的解决似乎变的更加简单，我们代码实现如下

```
/**
     * 从数据序列中查找第一个大于等于给定值的元素
     * @param array
     * @param value
     * @return
     */
    public int binarySearch3(int[] array,int value){
        //定义查找边界 low high
        int low = 0;
        int high = array.length-1;

        //循环的去判断
        while (low <= high){
            //计算 mid
            int mid = low + ((high-low)>>1);

            if(array[mid] >= value){
                if(mid==0 || array[mid-1] < value){
                    return mid;
                }else {
                    high = mid -1;
                }
            }else {
                low = mid +1;
            }
        }
        return -1;
    }

    /**
     * 测试从数据序列中查找第一个大于等于给定值的元素
     */
    @Test
    public void test3(){
        //准备一个 int 数组 6，12，15，19，24，26，29，29，29，67
        int[] array = new int[10];
        array[0] = 6;
        array[1] = 12;
        array[2] = 15;
        array[3] = 19;
```

```
        array[4] = 24;
        array[5] = 26;
        array[6] = 29;
        array[7] = 29;
        array[8] = 29;
        array[9] = 67;
        //进行排序
        System.out.println(Arrays.toString(array));
        // int i = cicleBinarySearch(array, 15);
        int i = binarySearch3(array, 28);
        //输出排序结果
        System.out.println("找到了目标元素的下标:"+i+"---对应的值为:"+arra
y[i]);
    }
```

4：从数据序列中查找出最后一个值小于等于给定值的元素

这个问题的实现思路是一样的，我们直接来实现即可

```
/**
     * 从数据序列中查找出最后一个值小于等于给定值的元素
     * @param array
     * @param value
     * @return
     */
    public int binarySearch4(int[] array,int value){
        //定义查找边界low high
        int low = 0;
        int high = array.length-1;

        //循环的去判断
        while (low <= high){
            //计算mid
            int mid = low + ((high-low)>>1);

            if(array[mid] <= value){
                if(mid== array.length-1 || array[mid + 1] > value){
                    return mid;
                }else{
                    low = mid + 1;
                }

            }else {
                high = mid -1;
            }
        }
        return -1;
    }
```

```
    /**
     * 测试从数据序列中查找出最后一个值小于等于给定值的元素
     */
    @Test
    public void test4(){
        //准备一个int 数组 6，12，15，19，24，26，29，29，29，67
        int[] array = new int[10];
        array[0] = 6;
        array[1] = 12;
        array[2] = 15;
        array[3] = 19;
        array[4] = 24;
        array[5] = 26;
        array[6] = 29;
        array[7] = 29;
        array[8] = 29;
        array[9] = 67;
        //进行排序
        System.out.println(Arrays.toString(array));
        // int i = cicleBinarySearch(array, 15);
        int i = binarySearch3(array, 30);
        //输出排序结果
        System.out.println("找到了目标元素的下标:"+i+"---对应的值为:"+arra
y[i]);
    }
```

6.4：二分查找的使用条件及场景

通过我们之前的分析可以知道，二分查找的时间复杂度是 O(log n)，其效率非常高，那是不是说所有情况下都可以使用二分查找呢？细心的你可能会发现在我们刚刚讲解这些场景的时候我们都有一些前提，下面我们讨论一下二分查找的应用前提

6.4.1：待查找的数据序列必须有序

二分查找对这一要求比较苛刻，**待查找的数据序列必须是有序的**，假如数据无序，那我们要先排序，然后二分查找，通过前面排序算法的学习我们知道，如果我们针对的是一组固定的静态数据，也就说该数据序列不会进行插入和删除操作，那我们完全可以先排序然后二分查找，这样子一次排序多次查找；但是如果数据序列本身不是固定的静态的，可能涉及数据序列的插入和删除操作，那我们每次查找前都需要进行排序然后才能查找，这样子成本非常的高。

6.4.2：数据的存储依赖数组

待查找的数据序列需要使用数组进存储，也就是说依赖顺序存储结构。那难道不能用其他的结构来存储待查找的数据序列吗？比如使用链表来存储，答案是不可以的，通过我们前面实现的二分查找的过程可知，二分查找，算法需要根据下标，low,high,mid 来访问数据序列中的元素，数组按照下标访问元素的复杂度是

O(1)，而链表访问元素的时间复杂度是 O(n)，因此如果使用链表来存储数据二分查找的时间复杂度就会变得很高。

6.4.3：数据量太小或太大都不适合用二分查找

数据量很小的情况下，没有必要使用二分查找，使用循环遍历就够了，因为只有在数据量比较大的情况下二分查找才能体出优势，不过在某些情况下即使数据量很小也建议大家使用二分查找，比如数据序列中的数据都是一些长度非常长的字符串，这些长度非常长的字符串比较起来也会非常的耗时，所以我们要尽可能的减少比较的次数，这样反倒有助于提高性能。

那为什么数据量太大的情况下也不建议使用二分查找呢？因为我们前面刚讲到二分查找底层需要依赖数组存储待查找的数据序列，而数组的特点是需要连续的内存空间，比如现在有 1G 的订单数据，如果用数组来存储就需要 1G 的连续内存，即便有 2G 的剩余内存空间，如果这 2G 的内存空间不连续那也无法申请到 1G 大小的数组空间，所以我们说数据量太大也不适合用二分查找。

6.5：小结

本章节我们学习了二分查找算法，在实际的开发中如果我们遇到有查找元素的需求时，如果满足我们所讲到的几个要求，我们就可以使用二分查找来提高查找的效率。

7：今日总结及作业安排

总结

今天我们主要学习了四个知识模块：

1：复杂度分析：

分析方式：看循环次数最多的代码，加法原则，乘法原则

常见的复杂度量级：O(1) , O(n)，O(n* log n)，O(log n)

2：递归：

满足递归的条件：递推公式(规律)，递归终止条件

警惕递归的问题：堆栈溢出，重复计算

递归的应用：阶乘，目录拷贝等

3：排序：

常见排序算法的原理及实现：

冒泡，插入，选择，归并，快排，桶排序，计数排序

4：二分查找：

作业安排

1：手动实现递归章节中的阶乘及目录拷贝问题

2：完成递归章节中留下的课后思考题

3：手动实现各个排序算法

4：手动实现二分查找的简单实现和变形情况。