# 附录 A 思考题答案

## 1.1 复杂度分析（上）：如何分析代码的执行效率和资源消耗

有人说，我们的项目都会进行性能测试，如果再做代码的时间复杂度分析、空间复杂度分析，那么是不是多此一举呢？而且，每段代码都分析一下时间复杂度、空间复杂度，是不是很浪费时间呢？读者怎么看待这个问题呢？

答案提示：尽管性能测试针对具体的运行环境和数据规模，给出的性能结果更加明确，但同时也导致了性能结果的局限性，只能代表某一特定情况下的性能表现。时间复杂度分析、空间复杂度分析能够让我们在不依赖测试机器的情况下，大致了解代码的运行效率。例如，最坏情况下的运行效率、最好情况下的运行效率和大约需要多少内存空间等。

当然，反过来说，有了时间复杂度分析、空间复杂度分析，性能测试也不能省略。复杂度分析只能给出粗略的估计结果，在特定运行环境和数据规模下，我们不能简单地认为低阶复杂度的算法就比高阶复杂度的算法运行时间少。

因此，我们既要进行复杂度分析，又要进行性能测试。两者不但不冲突，反而相辅相成。除此之外，分析时间复杂度分析、空间复杂度也并不浪费时间。对于简单的代码，往往一眼就能看出复杂度，而对于复杂的代码，因为其逻辑复杂，复杂度分析更加有必要，多花点时间去分析也是应该的。

## 1.2 复杂度分析（下）：详解最好、最坏、平均、均摊这 4 种时间复杂度

分析一下下面这段代码中 add() 函数的时间复杂度。

```
1 //类成员变量或全局变量：数组为array，长度为n，下标为i
2 int array[] = new int[10]; //初始大小为10
3 int n = 10;
4 int i = 0;
5 void add(int element) {
6   if (i >= n) { //数组空间不够了
7     //重新申请一个n的2倍大小的数组空间
8     int new_array[] = new int[n*2];
9     //把原来array数组中的数据依次复制到new_array
10    for (int j = 0; j < n; ++j) {
11      new_array[j] = array[j];
12    }
13    //new_array复制给array,array现在是n的2倍大小
14    array = new_array;
15    n = 2 * n;
16  }
17  array[i] = element;
18  ++i;
19 }
```

答案提示：这里所说的 add() 函数的时间复杂度指的是多次调用 add() 函数时，add()

函数执行效率的表现情况。尽管 n 初始化为 10，但 n 的大小一直在变化，因此，我们不能认为算法是常量级的时间复杂度。

当 i<n 时，即 i=0,1,2,…,n-1，代码不执行 for 循环，因此，这 n 次调用 add() 函数的时间复杂度都是 $O(1)$；当 i=n 时，for 循环进行数组的复制，因此，这次调用 add() 函数的时间复杂度是 $O(n)$。由此可知：

- 最好情况时间复杂度为 $O(1)$；
- 最坏情况时间复杂度为 $O(n)$；
- 平均或均摊情况时间复杂度为 $O(1)$。

其中，平均或均摊时间复杂度适合采用均摊时间复杂度分析法来分析。我们把时间复杂度为 $O(n)$ 的那次操作的耗时均摊到其他 $n$ 次时间复杂度为 $O(1)$ 的操作上，均摊下来的时间复杂度就是 $O(1)$。

## 2.1　数组（上）：为什么数组的下标一般从 0 开始编号

本节讲到了一维数组的内存寻址公式，类比一下，二维数组的内存寻址公式是怎样的呢？

答案提示：类比一维数组的寻址公式，对于 $m \times n$ 的二维数组，其寻址公式分下列两种情况。

第一种情况：如果数组是按行存储的（先存储第一行，再存储第二行，依此类推），那么二维数组的寻址公式为

$$a[i][j] \text{ 的 address}=\text{base\_address}+(i \times n+j) \times \text{type\_size}$$

第二种情况：如果数组是按列存储的（先存储第一列，再存储第二列，依此类推），那么二维数组的寻址公式为

$$a[i][j] \text{ 的 address}=\text{base\_address}+(j \times m+i) \times \text{type\_size}$$

除此之外，有很多编程语言对数组重新进行了定义和改造，它们的二维数组的寻址公式无法满足前面给出的标准形式，如 Java 中的多维数组的内存空间是不连续的，相关内容在 2.2 节中有详细讲解。

## 2.2　数组（下）：数据结构中的数组和编程语言中的数组的区别

对比 C/C++ 和 Java 中的数组的实现，分别有什么优缺点？

答案提示：C/C++ 把数组中的数据直接存储在一块连续的内存空间中，不需要像 Java 那样通过指针来多级索引，因此，相对更节省存储空间。不过，它对内存的要求也比较高，要求内存中可用的连续内存空间超过数组的需求，这样才能将数组存储。而在 Java 中，多维数组相当于使用多级索引组织内存，内存不需要完全连续，对内存的要求相对低很多。

## 2.3 链表（上）：如何基于链表实现 LRU 缓存淘汰算法

读者可能听说过如何判断一个字符串是否是回文字符串这个问题，本节的思考题是基于这个问题的改造版本。如果字符串存储在单链表中，而非数组中，那么如何判断字符串是否是回文字符串？相应的时间复杂度、空间复杂度是多少？

答案提示：回文字符串就是从左读和从右读都一样的字符串，换句话说，就是左右对称的字符串，如 abcba、abccba。

如果字符串存储在数组中，那么，我们在判断一个字符串是否是回文字符串时，只需要使用两个游标 $i$ 和 $j$，初始化 $i$ 指向字符串首，$j$ 指向字符串尾。判断 $i$ 和 $j$ 所指向的字符是否相等，如果相等，则 $i$++，$j$--，继续判断，直到 $i \geqslant j$（说明是回文串）或中途遇到不相等的字符（说明不是回文串）为止。

如果字符串存储在单链表中，那么我们借用上面的处理思路，把游标 $i$ 和 $j$ 换成指针 $p$ 和 $q$，$p$ 指向链表首节点，$q$ 指向链表尾节点。如果 $p$ 和 $q$ 所指节点包含的字符相等，我们就将 $p$ 更新为指向后继节点，$q$ 更新为指向前驱节点，剩下的处理思路与字符串存储在数组中的处理思路相同。

不过，$q$ 更新为指向前驱节点这个操作比较耗时。在单链表中寻找某个节点的前驱节点，需要遍历整个链表，时间复杂度是 $O(n)$。因此，按照这个实现思路，判断存储在单链表中的字符串是否是回文串的时间复杂度是 $O(n^2)$。

实际上，对于这个问题，我们还有时间复杂度为 $O(n)$ 的解决方法。大致思路如下：首先查找单链表的中间节点，然后逆转中间节点到尾节点这后半段的链表，比较前后两部分链表是否相同，判断是否是回文串，判断结束之后，将后半段链表重新逆转复原。

这个处理思路看似不难，实际上代码实现起来并不容易。查找单链表的中间节点需要使用快慢指针。逆转单链表的空间复杂度可以达到 $O(1)$，也就是在链表本身完成，不需要借助额外的非常量级的存储空间。

## 2.4 链表（下）：借助哪些技巧可以轻松地编写链表相关的复杂代码

本节提到了可以用“哨兵”来降低代码的实现难度，除文中举的例子，读者是否还能想到“哨兵”的其他一些应用场景呢？

答案提示：合并两个有序链表为一个有序链表这样一个操作，使用“哨兵”可以降低编码的实现难度。具体的 Java 代码如下所示。

```
public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
  if (l1 == null) return l2;
  if (l2 == null) return l1;
  ListNode p1 = l1;
```

```
  ListNode p2 = l2;
  ListNode head = new ListNode(); //不存储数据的“哨兵”
  ListNode r = head;
  while (p1 != null && p2 != null) {
    ListNode tmp;
    if (p1.val <= p2.val) {
     tmp = p1;
      p1 = p1.next;
   } else {
      tmp = p2;
      p2 = p2.next;
   }
    r.next = tmp; //有了“哨兵”之后，此处不需要特殊处理r为null的情况
    r = r.next;
  }
  if (p1 != null) r.next = p1;
  if (p2 != null) r.next = p2;
  return head.next;
}
```

## 2.5　栈：如何实现浏览器的前进和后退功能

1）本节讲到，编译器使用函数调用栈来保存临时变量，为什么要用“栈”来保存临时变量呢？用其他数据结构不行吗？

答案提示：其实，我们不一定非要用栈来保存临时变量，只不过函数调用符合后进先出的特性，用栈这种数据结构来实现是顺理成章的选择。

从调用函数进入被调用函数，对于数据，变化的是作用域。在进入被调用函数的时候，分配一段栈空间给这个函数的变量，在函数结束的时候，将栈顶复位，正好回到调用函数的作用域内。

2）在 Java 语言的 JVM 内存管理中，也有堆和栈的概念。栈内存用来存储局部变量和方法调用，堆内存用来存储 Java 对象。JVM 中的“栈”与本节提到的“栈”是不是一回事呢？如果不是，它为什么也称作“栈”呢？

答案提示：JVM 中的栈和数据结构中的栈都满足后进先出的特性，因此都称为栈。而 JVM 中的堆和数据结构中的堆是完全不相干的两个概念，JVM 中的堆可以理解为存储一“堆”对象的空间，而数据结构中的堆特指满足一定要求的一种完全二叉树。

## 2.6　队列：如何实现线程池等有限资源池的请求排队功能

如何用队列实现栈？如何用栈实现队列？

答案提示：我们可以用一个队列来模拟栈。同时，我们还需要记录队列中的数据个数。入栈时，我们将数据直接放入队列尾部。出栈时，假设当前队列中有 $k$ 个元素，我们先从队列头取 $k$-1 个元素，再依次放入队列尾部。此时，队列头就是要出栈的元素。

我们可以用两个栈来模拟队列。入队时，我们看哪个栈中的数据不为空，假设栈 *a* 不为空（如果都为空，则任选一个栈），将数据压入栈 *a*。出队时，我们将栈 *a* 中的数据依次弹出，压入栈 *b*，其中，栈 *a* 中的最后一个元素直接输出，不压入栈 *b*。最后，我们还要将栈 *b* 中的数据依次弹出，重新压入栈 *a*。

## 3.1 递归：如何用 3 行代码找到“最终推荐人”

在平时调试代码时，我们一般喜欢使用 IDE（集成开发环境）的单步跟踪功能，用以跟踪程序的运行，但对于规模比较大、递归层次很深的递归代码，几乎无法使用这种调试方式。对于递归代码，读者有什么好的调试方法呢？

答案提示：一般有以下两种方法。

- 打印日志。
- 结合条件断点进行调试。

## 3.2 尾递归：如何借助尾递归避免递归过深导致的堆栈溢出

根据如下代码，求解斐波那契数列的递归代码的空间复杂度。

```
int f(int n) {
  if (n == 0 || n == 1) {
    return n;
}
return f(n-1) + f(n-2);
}
```

答案提示：空间复杂度是指某一时刻代码所占的最大内存量，是一个峰值，而非代码运行过程中所消耗的总内存量。递归代码的空间复杂度等于栈最大深度乘以每层递归调用的空间消耗。斐波那契数列的递归代码每层递归调用只需要消耗很少的内存空间，是常量级的。而栈最大可以达到的深度大约是 *n*，因此，空间复杂度是 $O(n)$。

## 3.3 排序算法基础：从哪几个方面分析排序算法

我们平时提到的排序算法是将数据全部加载到内存中处理，如果数据量比较大，无法一次性把数据全部放到内存中，那么又该如何对数据进行排序呢？

答案提示：实际上，对于这类排序问题，我们把它称为外部排序。可以借助归并排序算法或桶排序算法来处理。

## 3.4　$O(n^2)$ 排序：为什么插入排序比冒泡排序更受欢迎

特定算法依赖特定数据结构。本节介绍的几种排序算法都是基于数组实现的。如果数据存储在链表中，这 3 种排序算法是否还能正常工作？如果能，对应的时间复杂度、空间复杂度是多少呢？

答案提示：对于冒泡排序，操作过程只涉及相邻元素的比较和交换，因此，使用链表存储数据，冒泡排序的时间复杂度、空间复杂度并不会升高。插入排序需要遍历查找插入位置，因此，时间复杂度、空间复杂度不变。选择排序每次从未排序区间遍历寻找最小元素，插入排序区间的末尾，因此，时间复杂度、空间复杂度也不变。

## 3.5　$O(n\log n)$ 排序：如何借助快速排序思想快速查找第 $K$ 大元素

假设有 10 个接口访问日志文件，每个日志文件的大小约 300MB，每个文件里的日志都是按照时间戳从小到大排序的。现在，我们希望将这 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序。如果处理上述排序任务的机器的内存只有 1GB，那么，在有限的机器资源的情况下，读者有什么好的解决思路能快速地将这 10 个日志文件合并？

答案提示：对于这个问题，我们借助归并排序中 merge() 函数的处理思路，不过 merge() 函数是两路合并，这里采用多路合并。

从每个文件中取一条数据，放入大小为 10 的数组中，比较找出其中的最小值，然后写入到最终的排序文件中。接着，从这个最小值对应的文件中，取出下一个数据，在数组中替换这个最小值，重复上面的过程，直到所有的数据都放入最终的排序文件为止。

不过，上面的处理思路没有充分利用内存。我们知道，磁盘的读写速度远慢于内存的读写速度。为了减少磁盘的读写次数，我们给每个小文件，以及最终的排序文件，都前置一个内存缓存（数组）。在读取数据时，一次性读取一批数据到内存，同理，写入数据时，先写数据到内存，等内存满了之后，再一次性地将内存中的数据写入到最终的排序文件中。

## 3.6　线性排序：如何根据年龄给 100 万个用户排序

本节讲的是针对特殊数据的排序算法。实际上，还有很多看似排序但又不需要使用排序算法就能处理的排序问题，例如下面这样一个问题。

对 [D,a,F,B,c,A,z] 这个字符数组进行排序，要求将其中所有的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，如 [a,c,z,D,F,B,A] 就是符合要求的一个排序结果。这个排序需求该如何实现呢？如果字符串中存储的不仅有大小写字母，还有数字，我们现在要将小写字母放到数组的最前面，大写字母放在数组的最后，数字放在数组的中间，不用排序算法，又该怎么实现呢？

答案提示：尽管第一个问题可以使用排序算法来解决，但是，对于这个问题的解决，排序算法有点“大材小用”了。对于这个问题，题目只要求小写字母在前、大写字母在后，并不要求每个数据都有序。我们可以使用两个游标 *i* 和 *j*，起始分别指向字符串数组的首部和尾部。更新 *i* 指向下一个大写字母，更新 *j* 指向下一个小写字母，交换 *i* 和 *j* 指向的字符。重复上面的过程，直到 *i* 和 *j* “相遇”为止。对应的代码如下所示。

```
void reorg(char[] str, int n) {
  int i = 0;
  int j = n-1;
  while (true) {
    while (i < j && str[i]>='a'&& str[i]<='z') {
      ++i;
    }
    while (i < j && str[j]>='A'&& str[j]<='Z') {
      --j;
    }
    if (i >= j) break;
    char tmp = str[i];
    str[i] = str[j];
    str[j] = tmp;
  }
}
```

对于升级后的问题，字符数组中包含小写字母、数字和大写字母，我们可以先把小写字母和数字看作一类字符，把大写字母单独看作另一类字符。我们先将第一类字符都放到第二类字符的前面，处理思路与上面给出的处理思路一样。在处理完之后，大写字母已经都放置到了数组的最后，我们只需要再对小写字母和数字组成的前半部分数组，按照同样的思路再进行处理，就可以把小写字母放到最前面，数字放置到中间。

## 3.7 排序优化：如何实现一个高性能的通用的排序函数

在本节中，作者分析了 C 语言中的 qsort() 的底层实现原理，读者能否像作者一样，分析一下自己熟悉的编程语言中的排序函数是用什么排序算法实现的？用了哪些优化手段？

答案提示：对于基本类型，Java 排序函数采用的是双枢轴快速排序（dual-pivot quicksort）算法，这个算法是在 Java 7 中引入的。在此之前，Java 采用的是普通的快速排序，双枢轴快速排序是对普通快速排序的优化，新算法的实现代码位于类 java.util.DualPivotQuicksort 中。

对于对象类型，Java 采用的算法是 TimSort 算法。TimSort 算法也是在 Java 7 中引入的。在此之前，Java 采用的是归并排序。TimSort 算法实际上是对归并排序的一系列优化。TimSort 算法的实现代码位于类 java.util.TimSort 中。

除此之外，如果数组长度比较小，那么 Java 排序函数会采用更加简单的插入排序。

## 3.8　二分查找：如何用最省内存的方式实现快速查找功能

1）如何编程实现“求一个数的平方根”？（要求精确到小数点后 6 位）

答案提示：实际上，这个问题可以分为两步来解决。第一步是确定平方根的整数部分，第二步是确定平方根的 6 位小数。对于第一步，为了加快速度，我们可以在 0 ～ $x$ 范围使用二分查找平方值小于或等于 $x$ 的最大数。对于第二步，每次确定一位小数，假设已经确定的数值是 a.bc，第 3 位小数只有可能是 0 ～ 9，我们就逐一考查 a.bc0 ～ a.bc9 这几个数，看哪个的平方值是小于或等于 $x$ 的最大值（假设是 a.bcd），我们再继续确定下一位小数，依此类推，直到 6 位小数都确定好。

对于第二步，因为每次只需要在 10 个数据中进行查找，所以顺序遍历就足够了。除此之外，查找最后一个小于或等于给定值的二分查找算法在 3.9 节中有详细讲解。

实际上，这个问题还有更加简单的解决思路，直接在 0 ～ $x$ 范围二分查找平方值等于 $x$ 的浮点数即可。不过，对于浮点数，二分查找结束的判断条件有所改变，需要引入精度，也就是题目给出的精确到小数点后 6 位。具体的代码如下所示。

```
public float calSQRT(float x) {
  float low = 0;
  float high = x;
  while (Math.abs(high - low) >= 0.000001) {
    float mid = (high + low) / 2;
    float mid2 = mid * mid;
    if (mid2 - x > 0.000001) {
      high = mid;
    } else if (x - mid2 > 0.000001) {
      low = mid;
    } else {
      return mid;
    }
  }
  return -1;
}
```

2）文中讲到，如果数据使用链表存储，二分查找的时间复杂就会变得很高，那么二分查找的时间复杂度究竟是多少呢？

答案提示：假设链表长度为 $n$，二分查找每次都要找中间点，因为时间复杂度的计算只需要计算得到一个量级，不需要具体值，所以，这里我们只是粗略地进行计算。

- 第一次查找中间点，大约需要遍历 $n/2$ 个节点。
- 第二次查找中间点，大约需要遍历 $n/4$ 个节点。
- 第三次查找中间点，大约需要遍历 $n/8$ 个节点。

依此类推，一直到遍历的节点个数为 1 为止。

粗略计算，总共遍历的节点个数大约为 $sum=n/2+n/4+n/8+\cdots+1$，这是个等比数列，根据等比数列求和公式得到：$sum=n-1$。对应的算法时间复杂度为 $O(n)$。也就是说，基于链表实现的二分查找的时间复杂度与顺序查找的时间复杂度相同。

## 3.9 二分查找的变体：如何快速定位 IP 地址对应的归属地

本节留给读者的思考题也是一个非常规的二分查找问题：如果有序数组是一个循环有序数组，如 [4,5,6,1,2,3]，那么，针对这种情况，如何实现一个求“值等于给定值”的二分查找算法呢？

答案提示：我们采用递归的处理思想来解决这个问题。下标 *low* 表示待查找区间的起始下标，下标 *high* 表示待查找区间的结束下标。下标 *mid* 表示区间的中间位置下标，也就是 *mid*=(*low*+*high*)/2。

- 如果下标 *low* 对应的元素小于下标 *mid* 对应的元素，就说明 [*low*,*high*] 区间的前半部分是有序数组，后半部分是循环有序数组。
- 如果下标 *low* 对应的元素大于下标 *mid* 对应的元素，就说明 [*low*,*high*] 区间的后半部分是有序数组，前半部分是循环有序数组。

我们用目标元素与有序数组进行比较，如果处于有序数组中，就在有序数组中继续递归查找，如果不在有序数组中，就在循环有序数组中继续递归查找。

对应的代码实现如下所示。

```
int bsearchInCycleSortedArray(int[] arr, int n, int value) {
  int low = 0;
  int high = n - 1;
  while (low <= high) {
    int mid = low + ((high - low) >> 1);
    if (arr[mid] == value) return mid;
    // 如果首元素大于mid元素，就说明后半部分有序，前半部分是循环有序数组
    if (arr[low] > arr[mid]) {
      if (arr[mid] < value && value <= arr[high]) low = mid + 1;
      else high = mid - 1;
    } else {
      if (arr[low] <= value && value < arr[mid]) high = mid - 1;
      else low = mid + 1;
    }
  }
  return -1;
}
```

## 4.1 哈希表（上）：Word 软件的单词拼写检查功能是如何实现的

1）假设有 10 万条 URL 访问日志，如何按照访问次数给 URL 排序？

答案提示：首先要统计每个 URL 对应的访问次数，然后按照访问次数对 URL 进行排序。假设 URL 的平均长度是 64B，那么存储 10 万条 URL 只需要大约 6MB 的存储空间，因此，我们可以将数据完全加载到内存中进行处理。

如何统计每个 URL 对应的访问次数呢？这里有两种处理思路。

第一种处理思路：首先，将数据加载到内存中；然后按照字符串的大小排序，排完序之后，相同的 URL 就放到一块，顺序遍历排好序的数组，就能得到每个 URL 对应的访问次数；最后，将 URL 和访问次数作为一个整体对象存储到另一个数组中。

第二种处理思路：首先，使用哈希表存储 URL 和对应的访问次数；然后，顺序遍历 10 万个 URL，用 URL 在哈希表中查找，如果找到，则将对应的访问次数加 1，如果没有找到，就将 URL 插入哈希表中，并将访问次数设置为 1，依此类推，当 10 万个 URL 都遍历完成之后，哈希表中就存储了每个 URL 及对应的访问次数；最后，将哈希表中的数据存储到数组中，方便接下来按照访问次数排序。

在得到存储了每个 URL 及访问次数的对象数组之后，我们使用排序算法按照访问次数从多到少给数组排序。至于排序算法，我们可以使用快速排序。实际上，如果访问次数的范围不大，那么我们可以采用桶排序来解决。

2）有两个字符串数组，每个数组大约有 10 万个字符串，如何快速找出两个数组中相同的字符串？

答案提示：这个问题有两种处理思路，可以使用哈希表，也可以不使用哈希表。

第一种处理思路：首先对两个数组 *a* 和 *b* 分别排序，然后申请两个游标 *i* 和 *j*，初始化分别指向两个数组中的起始下标，也就是 0。比较 *a*[*i*] 和 *b*[*j*]，对应以下 3 种情况：

- 如果 *a*[*i*]==*b*[*j*]，就说明两个字符串相同，输出到结果数组中，并且 *i*++，*j*++；
- 如果 *a*[*i*]<*b*[*j*]，那么 *i*++；
- 如果 *a*[*i*]>*b*[*j*]，那么 *j*++。

继续比较 *a*[*i*] 和 *b*[*j*]，直到某个数组为空为止。

第二种处理思路：将其中一个数组中的数据构建成哈希表，顺序遍历另一个数组，将每个字符串在哈希表中进行查找，如果找到，就说明此字符串在两个数组中都出现过，于是输出到结果数组中。

## 4.2　哈希表（中）：如何打造一个工业级的哈希表

本节讲到，Java 中的 HashMap 进一步做了优化，引入了红黑树。当链表长度大于或等于 8 时，就将链表转化成红黑树，而当红黑树中的节点个数小于或等于 6 时，又会将红黑树转化成链表。本节的思考题：为什么红黑树中的节点个数小于或等于 6 时才转化成链表，而不是小于 8 时就触发转化？

答案提示：我们举一个例子来解释一下。无论是红黑树转化成链表，还是链表转化成红黑树，我们都将触发转化的阈值设置为 8。当链表中有 7 个数据时，若我们执行插入操作，链表就会转化成为红黑树；若我们再执行删除操作，红黑树又会转化成链表。如果频繁、交替地插入、删除数据，就会导致频繁转化，而两种数据结构之间的转化是比较耗时的，会影响哈希表本身的性能。如果我们把红黑树转化成链表的阈值设置为 6，就能有效地避免这种情况的发生。

## 4.3 哈希表（下）：如何利用哈希表优化 LRU 缓存淘汰算法

如果将本节中的有序链表从双向链表改为单链表，LRU 缓存淘汰算法和 Java 中的 LinkedHashMap 是否还能正常工作？为什么？

答案提示：可以正常工作，但某些操作的时间复杂度会升高，因为在删除元素时，虽然通过哈希表可以在 $O(1)$ 时间复杂度内找到目标节点，但要删除该节点需要获取其前驱节点的指针，双向链表可以实现在 $O(1)$ 时间复杂度内找到前驱节点，但单链表查找某个节点的前驱节点的时间复杂度是 $O(n)$。

## 4.4 位图：如何实现网页“爬虫”中的网址链接去重功能

如何对一个存储了 1 亿个整数（范围为 1 ～ 10 亿）的文件的内容进行排序？

答案提示：如果直接将数据全部读取到内存中，并使用快速排序来排序，对应的内存消耗大约是 1 亿 ×4B=400MB。对应的时间复杂度是 $O(n\log n)$，其中 $n$ 是 1 亿。

如果使用位图，就需要申请包含 10 亿个二进制位大小的位图，对应的内存空间大约是 10 亿 /8=125MB，排序的时间复杂度是 $O(k)$，其中 $k$ 表示数字范围，也就是 10 亿。

不过，如果存在重复的数据，那么仅仅利用位图是不够的。对于重复的数据，我们还需要记录数据出现的次数。对于这部分功能的实现，我们可以使用哈希表来解决，其中哈希表中对象的 key 是数据，附属数据是此数据出现的次数，对应到 Java 编程语言中的 HashMap，那么 key 就是数据，value 就是出现的次数。

实际上，还有更加节省内存的处理思路。我们将这 1 亿个数据分批依次加载到内存。假设分为 10 批，每批数据只需要占用 40MB 大小的内存空间。我们分别对每批数据单独排序，然后写入一个小文件中。经过这一步处理之后，我们就得到了 10 个有序的小文件，再利用合并排序的 merge() 函数的处理思路，将多个有序小文件多路合并成一个大的有序文件。

## 4.5 哈希算法：如何防止数据库脱库后用户信息泄露

区块链是目前一个热门的领域，其底层的实现原理并不复杂。其中，哈希算法就是区块链的一个非常重要的理论基础。读者是否知道区块链使用的是哪种哈希算法？是为了解决什么问题而使用的？

答案提示：区块链是由一块块的区块组成的，每个区块分为两部分：区块头和区块体。区块头保存着自己的区块体和上一个区块头的哈希值。基于这种链式关系和哈希值的唯一性，只

要区块链中任意一个区块被修改过，后面的所有区块保存的哈希值就不对了。区块链使用的是 SHA256 哈希算法，计算哈希值非常耗时，如果要篡改一个区块，就必须重新计算该区块后面所有区块的哈希值，短时间内几乎不可能做到。

## 5.1　树和二叉树：什么样的二叉树适合用数组存储

本节讲解了二叉树的 3 种遍历方式：前序遍历、中序遍历和后序遍历。实际上，还有一种遍历方式，就是按层遍历，即首先遍历第一层节点，然后遍历第二层节点，依此类推。如何实现按层遍历呢？

答案提示：按层遍历需要借助具有先进先出特性的队列来实现。首先将根节点放入队列，然后循环从队列中取出节点并放入结果序列中，同时将其左子节点、右子节点依次放入队列。循环上面的处理过程，直到队列为空为止。对应的代码实现如下所示。

```
public void printByLevel(Node root) {
  if (root == null) return;
  Queue<Node> q = new ArrayDeque<>();
  q.add(root);
  while (!q.isEmpty()) {
    Node node = q.poll();
    System.out.println(node.data);
    if (node.left != null) q.add(node.left);
    if (node.right != null) q.add(node.right);
  }
}
```

## 5.2　二叉查找树：相比哈希表，二叉查找树有何优势

本节讲解了二叉树的高度的理论分析方法，只给出了粗略的估算值。本节的思考题：如何通过编程方式确切地求出一棵给定二叉树的高度？

答案提示：解决这个问题比较简单的方式是使用递归。递推公式如下所示。

节点 *A* 的高度 =max( 左子树的高度 , 右子树的高度 )+1

叶子节点的高度 =1 或 null 节点的高度 =0

对应的代码实现如下所示。

```
public int calHeight(Node root) {
  if (root == null) {
    return 0;
  }
  int leftTreeHeight = calHeight(root.left);
  int rightTreeHeight = calHeight(root.right);
  if (leftTreeHeight > rightTreeHeight) {
    return leftTreeHeight + 1;
  }
```

```
    return rightTreeHeight + 1;
}
```

在递归过程中，每个节点都被“考察”两次，因此，上述算法的时间复杂度是 $O(n)$，其中 $n$ 是节点的个数。递归的空间复杂度与函数调用栈的深度成正比，函数调用栈的深度与树的高度相等，因此，递归的空间复杂度是 $O(h)$，其中 $h$ 表示树的高度。

除递归的处理思路以外，还有一种非递归的处理思路。它是对按层遍历方式的一种改造。假设根节点的层数是 1，在将节点存入队列时，将其层数也附带着一起存入。当从队列中取出一个节点时，附带着将对应的层数也取出（假设为 $hx$），那么其左右子节点对应的层数就是 $hx+1$，将左右子节点及其对应的层数 $hx+1$ 再存入队列，其他过程不变。等队列为空时，最大的层数就是树的高度。

## 5.3 平衡二叉查找树：为什么红黑树如此受欢迎

在读者熟悉的编程语言中，哪种数据类型的实现用到了红黑树？

答案提示：Java 中的 TreeMap 使用红黑树来实现，除此之外，在 HashMap 中，当哈希表中某个“槽”对应的链表的长度大于或等于 8 时，链表会转化成红黑树。

## 5.4 递归树：如何借助树求递归算法的时间复杂度

假设 1 个细胞的生命周期是 3 小时，1 小时分裂一次。求 $n$ 小时后，容器内有多少个细胞？请读者用已经学过的递归时间复杂度的分析方法，分析一下这个递归问题的时间复杂度。

答案提示：假设在每个小时的起始时刻细胞先分裂后死亡，第 $n$ 个小时细胞个数 $f(n)$ 表示此小时分裂和死亡完成之后的净存细胞个数。那么，$f(n)$ 等于前一个小时细胞个数 $f(n-1)$ 分裂之后的个数（也就是 $2f(n-1)$）减去在第 $n$ 个小时死亡的细胞个数。

那么，第 $n$ 个小时会死亡多少细胞呢？

第 $n$ 个小时死亡的细胞肯定是第 $n-3$ 个小时分裂出来的新细胞，而第 $n-3$ 个小时分裂出来的细胞等于第 $n-4$ 个小时的细胞个数，也就是 $f(n-4)$。

综上所述，$f(n)=2f(n-1)-f(n-4)$。将此递推公式画成递归树，如图 A-1 所示。

每个节点分裂和合并（递和归）只需要常量级时间复杂度的操作耗时。我们只要统计树中有多少个节点，就能大致得到递归代码的时间复杂度。最左侧的路径是最长路径，长度约等于 $n$。最右侧的路径是最短路径，长度约为 $n/4$。因此，总节点个数介于

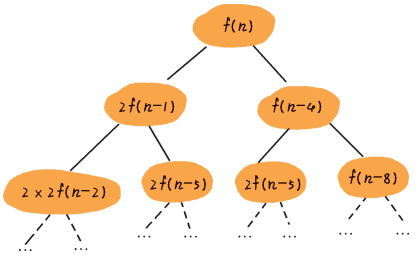


图 A-1 求解细胞分裂问题的递归代码对应的递归树

$2^{n/4}$ 和 $2^n$ 之间。也就是说，求解这个问题的递归代码的时间复杂度是指数级别的。

## 5.5　B+ 树：MySQL 数据库索引是如何实现的

在 B+ 树中，将叶子节点串起来的链表是单链表还是双向链表？为什么？

答案提示：是双向链表，而且是双向有序链表，主要是为了支持以下几个操作。

- 快速插入数据。在单链表中插入数据，需要先查找前驱节点，比较耗时。
- 支持降序查询。对于降序查询，如 `select * from user where uid > 12 and uid < 193 order by uid desc`。

尽管也可以先获取数据，再倒置输出，但是，如果涉及分页输出，显然，直接降序遍历是最快的。

## 6.1　堆：如何维护动态集合的最值

针对本节的开篇问题，我们分析了基于数组的解决方案。如果将数组换成链表，请读者试着分析一下插入数据、按值删除数据、查询最大值和删除最大值的时间复杂度。

答案提示：本节的开篇提到，假设有一个动态数据集合，其支持 4 个操作，包括插入数据、按值删除数据、查询最大值和删除最大值。如何实现这样一个动态集合，让每个操作的时间复杂度尽可能低？

如果单纯用链表来存储动态数据集合，插入数据时直接插入到链表的表头，查询最大值、删除最大值，以及按值删除数据都需要遍历整个链表，那么，尽管插入操作的时间复杂度是 $O(1)$，但其他 3 个操作的时间复杂度是 $O(n)$。

如果插入操作是比较频繁的操作，其他 3 个操作相比插入操作，没有那么频繁，那么这个解决方案已经是不错的了。如果查询最大值比较频繁，插入操作没有那么频繁，我们就需要调整一下方案。

我们还是使用链表来存储动态数据集合，但是，让链表中的数据一直维持从大到小有序。在插入数据时，为了保持链表在插入数据之后仍然有序，我们顺序遍历链表，查找数据应该插入的位置，然后将其插入，而非直接插入到链表表头或者表尾。这样，插入操作的时间复杂度是 $O(n)$，查询最大值、删除最大值操作的时间复杂度就变成了 $O(1)$，不过，按值删除操作的时间复杂度仍然是 $O(n)$。

## 6.2　堆排序：为什么说堆排序没有快速排序快

在介绍堆排序的建堆的时候，我们提到，对于完全二叉树，下标 $n/2+1 \sim n$ 的节点都是叶

子节点，这个结论是怎么推导出来的呢？

答案提示：这个问题采用反证法解决会非常简单。我们知道，堆是完全二叉树，适合用数组存储，节点 $i$ 的左子节点下标是 $2i$，右子节点下标是 $2i+1$。下标 $n$ 对应的节点肯定是叶子节点，其父节点下标是 $n/2$，肯定是非叶子节点。下标 $n/2+1$ 的节点肯定是叶子节点，如果是非叶子节点，那么其左子节点下标是 $n+2$，已经超出了下标范围。

重新梳理一下，下标为 $n/2$ 的节点肯定是非叶子节点，下标为 $n/2+1$ 的节点肯定是叶子节点，因此，下标从 $n/2+1$ 到 $n$ 的节点就是树中的叶子节点。

## 6.3 堆的应用：如何快速获取 Top 10 热门搜索关键词

假设有一个访问量非常大的新闻网站，我们希望将点击量排名 Top 10 的新闻滚动显示在网站首页上，并且每隔 1 小时更新一次。如何实现这个功能？

答案提示：用户每查看一个新闻网页，我们就将相应的 URL 记录在日志文件中。每小时会生成一个新的日志文件。每过一个小时，我们读取日志文件中的 URL，统计每个 URL 的访问次数。关于如何统计每个 URL 的访问次数，可以参见 4.1 节中的第一个思考题的解答。

如果 URL 访问日志非常多，我们就无法直接将其全部加载到内存中进行处理。针对这种情况，我们可以先利用哈希算法对日志文件进行分片，大的日志文件被分割成几个小的日志文件，并且，相同的 URL 访问记录被分配到同一个小日志文件中。针对每个小日志文件，我们再进行 URL 访问次数的统计。得到 URL 对应的访问次数之后，我们利用小顶堆，就可以找到访问次数排名前 10 的 URL 了。

## 7.1 跳表：Redis 中的有序集合类型是如何实现的

在本节的内容中，对于跳表的时间复杂度分析，我们分析了每两个节点抽取 1 个节点到上一级索引这种索引构建方式对应的查询操作的时间复杂度。如果索引构建方式变为每 3 个或每 5 个节点抽取 1 个节点到上一级索引，对应的查询数据的时间复杂度又是多少呢？

答案提示：无论每几个节点抽取 1 个节点到上层索引，在跳表中查询数据的时间复杂度都是 $O(\log n)$。因为时间复杂度并不是性能的准确体现，它只代表性能的量级。尽管每两个节点抽取 1 个节点到上层索引的方式，要比每 3 个节点抽取 1 个节点到上层索引的方式性能高，但性能的这点提升并不会体现在时间复杂度上。

每 3 个节点抽取 1 个节点到上层索引，对应跳表查询的时间复杂度分析方法，与文中讲的每两个节点抽取 1 个节点的分析方式是一样的。第一级索引大约有 $n/3$ 个节点，第二级索引大约有 $n/9$ 个节点，依此类推，我们可以得到整个跳表大约有 $\log_3 n$ 级索引。当查询数据时，每级索引最多遍历 4 个节点，因此，查询的时间复杂度是 $O(4\log_3 n)$，也就是 $O(\log n)$。

## 7.2　并查集：路径压缩和按秩合并这两个优化是否冲突

本节讲解了两种并查集的实现思路：基于链表的实现思路和基于树的实现思路。除链表和树，并查集是否可以使用其他数据结构来实现呢？例如数组，其对应的 union() 和 find() 操作的时间复杂度是多少？

答案提示：当然也可以使用数组来实现。与基于链表和树的实现思路类似，基于数组的实现思路也要有集合“代表”，并且，同样需要记录每个集合包含哪些元素，以及每个元素属于哪个集合。

我们使用动态数组来表示集合，并给每个集合分配一个唯一的 ID 编号（按 0、1、2、3 这样的顺序编号），并且把这个唯一的 ID 编号作为集合的“代表”。动态数组中的每个元素除存储数据之外，还存储对应所属集合的 ID 编号。而对于 ID 编号与数组之间的映射关系 , 我们将其存储在一个 Map（哈希表或者红黑树）中。

初始化每个集合中只包含一个元素。对于 union() 操作，我们通过元素找到对应的集合编号，然后通过哈希表得到集合。将一个集合中的数据全部加入另一个集合，并且更新这个集合中每个元素所属集合的编号。对于 find() 操作，我们对比两个元素对应的集合编号是否相同即可。

上述逻辑对应的代码实现如下所示，其中，union() 操作的时间复杂度是 $O(n)$，find() 操作的时间复杂度是 $O(1)$。

```
public class ArrayUnionFindSet {
  private Element elements[];
  private Map<Integer, List> hashmap = new HashMap<>();

  public ArrayUnionFindSet(int n) {
    elements = new Element[n];
    int setId = 0;
    for (int i = 0; i < n; ++i) {
      elements[i] = new Element(i, setId);
      List<Element> set = new ArrayList<>();
      set.add(elements[i]);
      hashmap.put(setId, set);
      setId++;
    }
  }

  public void union(int i, int j) {
    if (find(i, j)) return;
    List<Element> setA = hashmap.get(elements[i].setId);
    List<Element> setB = hashmap.get(elements[j].setId);
    for (Element e : setB) {
      e.setId = elements[i].setId;
    }
    setA.addAll(setB);
  }

  public boolean find(int i, int j) {
    return elements[i].setId == elements[j].setId;
  }
```

```
  public class Element {
    public int eid;
    public int setId;
    public Element(int eid, int setId) {
      this.eid = eid;
      this.setId = setId;
    }
  }
```

## 7.3 线段树：如何查找猎聘网中积分排在第 K 位的猎头

1）本节中的线段树针对的是整型数据。针对浮点型数据，又该如何构建线段树呢？

答案提示：对浮点数构建线段树，势必要考虑精度问题。假设要解决的问题中数据的最大精度是 0.0001，也就是小数点后 4 位。一旦确定了精度，就对应有两种解决思路。第一种解决思路是将所有的数据都乘以 10000，转化成整数处理。第二种解决思路是直接基于浮点数处理。区间 [*a*,*b*] 经过分解之后变为 [*a*,*mid*] 和 [*mid*+0.0001,*b*] 两个子区间。其他部分与整型数据类型的线段树无异。

2）在插入、删除数据，以及区间统计之前，我们需要先构建空线段树，相对来说比较耗时，是否可以不事先构建空线段树呢？

答案提示：完全可以不用先构建空的线段树。在插入数据的过程中，创建线段树中的节点也是可以的。但是，这样做会增加逻辑的复杂性，没有事先创建好空线段树的实现方式清晰。

## 7.4 树状数组：如何实现一个高性能、低延迟的实时排行榜

在数组中，如果两个元素满足 $a[i]>a[j]$ 且 $i<j$，就称这两个元素构成逆序对。本节的思考题：如何利用树状数组统计数组中的逆序对个数？

答案提示：我们申请一个数组 *b*，以及对应的树状数组 *c*。其中，数组 *b* 的下标是数组 *a* 中元素的值，数组 *b* 中的元素值表示下标对应的数组 *a* 中元素的个数。例如，数组 *a*={1, 5, 3, 2, 5}，那么对应的 *b*[1]=1，*b*[2]=1，*b*[3]=1，*b*[4]=0，*b*[5]=2。

初始化数组 *b* 中的每个元素都为 0，并且申请一个变量 *x* 用来记录有序对个数。顺序扫描数组 *a*，对于数组中的每个元素 *a*[*i*]，通过树状数组计算数组 *b* 的前缀和 *s*[*a*[*i*]]，也就是小于或等于 *a*[*i*] 的数据个数，将 *s*[*a*[*i*]] 值累加到 *x* 上，同时更新 *b*[*a*[*i*]]+=1，以及对应的树状数组 *c*。

当扫描完数组 *a* 的所有元素之后，变量 *x* 的值就等于数组 *a* 的有序对个数。逆序对个数就等于 $n(n-1)/2$ 减去有序对个数 *x*。

上述算法对应的代码实现如下所示，其中，数组 *b* 可以省略。

```
public class FenwickTree {
  private int n;
```

```
  private int c[];

  public FenwickTree(int n) {
    this.n = n;
    c = new int[n+1];
    for (int i = 1; i <= n; ++i) {
      c[i] = 0;
    }
  }

  private int lowbit(int i) {
    return i&(-i);
  }

  public int sum(int i) {
    int s = 0;
    while (i > 0) {
      s += c[i];
      i -= lowbit(i);
    }
    return s;
  }

  public void update(int i, int delta) {
    while (i <= n) {
      c[i] += delta;
      i += lowbit(i);
    }
  }
}

public class Solution {
  private int n = 6;
  private int[] a = {1, 4, 2, 5, 1, 7};
  private FenwickTree fenwickTree;

  public Solution() {
    int maxValue = Integer.MIN_VALUE;
    for (int i = 0;i < n; ++i) {
      if (maxValue < a[i]) maxValue = a[i];
    }
    fenwickTree = new FenwickTree(maxValue);
  }

  public int calInversions() {
    int x = 0;
    for (int i = 0; i < n; ++i) {
      x += fenwickTree.sum(a[i]);
      fenwickTree.update(a[i], 1);
    }
   return n*(n-1)/2 - x;
  }
}
```

## 8.1　BF 算法：编程语言中的查找、替换函数是怎样实现的

本节介绍的字符串匹配算法返回的结果是第一个匹配子串的首地址，如果要返回所有匹配

子串的首地址，该如何实现呢？如何实现替换函数 replace() ？

答案提示：返回所有的匹配只需要对文中的代码稍作修改，如下所示。

```
//返回所有匹配的起始下标位置
int matchedPos[] = new int[n];//申请大一点的空间
int bf(char mainStr[], int n, char subStr[], int m, int matchedPos[]) {
  int matchedNum = 0;
  for (int i = 0; i < n-m; ++i) {
    int j = 0;
    while (j < m) {
      if (mainStr[i+j] != subStr[j]) {
        break;
      }
      j++;
    }
    if (j == m) {
      matchedPos[matchedNum] = i;
      matchedNum++;
    }
  }
  return matchedNum;
}
```

实现 replace() 函数也不难，借助上面的匹配函数，实现代码如下所示。

```
char* replace(char mStr[], int n,
              char pStr[], int m, char[] rStr, int k) {
  int[] matchedPos = new int[n];
  int matchedNum = bf(mStr, n, pStr, m, matchedPos);
  char[] newStr = new char[n+(k-m)*matchedNum];
  int p = 0; //mStr上的游标
  int q = 0; //newStr上的游标
  for (int i = 0; i < matchedNum; ++i) {
    while (p < matchedPos[i]) {
      newStr[q++] = mStr[p++];
    }
    for (int j = 0; j < k; ++j) {
      newStr[q++] = rStr[j];
    }
    p+=m;
  }
  while (p<n) {
    newStr[q++] = mStr[p++];
  }
  return newStr;
}
```

## 8.2 RK 算法：如何借助哈希算法实现高效的字符串匹配

8.1 节和本节讲的是一维字符串的匹配方法，实际上，BF 算法和 RK 算法都可以类比到二维空间。假设有一个二维字符矩阵（如图 8-7 中的主串），借鉴 BF 算法和 RK 算法的处理思路，如何在其中查找另一个二维字符矩阵（如图 8-7 中的模式串）呢？

答案提示：假设二维主串是 $N\times M$，模式串是 $n\times m$。我们用模式串在主串中尝试匹配，总共有 $(N-n+1)\times(M-m+1)$ 种匹配方式。我们需要依次考察每种匹配方式是否真的完全匹配。

按照 BF 算法的处理思想，考察每个匹配方式的耗时是 $nm$，因此，整体的时间复杂度是 $O(NMnm)$。按照 RK 算法的处理思想，考察每个匹配方式的耗时是 $n$ 或者 $m$（右移或下移），因此，整体的时间复杂度是 $O(NMn+NMm)$。

## 8.3　BM 算法：如何实现文本编辑器中的查找和替换功能

如果我们单独来看时间复杂度，那么，当模式串的长度 $m$ 比较小的时候，BF、RK 和 BM 算法的性能表现相差不大。在实际的软件开发中，大部分模式串也不会很长，那么，BM 算法是不是就没有太大的实践意义了呢？是不是只有在模式串很长的情况下，BM 算法才能发挥绝对优势呢？

答案提示：时间复杂度只能粗略地代表性能的量级差距，具体的性能表现不能完全依靠它。在实际的软件开发中，对于某些核心、高频、耗时多的代码，几倍甚至百分之几的性能提升都是值得努力实现的。例如网络安全入侵检测这样的应用场景，字符串匹配是其中核心的逻辑，尽管每个模式串可能都不长，单一聚焦在一次字符串匹配上，可能用什么算法都相差无几，但微小性能的差距累加起来，整个应用的运行效率就相差很多。综上所述，并不是只有模式串在很长的情况下，BM 算法才能发挥优势。

## 8.4　KMP 算法：如何借助 BM 算法理解 KMP 算法

我们已经学习了 4 种字符串匹配算法。对于每种算法，我们都给出了理论上的性能分析，也就是时间复杂度分析。对于性能，除理论分析以外，有时我们还需要真实数据的验证。如何设计测试数据、测试方法，对比各种字符串匹配算法的执行效率呢？

答案提示：根据测试的精细程度，我们可以设计不同的测试数据。

如果只是粗略测试，那么我们只需要随机生成多组模式串和主串，用随机数据直接测试各个字符串匹配算法。对比每组测试数据的测试结果，可以看到 RK 算法的执行时间是 BF 算法的多少倍，BM 算法的执行时间是 BF 算法的多少倍。最后，对所有的测试结果取平均值，就能得到各个算法之间的性能关系。

如果要精细测试，那么会测试对于不同长度的模式串和主串、不同特点的模式串和主串，各个算法的性能表现，这就要根据不同的测试要求，生成不同的测试数据，测试的方式不变，仍然是对比各个算法的执行时间。

## 8.5　Trie 树：如何实现搜索引擎的搜索关键词提示功能

在网络传输中，数据包通过路由器来中转。路由器中的路由表记录了路由规则。一条路由

规则包含目标 IP 地址段及相应的路由信息（如数据包的转发地址）。数据包携带的目标 IP 地址有可能与多个路由规则的目标 IP 地址段的前缀匹配，在这种情况下，我们会选择最长前缀匹配的规则作为最终的路由规则。本节的思考题：如何存储路由表信息，才能做到快速地查找某个数据包对应的转发地址？

答案提示：我们将路由规则按照 IP 地址组织成 Trie 树。因为 IP 地址的格式为 *xxx.xxx.xxx.xxx*，因此，Trie 树最多有 4 层，每层最多有 256 个节点。为了节省内存消耗，我们将每层的节点组织成有序数组（因为路由表很少更新）。这样就能快速地查询数据包的目标 IP 地址最长前缀匹配的路由规则。

## 8.6 AC 自动机：如何用多模式串匹配实现敏感词过滤

到此为止，对于字符串匹配算法，我们全部介绍完毕。本节的思考题：各个字符串匹配算法的特点分别是什么？它们比较适合的应用场景有哪些？

答案提示：BF、RK、BM 和 KMP 是单模式串匹配算法，Trie 树和 AC 自动机是多模式串匹配算法。在单模式串匹配算法中，BF 算法的时间复杂度最高，但代码实现简单，对于小规模字符串匹配，使用 BF 算法就足够了，因此，大部分编程语言中提供的字符串匹配函数是利用 BF 算法实现的。当然，如果字符串匹配是核心、高频、耗时多的操作，那么我们就要优先选择 BM 或 KMP 这样更加高效的算法。

在多模式串匹配算法中，Trie 树常用在前缀匹配中，AC 自动机类似 KMP 算法，执行效率更高，因此，真正需要用到多模式串匹配的应用场景会优先选择使用 AC 自动机。

## 9.1 图的表示：如何存储微博、微信等社交网络中的好友关系

关于本节的开篇问题，我们只介绍了微博这种有向图的解决思路，像微信这种无向图，应该怎么存储呢？读者可以按照作者的思路，自己进行练习。

答案提示：因为数据结构是为算法服务的，所以，具体选择哪种存储方法与期望支持的操作有关。针对微信的用户关系，假设需要支持下面这样几个操作：

- 判断用户 A 与用户 B 是否是好友关系；
- 用户 A 和用户 B 建立好友关系；
- 用户 A 和用户 B 删除好友关系；
- 根据用户名称的首字母排序，分页获取用户的好友列表。

关于如何存储一个图，本节介绍了两种方法：邻接矩阵和邻接表。因为社交网络是一个稀疏图，使用邻接矩阵比较浪费存储空间，所以，我们采用邻接表来存储微信的好友关系。

在无向图的邻接表中，如果用户 A 和用户 B 是好友关系，就在 A 的链表中添加一个顶点 B，同时，在 B 的链表中添加一个顶点 A。基于这样的存储结构，我们就能很容易地实现上述

几个操作。

## 9.2　深度优先搜索和广度优先搜索：如何找出社交网络中的三度好友关系

1）我们用广度优先搜索解决了本节开篇提到的问题，请读者思考一下，本节的开篇问题是否可以用深度优先搜索来解决？

答案提示：基于深度优先搜索，当遍历到三度人脉之后，就停止继续递归，回溯到上一层顶点继续探索。当递归结束之后，我们就能找到所有的三度人脉了。需要注意的是，遍历之后的顶点还能再重复遍历，只需要通过新的路径到达这个顶点的路径长度更小，并且不超过 3。

2）对于数据结构和算法的学习，最难的不是掌握原理，而是能灵活地将各种场景和问题抽象成对应的数据结构和算法。本节提到，迷宫可以抽象成图，走迷宫可以抽象成搜索算法，那么，如何将迷宫抽象成一个图？（换个说法，如何在计算机中存储一个迷宫？）

答案提示：我们可以把每个岔路口看成一个顶点，把岔路口与岔路口之间的路径看成边，由此来构建一个无向图。

## 9.3　拓扑排序：如何确定代码源文件的编译依赖关系

在本节的讲解中，我们用 $a$ 到 $b$ 的有向边来表示 $a$ 先于 $b$ 执行，也就是 $b$ 依赖于 $a$。如果我们换一种依赖关系的表示方法，用 $b$ 到 $a$ 的有向边来表示 $a$ 先于 $b$ 执行，也就是 $b$ 依赖于 $a$，那么，本节讲的 Kahn 算法和深度优先搜索是否还能正确工作呢？如果不能，应该如何改造呢？

答案提示：不能正确工作。但只要我们稍微改造一下，就能正确工作。对于 Kahn 算法，我们需要记录每个节点的出度而非入度，如果出度是 0，则执行这个节点对应的任务。对于深度优先搜索，处理起来更加方便，我们直接使用邻接表即可，不需要先生成逆邻接表。

## 9.4　单源最短路径：地图软件如何“计算”最优出行路线

在计算最短时间的出行路线时，如何获得通过某条路的时间呢？（这个思考题很有意思，在作者之前面试时也曾被问到过，考验的是一个人是否思维活跃，读者也可以思考一下。）

答案提示：在将地图抽象成图时，我们把岔路口抽象成顶点，将岔路口之间的路抽象成有向边。在计算最短时间出行路线时，边的权重是通行时间。通行时间如何得到呢？这是一个比较开放的问题。某段路的通行时间与很多因素有关，基础的两个因素是路长和拥堵情况。路长是固定的，但拥堵情况是动态变化的，因此，通行时间也是动态变化的。比较简单的获取方法

是，通过跟踪其他用户通过此条路的时间，来统计得到在最近一段时间通过此条路的时间。当然，我们也可以根据拥堵情况（如用车辆个数表征拥堵程度）建立模型，通过模型来计算通行时间。

## 9.5 多源最短路径：如何利用 Floyd 算法解决传递闭包问题

在本节给出的 Floyd 算法的代码实现中，最终得到的 dist 数组只存储了顶点之间的最短路径长度，并没有给出最短路径包含了哪些边。那么，如何改造代码，在求解最短路径长度的同时，得到最短路径具体包含了哪些边？

答案提示：实际上，我们可以参照 Dijkstra 算法的处理思路，用另外一个二维数组来记录前驱顶点，然后，递归输出最短路径。具体的代码实现如下所示。其中，`pre[i][j]` 记录从顶点 `i` 到顶点 `j` 的路径中顶点 `j` 的前驱顶点的编号。

```
int v;
int g[v][v];
int dist[v][v];
int pre[v][v];
void floyd() {
  for (int i = 0; i < v; ++i) {
    for (int j = 0; j < v; ++j) {
      dist[i][j] = g[v][v];
      if (dist[i][j] != Integer.MAX_INF) {
        pre[i][j] = i;
      } else {
        pre[i][j] = -1;
      }
    }
  }
  for(int k=0; k<v; ++k) {
    for(int i=0; i<v; ++i) {
      for(int j=1; j<v; ++j) {
        if (dist[i][j] > dist[i][k]+dist[k][j]) {
          dist[i][j] = dist[i][k]+dist[k][j];
          pre[i][j] = pre[k][j];
        }
      }
    }
  }
}

void printShortestPath(int i, int j) {
  if (i == j) {
    return;
  }
  if (pre[i][j] == -1) {
    System.out.println("No path!");
    return;
  }
  printShortestPath(i, pre[i][j]);
  System.out.println(j);
}
```

## 9.6　启发式搜索：如何用 A* 算法实现游戏中的寻路功能

之前提到的“迷宫问题”是否可以借助 A* 算法来更快速地找到一个走出去的路线呢？如果可以，请读者具体讲一下该怎么做；如果不可以，请读者说明原因。

答案提示：迷宫问题不适合使用 A* 算法，因为 A* 算法本质上是利用了到终点的距离这一信息来辅助解决问题。而迷宫有很多折返，距终点的距离对于能否走出迷宫不是一个有效信息。因此，使用 A* 算法并不能加快找到走出去的路线的速度。

## 9.7　最小生成树：如何随机生成游戏中的迷宫地图

在本节，我们讲解了如何生成最小生成树。那么，如何生成次小生成树呢？（次小生成树就是树中所有边的权重和仅次于最小生成树的那个生成树。）

答案提示：这个问题可以借助最小生成树来实现。我们先求得图的最小生成树，然后逐一将树中的边从图中删除，然后在删除了一条边的图中求最小生成树。对比这 $V-1$ 个最小生成树，最小的那个就是次小生成树。

## 9.8　最大流：如何解决单身交友联谊中的最多匹配问题

本节介绍的是针对一个源点到一个汇点的最大流，如果网络流中有多个源点和多个汇点，又该如何实现多源点到多汇点的最大流？

答案提示：与最大二分匹配类似，这个问题也可以转化成一个源点到一个汇点的最大流问题。我们给图添加一个超级源点，并且在超级源点与其他源点之间建立权值为无穷大的边。同理，我们再添加一个超级汇点，并且在其他汇点与超级汇点之间建立权值为无穷大的边。多源点到多汇点的最大流问题就转化成了从超级源点到超级汇点的单源点单汇点最大流问题。

## 10.1　贪心算法：如何利用贪心算法实现霍夫曼编码

在一个非负整数 $a$ 中，我们希望从中移除 $k$ 个数字，让剩下的数字值最小，如何选择移除哪 $k$ 个数字呢？

答案提示：解决这个问题的核心是掌握这样一个规律，即先处理高位，再处理低位，尽

量让高位最小。移除 $k$ 位之后的数据的最高位肯定出现在 $a$ 的前 $k$+1 位之中。例如 $a$=321574，$k$=3，移除 3 位数字之后的数据的最高位肯定出现在 3215 之中。为了让最高位最小，我们要在这 $k$+1 个数中查找最小的那个，并将它前面的数字移除。针对这个例子，3215 中最小的数是 1，为了让它成为最终结果的最高位，需要把 3 和 2 移除。这样，1 就成了最高位。对于其他任何移除方法产生的数据，其最高位都不会比它小。因为现在已经移除了两个数字，所以 $k$ 变为了 1，$a$ 变成了 574。继续按照上面的处理思路，确定次高位。依此类推，直到所有的数字都确定为止。

这种处理思路比较好理解，但编程实现过程会比较烦琐。处理思想不变，还有一个更加简单的编程实现思路。我们借助栈，从高位数字开始逐一考察数据中的数字，如果数字大于栈顶元素，则入栈，如果数字小于栈顶元素，弹出栈顶元素，同时计数 $k$ 减 1，直到数字大于或等于栈顶元素或者 $k$ 为 0 时，停止出栈，然后将数字入栈。继续按照此规律处理后续数字。如果所有数据都已经入栈，$k$ 仍不为 0，则从栈中弹出 $k$ 个数。最后栈内数据就是最终求解的结果。

## 10.2 分治算法：谈一谈大规模计算框架 MapReduce 中的分治思想

在前面讲过的数据结构、算法和解决思路中，有哪些用到了分治算法思想？除此之外，在生活、工作中，还有没有用到分治算法思想的地方？读者可以自己回忆、总结一下，这对将零散的知识提炼成体系非常有帮助。

答案提示：分治思想用一句话总结，就是分而治之。分而治之这种思想随处可见。例如公司的管理，公司的员工由不同的部门来管理，不同的部门又分为不同的小组。在日常的开发中，分治思想也经常被用到，特别是针对一些大规模数据处理的问题。除此之外，现在流行的微服务架构也可以算是一种分治思想。

当然，分治思想不仅用在大的架构、技术解决方案中，还能指导算法的设计，如快速排序、归并排序、桶排序和二分查找等，或多或少地用到了分治思想。

## 10.3 回溯算法：从电影《蝴蝶效应》中学习回溯算法的核心思想

现在我们对本节讲到的 0-1 背包问题稍加改造，如果每个物品不仅重量不同，价值也不同，那么，如何在不超过背包承载的最大重量的前提下，让背包中所装物品的总价值最大？

答案提示：回溯是一种穷举算法，肯定也能解决这个问题。代码实现如下所示。

```
private int maxV = Integer.MIN_VALUE; //结果放到maxV中
private int[] items = {2,2,4,6,3};  //物品的重量
private int[] value = {3,4,8,9,6}; //物品的价值
private int n = 5; //物品的个数
private int w = 9; //背包可承载的最大重量
public void f(int i, int cw, int cv) { //调用f(0,0,0)
  if (cw == w || i == n) { //cw==w表示装满了,i==n表示物品都考察完了
```

```
    if (cv > maxV) maxV = cv;
    return;
  }
  f(i+1, cw, cv); //选择不装第i个物品
  if (cw + weight[i] <= w) {
    f(i+1,cw+weight[i], cv+value[i]); //选择装第i个物品
  }
}
```

## 10.4　初识动态规划：如何巧妙解决“双 11”购物时的凑单问题

对于“杨辉三角”，不知道读者是否听说过？我们现在对它进行一些改造。如图 A-2 所示，每个位置的数字可以随意填写，经过某个数字只能到达下面一层相邻的两个数字。假设我们从第一层开始往下移动，那么，把移动到最底层所经过的所有数字的和定义为路径的长度。请读者通过编程求出从第一层移动到最底层的最短路径长度。

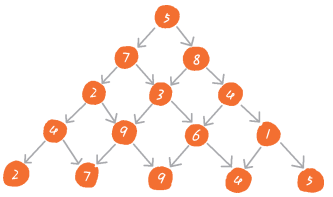


图 A-2　杨辉三角的改造版

答案提示：我们把杨辉三角存储在二维数组中。这个问题也可以使用回溯算法来解决。回溯代码对应的递归树，读者可以自己画一下，其中每个节点对应的状态包含 3 个值 $(i,j,dist)$，$i$、$j$ 分别表示行和列的下标，$dist$ 表示从根节点到此节点的路径长度。重复子问题存在到达某个节点的路径可能很多的情况，但是，我们只选择 $dist$ 最短的那个路径继续往下走，其他路径都可以丢弃。

基于上面的分析，我们记录每个节点对应的最短路径，基于上层节点的最短路径推导下层节点的最短路径，推导公式为：$S[i][j]=\min(S[i-1][j],S[i-1][j-1])+a[i][j]$，实际上，这就是 10.5.3 节讲到的状态转移方程。基于此来写代码就简单多了，如下所示。

```
int[][] matrix = {{5},{7,8},{2,3,4},{4,9,6,1},{2,7,9,4,5}};
int n = 4;
int yanghuiTriangle(int[][] matrix, int n) {
  int[][] state = new int[n][n];
  state[0][0] = matrix[0][0];
  for (int i = 1; i < n; i++) {
    for (int j = 0; j < i; j++) {
      if (j == 0) state[i][j] = state[i-1][j] + matrix[i][j];
      else if (j == i - 1) state[i][j] = state[i-1][j-1] + matrix[i][j];
      else {
        int leftTop = state[i-1][j-1];
        int rightTop = state[i-1][j];
        state[i][j] = Math.min(leftTop, rightTop) + matrix[i][j];
      }
    }
  }
  int minDis = Integer.MAX_VALUE;
  for (int i = 0; i < n; i++) {
    int distance = state[n-1][i];
    if (distance < minDis) minDis = distance;
  }
  return minDis;
}
```

## 10.5 动态规划理论：彻底理解最优子结构、无后效性和重复子问题

假设有几种不同的硬币，币值分别为 v1、v2……vn（单位是元）。如果要支付 w 元，那么最少需要多少个硬币？例如，有 3 种不同的硬币，币值分别为 1 元、3 元和 5 元。如果我们要支付 9 元，那么最少需要 3 个硬币（如 3 个 3 元的硬币）。

答案提示：实际上，这个问题可以看成 0-1 背包问题。我们把硬币看成物品，把硬币的面值看成物品的重量，把硬币的个数看成物品的价值。每个物品的价值相同，都是 1。每个物品的数量不做限制。我们可以把这个问题理解为如何选择放入哪些物品，装满背包并且总价值最小，也就是物品个数最少。具体的实现代码如下所示。

```
public int pay(int[] coins, int n, int value) {
  int[] minNums = new int[value + 1];
  for (int i = 0; i < value+1; ++i) {
    minNums[i] = Integer.MAX_VALUE;
  }
  minNums[0] = 0;
  for (int i = 0; i < n; ++i) {
    for (int j = value; j >= 0; --j) {
      if (minNums[j] == Integer.MAX_VALUE) continue;
      for (int k = 1; j + k * coins[i] <= value; k++) {
        int p = j + k* coins[i];
        minNums[p] = Math.min(minNums[p], minNums[j] + k);
      }
    }
  }
  if (minNums[value] == Integer.MAX_VALUE) return -1;
  return minNums[value];
}
```

实际上，这个问题更像爬楼梯问题：每步可以走 1、3 或 5 级台阶，要走完 9 级台阶，如何才能用最少的步数走完？对应的递归公式如式（A-1）所示。

$f(n)=1+\min(f(n-1),f(n-3),f(n-5))$，$f(n)$ 表示走 $n$ 级台阶最少需要的步数

$$\begin{aligned} f(0)&=0\\ f(1)&=1\\ f(3)&=1\\ f(5)&=1 \end{aligned} \qquad \text{（A-1）}$$

## 10.6 动态规划实战：如何实现搜索引擎中的拼写纠错功能

有一个数字序列，包含 n 个不同的数字，如何求出这个序列中的最长递增子序列的长度？例如 2、9、3、6、5、1 和 7 这样一组数字序列，它的最长递增子序列就是 2、3、5 和 7，因此，

其最长递增子序列的长度是 4。

答案提示：我们直接寻找状态转移方程。假设 maxl($i$) 表示以第 $i$ 个元素为结尾元素的最长递增子序列的长度。假设子问题 maxl(0),⋯,maxl($i$−1) 都已知，我们只需要选出 $ai$>$aj$（$j$=0 ～ $i$−1）的元素对应的 maxl 值，取其中的最大者，然后加 1，就是 maxl($i$) 的值。当求得所有的 maxl($i$)（$i$=0 ～ $n$−1）之后，再从中选出一个最大值，就是最后要求的结果。

根据上面的算法思路，对应的状态转移方程如式（A-2）所示。

$$\text{maxl}(i)=\max(\text{maxl}(j))+1，\text{其中 } j=0 \sim i-1，\text{并且满足 } ai>aj \qquad (A\text{-}2)$$

将上面的状态转移方程“翻译”成代码，如下所示。

```
int lis(int[] a, int n) {
  int[] maxl = new int[n];
  maxl[0] = 1;
  for (int i = 1; i < n; ++i) {
    int maxv = 0;
    for (int j = 0; j < i; ++j) {
      if (a[i] > a[j] && maxv < maxl[j]) {
        maxv = maxl[j];
      }
    }
    maxl[i] = maxv+1;
  }
  int res = 0;
  for (int i = 0; i < n; ++i) {
    if (res < maxl[i]) res = maxl[i];
  }
  return res;
}
```

## 11.1　实战 1：剖析 Redis 的常用数据类型对应的数据结构

1）Redis 中的很多数据类型，如哈希、有序集合等，是通过多种数据结构来实现的，为什么会这样设计呢？用一种固定的数据结构来实现不是更加简单吗？

答案提示：对于小规模数据的处理，我们在选择算法或数据结构的时候，不能只看时间复杂度、空间复杂度。之前，我们多次强调，大 O 表示法的复杂度只是表示执行时间随数据规模的增长趋势，而不是绝对的执行时间的大小。对于小规模数据的处理，为了提高性能，我们要考虑更多的因素。

Redis 在存储小规模数据的时候，采用了压缩列表这种数据结构。尽管压缩列表不能像数组那样支持按照下标随机访问，但因为数据规模比较小，顺序遍历并不会耗时太多。而且，因为其存储结构简单，节省内存，还对 CPU 缓存友好，所以性能表现不会比高级、复杂的数据结构差。

2）在本节，我们讲到了数据结构持久化的两种方式。对于二叉查找树，我们如何将它持久化到磁盘中呢？

答案提示：对于哈希表的序列化，文中提到了两种思路。对于二叉查找树的序列化，我们也可以借鉴这两种思路。

第一种思路就是直接将纯数据存储在磁盘中，当需要将磁盘中的数据反序列化加载到内存中时，再将数据重新构建成二叉查找树。我们知道，相同的一组数据对应的二叉查找树有多种。这样反序列化出来的二叉查找树与原来的二叉查找树可能形状就不相同了。

第二种思路是记录每个节点的父节点。在内存中，父节点是通过指针来表示的，也就是内存地址。序列化到磁盘不可能保存内存地址。为了解决这个问题，我们给每个节点编号，通过编号来标识父节点。如何编号呢？我们前序遍历二叉查找树，按照遍历的先后顺序依次给节点编号，并且按照遍历的先后顺序将节点存储到磁盘（数据 + 父节点编号）。

对于反序列化，也就是在内存中重新生成二叉查找树，我们顺序读取磁盘中的数据，为每个数据创建一个节点，并根据记录的父节点编号，插入到父节点的下面。如何快速地根据父节点编号找到父节点呢？我们可以利用哈希表，记录编号和节点的对应关系。

另外，我们可以精确地还原二叉查找树，也就是反序列化出来的二叉查找树与原来的二叉查找树形状一样。

## 11.2 实战 2：剖析搜索引擎背后的经典数据结构和算法

图的遍历方法有两种：深度优先搜索和广度优先搜索。本节讲到，搜索引擎中的“爬虫”是通过广度优先策略来“爬取”网页的。对于搜索引擎，为什么我们选择广度优先搜索策略，而不是深度优先搜索策略呢？

答案提示：从理论上来讲，如果搜索引擎可以把整个互联网的全部网页都“爬取”下来，那么无论使用广度优先搜索还是深度优先搜索，没有什么差别。而实际上，限于时间和资源，搜索引擎能“爬取”和索引的网页只是一小部分。为了利用有限的时间和资源“爬取”尽可能多的高权重网页，广度优先搜索的“爬取”策略就更加合适了。一般来说，我们会选择权重比较高的网页链接作为“爬虫”的种子链接，与种子链接离得越近的网页相应的权重会越高。利用广度优先搜索逐层遍历，会优先“爬取”与种子链接近的网页。而深度优先搜索会基于一个种子链接，一直递归“爬取”下去，完全不考虑权重因素。

## 11.3 实战 3：剖析微服务鉴权和限流背后的数据结构和算法

除用循环队列来实现滑动时间窗口限流算法之外，我们还可以利用哪些数据结构来实现呢？请读者对比一下这些数据结构在解决这个问题时的优劣。

答案提示：除使用循环队列实现滑动时间窗口限流算法之外，我们还可以使用双向链表或堆来实现。

基于双向链表，我们申请两个指针，分别指向链表的头节点和尾节点。头节点存储的是最新的接口访问，尾节点存储的是最早的接口访问。除此之外，我们使用一个变量 $k$，记录链表中的节点个数。新的接口访问到来时，我们用这个接口访问时间 $T$ 与尾节点存储的接口访问时

间对比，如果两者相差大于 1s，我们就将尾节点删除，并且 $k$—，继续用 $T$ 与新的尾节点存储的接口访问时间对比，直到两者之差小于 1s 时，我们看 $k$ 是否小于限流值，如果是，允许新的接口访问，并将这个接口访问时间包裹成节点，插入链表的头部，如果不是，则拒绝新的接口访问。

基于堆，我们根据访问时间建立小顶堆，也就是说，堆顶记录的是最早的接口访问时间。在新的接口访问到来时，我们用这个接口访问时间与堆顶的访问时间对比，如果相差大于 1s，就将堆顶元素删除。继续用这个接口访问时间与新的堆顶元素对比，直到相差小于 1s 时，我们看堆中的元素个数是否小于限流阈值，如果是，允许新接口访问，并将此接口访问时间包裹成节点，插入堆中，如果不是，则拒绝新的接口访问。

实际上，基于链表的解决方案，有点类似循环队列，不过，它涉及频繁的节点创建和删除操作，比循环队列更加消耗内存和时间。

## 11.4　实战 4：用学过的数据结构和算法实现短网址服务

如果需要额外支持用户自定义短网址功能（http://t.cn/{ 用户自定义部分 }），那么该如何改造上文提到的算法呢？

答案提示：我们在数据库表的短网址字段上建立唯一索引，尝试将用户自定义的短网址和原始网址插入数据库，如果插入成功，表示短网址可用，提示用户短网址生成成功。如果插入失败，就说明存在冲突，此时对应两种情况：如果数据库中的短网址对应的原始网址与当前正在处理的原始网址相同，则提示短网址生成成功；如果数据库中的短网址对应的原始网址与当前正在处理的原始网址不相同，则提示用户短网址已经被占用。