目　录

第 1 章

数组与字符串

1.1 判定是否互为字符重排

给定两个字符串，请编写程序，确定其中一个字符串的字符重新排列后，能否变成另一个字符串。

题目解法

首先确认变位词[①]（anagram）比较是否区分大小写？例如，God 是否为 dog 的变位词？同时问清楚是否要考虑空白字符。这里假定变位词比较区分大小写。

注意不同长度的字符串不可能互为重排字符串。

解法 1：排序字符串

若两个字符串互为重排字符串，那么它们拥有同一组字符，只是顺序不同。因此，对字符串排序后，组成这两个重排字符串的字符就会有相同的顺序。我们只需比较排序后的字符串。

```
String sort(String s) {
  char[] content = s.toCharArray();
  java.util.Arrays.sort(content);
  return new String(content);
}

boolean permutation(String s, String t) {
  if (s.length() != t.length()) {
    return false;
  }
  return sort(s).equals(sort(t));
}
```

解法 2：检查两个字符串的字符数是否相同

创建一个类似于散列表的数组（从第 4 行到第 7 行），将其每个字符映射到其字符出现的次数上。增加第一个字符串，然后减少第二个字符串，如果两者互为重排，则该数组最终将为 0。

若值为负值（一旦为负，则值将永为负值，不会为非 0），需提早终止。则数 0。因为，字符串长度相同时，增加的次数与减少的次数也相同，若数组无负值，则不会有正值。

```
boolean permutation(String s, String t) {
  if (s.length() != t.length()) return false; // 排列必须长度相同

  int[] letters = new int[128]; // 假设为 ASCII 字符
  for (int i = 0; i < s.length(); i++) {
```

① 变位词：一种把某个词或句子的字母位置顺序加以改变所形成的新词，英文叫做 Anagram，例如，said（说）就有 dais（讲台）这个变位词。

```
6       letters[s.charAt(i)]++;
7     }
8
9     for (int i = 0; i < t.length(); i++) {
10      letters[t.charAt(i)]--;
11      if (letters[t.charAt(i)] < 0) {
12        return false;
13      }
14    }
15    return true; // 字母没有负值，因此也没有正值
16  }
```

注意第 4 行的假设条件。在面试中，需要跟面试官核实一下字符集的大小。这里假设字符集为 ASCII。

1.2　一次编辑

字符串有三种编辑操作：插入一个字符、删除一个字符或者替换一个字符。给定两个字符串，编写一个函数判定它们是否只需要一次（或者零次）编辑。

示例：

```
pale,  ple  ->  true
pales, pale ->  true
pale,  bale ->  true
pale,  bake ->  false
```

题目解法

该题目可借助蛮力法。通过移除每一个字符（并比较），替换每一个字符（并比较），插入每一个字符（并比较）等方法，得到所有可能的字符串，然后检查只需一次编辑的字符串。

首先把插入和删除操作合并为一个步骤，让替换操作成为一个单独的步骤。

此外无需对所有字符串的插入、删除和替换操作进行检查，字符串的长度会告诉你需要检查哪一项操作。

```
1   boolean oneEditAway(String first, String second) {
2     if (first.length() == second.length()) {
3       return oneEditReplace(first, second);
4     } else if (first.length() + 1 == second.length()) {
5       return oneEditInsert(first, second);
6     } else if (first.length() - 1 == second.length()) {
7       return oneEditInsert(second, first);
8     }
9     return false;
10  }
11
12  boolean oneEditReplace(String s1, String s2) {
13    boolean foundDifference = false;
14    for (int i = 0; i < s1.length(); i++) {
15      if (s1.charAt(i) != s2.charAt(i)) {
16        if (foundDifference) {
17          return false;
18        }
19
20        foundDifference = true;
21      }
22    }
23    return true;
24  }
```

```
25
26 /* 检测是否可以通过向 s1 插入一个字符构造 s2 */
27 boolean oneEditInsert(String s1, String s2) {
28   int index1 = 0;
29   int index2 = 0;
30   while (index2 < s2.length() && index1 < s1.length()) {
31     if (s1.charAt(index1) != s2.charAt(index2)) {
32       if (index1 != index2) {
33         return false;
34       }
35       index2++;
36     } else {
37       index1++;
38       index2++;
39     }
40   }
41   return true;
42 }
```

该算法的时间复杂度为 $O(n)$，n 是较短字符串的长度（几乎所有合理的算法都为该时间复杂度）。

两个字符串长度相同，或只有一个延长时都不会增加运行时间。只有当两个字符串都很长的时候，时间复杂度才会增加。

我们或许会注意到代码 oneEditReplace 和代码 oneEditInsert 相差无几。因此，可以将二者合并为一个方法。

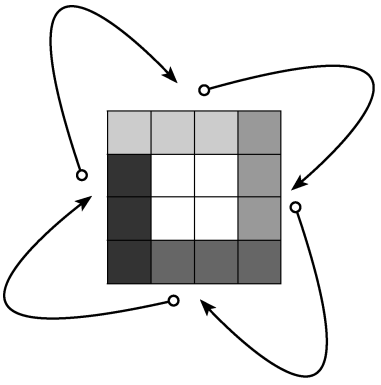
```
1  boolean oneEditAway(String first, String second) {
2    /* 检查长度 */
3    if (Math.abs(first.length() - second.length()) > 1) {
4      return false;
5    }
6
7    /* 获取较长和较短的字符串 */
8    String s1 = first.length() < second.length() ? first : second;
9    String s2 = first.length() < second.length() ? second : first;
10
11   int index1 = 0;
12   int index2 = 0;
13   boolean foundDifference = false;
14   while (index2 < s2.length() && index1 < s1.length()) {
15     if (s1.charAt(index1) != s2.charAt(index2)) {
16       /* 确保此处为发现的第一处不同 */
17       if (foundDifference) return false;
18       foundDifference = true;
19
20       if (s1.length() == s2.length()) { // 更换后，移动较短字符串的指针
21         index1++;
22       }
23     } else {
24       index1++; // 如果相匹配，就移动较短字符串的指针
25     }
26     index2++; // 总是移动较长字符串的指针
27   }
28   return true;
29 }
```

1.3　旋转矩阵

给定一幅由 *N*×*N* 矩阵表示的图像，其中每个像素的大小为 4 字节，编写一种方法，将图像旋转 90 度。不占用额外内存空间能否做到?

题目解法

要将矩阵旋转 90 度,最简单的做法就是一层一层进行旋转。对每一层执行环状旋转(circular rotation)：将上边移到右边，右边移到下边，下边移到左边，左边移到上边。



可按索引一个一个进行交换，具体做法如下。

```
for i = 0 to n
  temp = top[i];
  top[i] = left[i]
  left[i] = bottom[i]
  bottom[i] = right[i]
  right[i] = temp
```

可由外向内逐层执行（由内向外也可）。

下面是该算法的实现代码。

```
boolean rotate(int[][] matrix) {
  if (matrix.length == 0 || matrix.length != matrix[0].length) return false;
  int n = matrix.length;
  for (int layer = 0; layer < n / 2; layer++) {
    int first = layer;
    int last = n - 1 - layer;
    for(int i = first; i < last; i++) {
      int offset = i - first;
      int top = matrix[first][i]; // 存储上边

      // 左边移到上边
      matrix[first][i] = matrix[last-offset][first];

      // 下边移到左边
      matrix[last-offset][first] = matrix[last][last - offset];

      // 右边移到下边
      matrix[last][last - offset] = matrix[i][last];

      // 上边移到右边
      matrix[i][last] = top; // 前述存储的上边移到右边
    }
```

```
23    }
24    return true;
25  }
```

这个算法的时间复杂度为 $O(N^2)$，这已是最优解，因为任何算法都需要访问 N^2 的所有元素。

1.4 零矩阵

编写一种算法，若 $M \times N$ 矩阵中某个元素为 0，则将其所在的行与列清零。

题目解法

如直接遍历整个矩阵，只要发现值为 0 的元素，就将其所在的行与列清零。在读取被清零的行或列时，读到的尽是 0，于是所在行与列都得变成 0，最终整个矩阵的所有元素都会变为 0。

所以需要新建一个矩阵，标记 0 元素位置，然后，在第二次遍历矩阵时，将 0 元素所在行与列清零。这样空间复杂度为 $O(MN)$。

因为既然打算整行整和整列清 0，就没有必要记录 0 元素的确切位置？

下面是这个算法的实现代码。这里用两个数组分别记录包含 0 元素的所有行和列。在这之后，若所在行或列标记为 0，则将元素清为 0。

```
1   void setZeros(int[][] matrix) {
2     boolean[] row = new boolean[matrix.length];
3     boolean[] column = new boolean[matrix[0].length];
4
5     // 将值为 0 的元素的行、列索引保存
6     for (int i = 0; i < matrix.length; i++) {
7       for (int j = 0; j < matrix[0].length;j++) {
8         if (matrix[i][j] == 0) {
9           row[i] = true;
10          column[j] = true;
11        }
12      }
13    }
14
15    // 置空行
16    for (int i = 0; i < row.length; i++) {
17      if (row[i]) nullifyRow(matrix, i);
18    }
19
20    // 置空列
21    for (int j = 0; j < column.length; j++) {
22      if (column[j]) nullifyColumn(matrix, j);
23    }
24  }
25
26  void nullifyRow(int[][] matrix, int row) {
27    for (int j = 0; j < matrix[0].length; j++) {
28      matrix[row][j] = 0;
29    }
30  }
31
32  void nullifyColumn(int[][] matrix, int col) {
33    for (int i = 0; i < matrix.length; i++) {
34      matrix[i][col] = 0;
35    }
36  }
```

为了提高空间利用率，可以选用位向量替代布尔数组。存储空间的复杂度仍然为 $O(N)$。

通过使用第一行替代 row 数组，第一列替代 column 数组，可以将算法的空间复杂度降为 $O(1)$，其具体步骤如下。

(1) 检查第一行和第一列是否存在 0 元素，并根据结果设置 rowHasZero 和 columnHasZero 的值。

(2) 遍历矩阵中的其余元素，如果 matrix[i][j]为 0，则将 matrix[i][0]和 matrix[0][j]置为 0。

(3) 遍历矩阵中的其余元素，如果 matrix[i][0]为 0，则将第 i 行清零。

(4) 遍历矩阵中的其余元素，如果 matrix[0][j]为 0，则将第 j 行清零。

(5) 根据第(1)步的结果，如果需要则将第一行和第一列清零。

该方法的实现代码如下。

```
void setZeros(int[][] matrix) {
  boolean rowHasZero = false;
  boolean colHasZero = false;

  // 检查第一行是否有 0
  for (int j = 0; j < matrix[0].length; j++) {
    if (matrix[0][j] == 0) {
      rowHasZero = true;
      break;
    }
  }

  // 检查第一列是否有 0
  for (int i = 0; i < matrix.length; i++) {
    if (matrix[i][0] == 0) {
      colHasZero = true;
      break;
    }
  }

  // 检查数组其余元素是否有 0
  for (int i = 1; i < matrix.length; i++) {
    for (int j = 1; j < matrix[0].length;j++) {
      if (matrix[i][j] == 0) {
        matrix[i][0] = 0;
        matrix[0][j] = 0;
      }
    }
  }

  // 根据第一列的值置空行
  for (int i = 1; i < matrix.length; i++) {
    if (matrix[i][0] == 0) {
      nullifyRow(matrix, i);
    }
  }

  // 根据第一行的值置空列
  for (int j = 1; j < matrix[0].length; j++) {
    if (matrix[0][j] == 0) {
      nullifyColumn(matrix, j);
    }
  }

```

```
45    // 置空第一行
46    if (rowHasZero) {
47      nullifyRow(matrix, 0);
48    }
49
50    // 置空第一列
51    if (colHasZero) {
52      nullifyColumn(matrix, 0);
53    }
54  }
```

解题思路：先对行进行某操作，再对列做同样的操作。在面试中，你可以通过添加注释与TODO来简化代码。

1.9 字符串轮转

假定有一种 isSubstring 方法，可检查一个单词是否为其他字符串的子串。给定两个字符串 s1 和 s2，请编写代码检查 s2 是否为 s1 旋转而成，要求只能调用一次 isSubstring（比如，waterbottle 是 erbottlewat 旋转后的字符串）。

题目解法

假定 s2 由 s1 旋转而成，那么，我们可以找出旋转点在哪儿。例如，若以 wat 对 waterbottle 旋转，就会得到 erbottlewat。在旋转字符串时，会把 s1 切分为两部分：x 和 y，并将它们重新组合成 s2。

```
s1 = xy = waterbottle
x = wat
y = erbottle
s2 = yx = erbottlewat
```

因此，我们需要直接调用 isSubstring(s1s1, s2)即可。因为不论 x 和 y 之间的分割点在何处，我们会发现 yx 肯定是 xyxy 的子串，也即，s2 总是 s1s1 的子串。

下面是上述算法的实现代码。

```
1   boolean isRotation(String s1, String s2) {
2     int len = s1.length();
3     /* 检查 s1 和 s2 长度相等且非空 */
4     if (len == s2.length() && len > 0) {
5       /* 在新空间中将 s1 与 s1 合并 */
6       String s1s1 = s1 + s1;
7       return isSubstring(s1s1, s2);
8     }
9     return false;
10  }
```

该算法的时间复杂度随 isSubstring 的时间复杂度的不同而变化。如果假设 isSubstring 的运行时间是 $O(A+B)$（对于长度分别为 A 和 B 的两个字符串），那么 isRotation 的运行时间则为 $O(N)$。

第 2 章

链　　表

2.1　移除重复节点

编写代码，移除未排序链表中的重复节点。

进阶：如果不得使用临时缓冲区，该怎么解决？

题目解法

要想移除链表中的重复节点，需要设法记录有哪些是重复的。

直接迭代访问整个链表，将每个节点加入散列表。若发现有重复元素，则将该节点从链表中移除，然后继续迭代。因为这个题目使用了链表，所以只需扫描一次。

```
1  void deleteDups(LinkedListNode n) {
2    HashSet<Integer> set = new HashSet<Integer>();
3    LinkedListNode previous = null;
4    while (n != null) {
5      if (set.contains(n.data)) {
6        previous.next = n.next;
7      } else {
8        set.add(n.data);
9        previous = n;
10     }
11     n = n.next;
12   }
13 }
```

代码的时间复杂度为 $O(N)$，其中 N 为链表节点数目。

进阶：不得使用缓冲区

如不借助额外的缓冲区，可以用两个指针来迭代：current 迭代访问整个链表，runner 用于检查后续的节点是否重复。

```
1  void deleteDups(LinkedListNode head) {
2    LinkedListNode current = head;
3    while (current != null) {
4      /* 删除所有其余有相同值的节点 */
5      LinkedListNode runner = current;
6      while (runner.next != null) {
7        if (runner.next.data == current.data) {
8          runner.next = runner.next.next;
9        } else {
10         runner = runner.next;
11       }
12     }
13     current = current.next;
14   }
15 }
```

这段代码的空间复杂度为 $O(1)$，时间复杂度为 $O(N^2)$。

2.2 删除中间节点

实现一种算法，删除单向链表中间的某个节点（除了第一个和最后一个节点，不一定是中间节点），假定你只能访问该节点。

示例：

输入：单向链表 a->b->c->d->e->f 中的节点 c

结果：不返回任何数据，但该链表变为 a->b->d->e->f

题目解法

因访问不到链表的首节点，只能访问那个待删除节点。所以直接将后继节点的数据复制到当前节点，然后删除这个后继节点。

下面是该算法的实现代码。

```
boolean deleteNode(LinkedListNode n) {
  if (n == null || n.next == null) {
    return false; // 失败
  }
  LinkedListNode next = n.next;
  n.data = next.data;
  n.next = next.next;
  return true;
}
```

注意，若待删除节点为链表的尾节点，这个问题无解。

2.3 分割链表

编写程序以 x 为基准分割链表，使得所有小于 x 的节点排在大于或等于 x 的节点之前。如果链表中包含 x，x 只需出现在小于 x 的元素之前（如下所示）。分割元素 x 只需处于“右半部分”即可，其不需要被置于左右两部分之间。

示例：

输入：3 -> 5 -> 8-> 5 -> 10 -> 2 -> 1 [分节点为 5]

输出：3 -> 1 -> 2 -> 10 -> 5-> 5 -> 8

题目解法

如本题描述的是一个数组，如何移动元素则要非常谨慎。

但在链表当中，可以通过创建两个链表完成该操作，其中一个链表包含小于 x 的元素，而另一个链表包含大于或等于 x 的元素。

遍历链表，不断地将元素插入到 before 链表和 after 链表当中。当到达链表尾部并完成了分离操作后，将得到的两个链表合并。

如需链表中元素保持“稳定”可使用以下代码实现。

```
/* 将链表头节点和其节点值传递到分割链表 */
LinkedListNode partition(LinkedListNode node, int x) {
  LinkedListNode beforeStart = null;
  LinkedListNode beforeEnd = null;
  LinkedListNode afterStart = null;
  LinkedListNode afterEnd = null;

  /* 分割链表 */
```

```
    while (node != null) {
      LinkedListNode next = node.next;
      node.next = null;
      if (node.data < x) {
        /* 将节点插入到 before 链表尾部 */
        if (beforeStart == null) {
          beforeStart = node;
          beforeEnd = beforeStart;
        } else {
          beforeEnd.next = node;
          beforeEnd = node;
        }
      } else {
        /* 将节点插入到 after 链表尾部 */
        if (afterStart == null) {
          afterStart = node;
          afterEnd = afterStart;
        } else {
          afterEnd.next = node;
          afterEnd = node;
        }
      }
      node = next;
    }

    if (beforeStart == null) {
      return afterStart;
    }

    /* 合并 after 链表和 before 链表 */
    beforeEnd.next = afterStart;
    return beforeStart;
  }
```

如果不介意链表中的元素是否保持“稳定”。

我们创建了一个“新”链表（借助已有节点），将大于基准点的元素加入到链表尾部，将小于基准点的元素加入到链表头部，以便整理链表。

```
LinkedListNode partition(LinkedListNode node, int x) {
  LinkedListNode head = node;
  LinkedListNode tail = node;

  while (node != null) {
    LinkedListNode next = node.next;
    if (node.data < x) {
      /* 在头部插入节点 */
      node.next = head;
      head = node;
    } else {
      /* 在尾部插入节点 */
      tail.next = node;
      tail = node;
    }
    node = next;
  }
  tail.next = null;

  // 头部已改变，所以我们需要将其返回
  return head;
}
```

2.4 回文链表

编写一个函数，检查链表是否为回文。

题目解法

将回文（palindrome）定义为 `0 -> 1 -> 2 -> 1 -> 0`。若链表是回文，不管正着看还是反着看，都是一样的。由此可以得出解法一。

解法 1：反转并比较

反转整个链表，然后比较反转链表和原始链表。若两者相同，则该链表为回文。

注意，在比较原始链表和反转链表时，只需比较链表的前半部分。若原始链表和反转链表的前半部分相同，两者的后半部分肯定相同。

```
boolean isPalindrome(LinkedListNode head) {
  LinkedListNode reversed = reverseAndClone(head);
  return isEqual(head, reversed);
}

LinkedListNode reverseAndClone(LinkedListNode node) {
  LinkedListNode head = null;
  while (node != null) {
    LinkedListNode n = new LinkedListNode(node.data); // 复制
    n.next = head;
    head = n;
    node = node.next;
  }
  return head;
}

boolean isEqual(LinkedListNode one, LinkedListNode two) {
  while (one != null && two != null) {
    if (one.data != two.data) {
      return false;
    }
    one = one.next;
    two = two.next;
  }
  return one == null && two == null;
}
```

该段代码被模块化为 `reverse` 函数和 `isEqual` 函数。

解法 2：迭代法

要想检查链表的前半部分是否为后半部分反转而成，只需将链表前半部分反转，可以利用栈来实现。

需要将前半部分节点入栈。根据链表长度已知与否，入栈有两种方式。

- 若链表长度已知，可以用标准 `for` 循环迭代访问前半部分节点，将每个节点入栈。当然，要小心处理链表长度为奇数的情况。
- 若链表长度未知，可以利用本章开头描述的快慢 runner 方法迭代访问链表。在迭代循环的每一步，将慢速 runner 的数据入栈。在快速 runner 抵达链表尾部时，慢速 runner 刚好位于链表中间位置。至此，栈里就存放了链表前半部分的所有节点，不过顺序是相反的。

接下来，只需迭代访问链表余下节点。每次迭代时，比较当前节点和栈顶元素，若完成迭代时比较结果完全相同，则该链表是回文序列。

```
boolean isPalindrome(LinkedListNode head) {
  LinkedListNode fast = head;
  LinkedListNode slow = head;

  Stack<Integer> stack = new Stack<Integer>();

  /* 将链表前半部分元素插入到栈中。当快指针（2 倍速移动）
   * 移动到链表尾部，我们得知已经到达中点 */
  while (fast != null && fast.next != null) {
    stack.push(slow.data);
    slow = slow.next;
    fast = fast.next.next;
  }

  /* 因为有奇数个节点，所以跳过中间节点 */
  if (fast != null) {
    slow = slow.next;
  }

  while (slow != null) {
    int top = stack.pop().intValue();

    /* 如果值不同，则不是回文 */
    if (top != slow.data) {
      return false;
    }
    slow = slow.next;
  }
  return true;
}
```

解法 3：递归法

首先，简要介绍下面的解法用到的记号：用记号 `Kx` 表示节点时，变量 `K` 指示节点数据的值，而 `x`（取 `f` 或 `b`）指示该节点是值为 `K` 的前方节点还是后方节点。例如，在下面的链表中，节点 `2b` 指的是值为 2 的第二个（`b` → back，即后方）节点。

用递归法。比较元素 0 和元素 $n-1$，元素 1 和元素 $n-2$，元素 2 和元素 $n-3$，以此类推，直至中间元素。

例如：

```
0 ( 1 ( 2 ( 3 ) 2 ) 1 ) 0
```

运用这种方法，首先必须知道递归的基线条件，即什么时候到达中间元素。每次递归调用传入 `length - 2` 为长度，当长度等于 0 或 1 时，表明当前已处于链表中间位置。这是因为 `length` 每次都会缩减 2。一旦递归进行了 $N/2$ 次，`length` 将会减至 0。

```
recurse(Node n, int length) {
  if (length == 0 || length == 1) {
    return [something]; // 中点
  }
  recurse(n.next, length - 2);
  ...
}
```

至此，`isPalindrome` 方法便初具雏形了，该算法的实质是比较节点 `i` 和节点 `n-i`，检查链表是否为回文序列。

仔细分析下面的调用栈。

```
v1 = isPalindrome: list = 0 ( 1 ( 2 ( 3 ) 2 ) 1 ) 0. length = 7
  v2 = isPalindrome: list = 1 ( 2 ( 3 ) 2 ) 1 ) 0. length = 5
    v3 = isPalindrome: list = 2 ( 3 ) 2 ) 1 ) 0. length = 3
      v4 = isPalindrome: list = 3 ) 2 ) 1 ) 0. length = 1
      returns v3
    returns v2
  returns v1
returns ?
```

在上面的调用栈中，每次调用都会比较其头节点和链表后半部分对应节点，检查链表是否为回文序列，执行如下操作。

- 第 1 行需要比较节点 `0f` 和节点 `0b`；
- 第 2 行需要比较节点 `1f` 和节点 `1b`；
- 第 3 行需要比较节点 `2f` 和节点 `2b`；
- 第 4 行需要比较节点 `3f` 和节点 `3b`。

若将上面的栈倒过来，按如下顺序传回节点，只需这样做。

- 第 4 行发现传入节点为中间节点（因为 `length = 1`），传回 `head.next`，其中 `head` 为节点 `3`，因此 `head.next` 为节点 `2b`。
- 第 3 行比较头节点即节点 `2f` 和 `returned_node`（上次递归调用返回的值）即节点 `2b`。若两个节点的值相等，则传送节点 `1b` 的引用（`returned_node.next`）至第 2 行。
- 第 2 行比较头节点（节点 `1f`）和 `returned_node`（节点 `1b`）。若两个节点的值相等，则传送节点 `0b` 的引用（或 `returned_node.next`）至第 1 行。
- 第 1 行比较头节点（节点 `0f`）和 `returned_node`（节点 `0b`）。若两个节点的值相等，则返回 `true`。

总而言之，每次调用都会比较其头节点和 `returned_node`，然后回传 `returned_node.next`。最终每个节点 `i` 都会与节点 `n-i` 进行比较。只要有任意一对节点的值不相等，就立即返回 `false`，调用栈的上一级调用都会检查这个布尔值。

等一等，你可能会问，一会儿说要返回一个布尔值，一会儿说要返回一个节点，到底要返回什么？

两个都要返回。我们创建了一个包含布尔值和节点两个成员的简单类，调用时只需返回该类的实例。

```
class Result {
  public LinkedListNode node;
  public boolean result;
}
```

下面举例说明示例链表每次递归调用的参数和返回值。

```
isPalindrome: list = 0 ( 1 ( 2 ( 3 ( 4 ) 3 ) 2 ) 1 ) 0. len = 9
  isPalindrome: list = 1 ( 2 ( 3 ( 4 ) 3 ) 2 ) 1 ) 0. len = 7
    isPalindrome: list = 2 ( 3 ( 4 ) 3 ) 2 ) 1 ) 0. len = 5
      isPalindrome: list = 3 ( 4 ) 3 ) 2 ) 1 ) 0. len = 3
        isPalindrome: list = 4 ) 3 ) 2 ) 1 ) 0. len = 1
        returns node 3b, true
      returns node 2b, true
    returns node 1b, true
  returns node 0b, true
returns null, true
```

填入细节。

```
boolean isPalindrome(LinkedListNode head) {
  int length = lengthOfList(head);
  Result p = isPalindromeRecurse(head, length);
  return p.result;
}

Result isPalindromeRecurse(LinkedListNode head, int length) {
  if (head == null || length <= 0) { // 偶数个节点
    return new Result(head, true);
  } else if (length == 1) { // 奇数个节点
    return new Result(head.next, true);
  }

  /* 在子链表上递归 */
  Result res = isPalindromeRecurse(head.next, length - 2);

  /* 如果递归调用返回非回文，则向上传递失败信息 */
  if (!res.result || res.node == null) {
    return res;
  }

  /* 检查与另一侧的节点值是否匹配 */
  res.result = (head.data == res.node.data);

  /* 返回对应的节点 */
  res.node = res.node.next;

  return res;
}

int lengthOfList(LinkedListNode n) {
  int size = 0;
  while (n != null) {
    size++;
    n = n.next;
  }
  return size;
}
```

使用 Java 实现，必需专门创建一个 `Result` 类。

若用 C 或 C++实现，可以传入一个指针的指针。

```
bool isPalindromeRecurse(Node head, int length, Node** next) {
  ...
}
```

第 3 章

栈与队列

3.1 栈的最小值

请设计一个栈，除了 pop 与 push 函数，还支持 min 函数，其可返回栈元素中的最小值。执行 push、pop 和 min 操作的时间复杂度必须为 $O(1)$。

题目解法

一种解法是在 Stack 类里添加一个 int 型的 minValue。当 minValue 出栈时，搜索整个栈，找出新的最小值。可惜，这不符合入栈和出栈操作时间为 $O(1)$的要求。

为进一步理解这个问题，下面用一个简短的例子加以说明。

```
push(5); // 栈为{5}，最小值为 5
push(6); // 栈为{6，5}，最小值为 5
push(3); // 栈为{3，6，5}，最小值为 3
push(7); // 栈为{7，3，6，5}，最小值为 3
pop(); // 弹出 7，栈为{3，6，5}，最小值为 3
pop(); // 弹出 3，栈为{6，5}，最小值为 5
```

注意观察，当栈回到之前的状态（{6，5}）时，最小值也回到之前的状态（5）。

这样，只要每个节点记录当前最小值即可。要找到 min，直接查看栈顶元素就能得到最小值。

当一个元素入栈时，该元素会记下当前最小值，将 min 记录在自身数据结构的 min 成员中。

```
1  public class StackWithMin extends Stack<NodeWithMin> {
2    public void push(int value) {
3      int newMin = Math.min(value, min());
4      super.push(new NodeWithMin(value, newMin));
5    }
6
7    public int min() {
8      if (this.isEmpty()) {
9        return Integer.MAX_VALUE; // 错误的值
10     } else {
11       return peek().min;
12     }
13   }
14 }
15
16 class NodeWithMin {
17   public int value;
18   public int min;
19   public NodeWithMin(int v, int min){
20     value = v;
21     this.min = min;
22   }
23 }
```

但是，这种做法有个缺点：当栈很大时，每个元素都要记录 min，就会浪费大量空间。所以建议利用其他的栈来记录这些 min。

```
public class StackWithMin2 extends Stack<Integer> {
  Stack<Integer> s2;
  public StackWithMin2() {
    s2 = new Stack<Integer>();
  }

  public void push(int value){
    if (value <= min()) {
      s2.push(value);
    }
    super.push(value);
  }

  public Integer pop() {
    int value = super.pop();
    if (value == min()) {
      s2.pop();
    }
    return value;
  }

  public int min() {
    if (s2.isEmpty()) {
      return Integer.MAX_VALUE;
    } else {
      return s2.peek();
    }
  }
}
```

这样我们只需存储几项数据：第二个栈（只有一个元素）以及栈本身数据结构的若干成员。

3.2　堆盘子

设想有一堆盘子，堆太高可能会倒下来。因此，在现实生活中，盘子堆到一定高度时，我们就会另外堆一堆盘子。请实现数据结构 SetOfStacks，模拟这种行为。SetOfStacks 应该由多个栈组成，并且在前一个栈填满时新建一个栈。此外，SetOfStacks.push()和 SetOfStacks.pop()应该与普通栈的操作方法相同（也就是说，pop()返回的值，应该跟只有一个栈时的情况一样）。

进阶：实现一个 popAt(int index)方法，根据指定的子栈，执行 pop 操作。

题目解法

在这个问题中，根据题意，数据结构应该类似下面这样。

```
class SetOfStacks {
  ArrayList<Stack> stacks = new ArrayList<Stack>();
  public void push(int v) { ... }
  public int pop() { ... }
}
```

push()的行为必须跟单一栈的一样，这就意味着 push()要对栈数组的最后一个栈调用 push()。不过，若最后一个栈被填满，就需新建一个栈。实现代码大致如下。

```
void push(int v) {
  Stack last = getLastStack();
  if (last != null && !last.isFull()) { // 加入到 last 栈中
    last.push(v);
  } else { // 必须创建新栈
    Stack stack = new Stack(capacity);
    stack.push(v);
    stacks.add(stack);
  }
}
```

若最后一个栈为空（执行出栈操作后），就必须从栈数组中移除这个栈。

```
int pop() {
  Stack last = getLastStack();
  if (last == null) throw new EmptyStackException();
  int v = last.pop();
  if (last.size == 0) stacks.remove(stacks.size() - 1);
  return v;
}
```

进阶：实现 popAt(int index)

我们可以设想一个“推入”动作。从栈 1 弹出元素时，我们需要移出栈 2 的栈底元素，并将其推到栈 1 中。随后，将栈 3 的栈底元素推入栈 2，将栈 4 的栈底元素推入栈 3，以此类推。

如果不执行“推入”操作，当面临所有栈（最后一个栈除外）都填满时，就会导致继续推进。不过这个问题没有标准答案，你可以和面试官进行讨论。

```
public class SetOfStacks {
  ArrayList<Stack> stacks = new ArrayList<Stack>();
  public int capacity;
  public SetOfStacks(int capacity) {
    this.capacity = capacity;
  }

  public Stack getLastStack() {
    if (stacks.size() == 0) return null;
    return stacks.get(stacks.size() - 1);
  }

  public void push(int v) { /* 见前述代码 */ }
  public int pop() { /* 见前述代码 */ }
  public boolean isEmpty() {
    Stack last = getLastStack();
    return last == null || last.isEmpty();
  }

  public int popAt(int index) {
    return leftShift(index, true);
  }

  public int leftShift(int index, boolean removeTop) {
    Stack stack = stacks.get(index);
    int removed_item;
    if (removeTop) removed_item = stack.pop();
    else removed_item = stack.removeBottom();
    if (stack.isEmpty()) {
      stacks.remove(index);
    } else if (stacks.size() > index + 1) {
      int v = leftShift(index + 1, false);
      stack.push(v);
```

```
    }
    return removed_item;
  }
}

public class Stack {
  private int capacity;
  public Node top, bottom;
  public int size = 0;

  public Stack(int capacity) { this.capacity = capacity; }
  public boolean isFull() { return capacity == size; }

  public void join(Node above, Node below) {
    if (below != null) below.above = above;
    if (above != null) above.below = below;
  }

  public boolean push(int v) {
    if (size >= capacity) return false;
    size++;
    Node n = new Node(v);
    if (size == 1) bottom = n;
    join(n, top);
    top = n;
    return true;
  }

  public int pop() {
    Node t = top;
    top = top.below;
    size--;
    return t.value;
  }

  public boolean isEmpty() {
    return size == 0;
  }

  public int removeBottom() {
    Node b = bottom;
    bottom = bottom.above;
    if (bottom != null) bottom.below = null;
    size--;
    return b.value;
  }
}
```

但要解决这个问题需要编写大量代码（面试时，可能不会要求完整解答）。

但将代码分离出来，写成独立的，比如 `popAt` 可以调用的 `leftShift`。这样，代码就会更加清晰，在处理细节之前，也有机会先铺设好代码的骨架。

3.3　动物收容所

有家动物收容所只收容狗与猫，且严格遵守“先进先出”的原则。收养人不能自由选择收养对象，仅可在以下方式中选择：

⑴ 收养所中最老的动物

⑵ 收养所中最老的猫

⑶ 收养所中最老的狗

注：由动物进入收容所的时间长短，定义老的程度。

请创建适用于这个系统的数据结构，实现各种操作方法，比如 enqueue、dequeueAny、dequeueDog 和 dequeueCat。允许使用 Java 内置的 LinkedList 数据结构。

题目解法

方法一，我们可以只维护一个队列。这么做的话，dequeueAny（收养任意一种动物）实现起来很简单，但 dequeueDog（收养狗）和 dequeueCat（收养猫）就要迭代访问整个队列，才能找到第一只该被收养的狗或猫。这会增加整个解法的复杂度，降低执行效率。

方法二，只需为狗和猫各自创建一个队列，然后将两者放进名为 AnimalQueue 的包裹类，并且存储某种形式的时间戳，以标记每只动物进入队列（即收容所）的时间。当调用 dequeueAny 时，查看狗队列和猫队列的首部，并返回“最老”的那一只。

```
abstract class Animal {
  private int order;
  protected String name;
  public Animal(String n) { name = n; }
  public void setOrder(int ord) { order = ord; }
  public int getOrder() { return order; }

  /* 比较动物的顺序以返回较早的项目 */
  public boolean isOlderThan(Animal a) {
    return this.order < a.getOrder();
  }
}

class AnimalQueue {
  LinkedList<Dog> dogs = new LinkedList<Dog>();
  LinkedList<Cat> cats = new LinkedList<Cat>();
  private int order = 0; // 作为时间戳使用

  public void enqueue(Animal a) {
    /* order 被作为时间戳使用，这样一来我们可以比较一只猫和一只狗的插入顺序 */
    a.setOrder(order);
    order++;

    if (a instanceof Dog) dogs.addLast((Dog) a);
    else if (a instanceof Cat) cats.addLast((Cat)a);
  }

  public Animal dequeueAny() {
    /* 查看猫和狗队列的顶部元素，对最久的元素做出列操纵 */
    if (dogs.size() == 0) {
      return dequeueCats();
    } else if (cats.size() == 0) {
      return dequeueDogs();
    }

    Dog dog = dogs.peek();
    Cat cat = cats.peek();
    if (dog.isOlderThan(cat)) {
      return dequeueDogs();
    } else {
      return dequeueCats();
    }
  }
```

```
44
45    public Dog dequeueDogs() {
46      return dogs.poll();
47    }
48
49    public Cat dequeueCats() {
50      return cats.poll();
51    }
52  }
53
54  public class Dog extends Animal {
55    public Dog(String n) { super(n); }
56  }
57
58  public class Cat extends Animal {
59    public Cat(String n) { super(n); }
60  }
```

dequeueAny()方法需要同时支持返回 Dog 对象与 Cat 对象，因此，Dog 类与 Cat 类均继承于 Animal 类至关重要。

order 可以是一个有着真实日期和时间的时间戳。这样做的优势在于不再需要设置并维护一个数字化的顺序。如果出现两个时间戳相同的最老动物，返回任意一只。

第 4 章

树 与 图

4.1 节点间通路

给定有向图，设计一个算法，找出两个节点之间是否存在一条路径。

题目解法

只需通过图的遍历，比如深度优先搜索或广度优先搜索，就能解决这个问题。我们从两个节点的其中一个出发，在遍历过程中，检查是否能找到另一个节点。在这个算法中，访问过的节点都应标记为“已访问”，以免循环和重复访问节点。

下面是广度优先搜索的迭代实现。

```
enum State { Unvisited, Visited, Visiting; }

boolean search(Graph g, Node start, Node end) {
  if (start == end) return true;

  // 按队列使用
  LinkedList<Node> q = new LinkedList<Node>();

  for (Node u : g.getNodes()) {
    u.state = State.Unvisited;
  }
  start.state = State.Visiting;
  q.add(start);
  Node u;
  while (!q.isEmpty()) {
    u = q.removeFirst(); // 例如出列
    if (u != null) {
      for (Node v : u.getAdjacent()) {
        if (v.state == State.Unvisited) {
          if (v == end) {
            return true;
          } else {
            v.state = State.Visiting;
            q.add(v);
          }
        }
      }
      u.state = State.Visited;
    }
  }
  return false;
}
```

碰到这类问题时，建议面试官探讨一下广度优先搜索和深度优先搜索的利弊。例如，深度优先搜索实现起来比较简单，因为只需简单的递归即可。广度优先搜索很适合用来查找最短路

径，而深度优先搜索在访问邻近节点之前，可能会先深度遍历其中一个邻近节点。

4.2　最小高度树

给定一个有序整数数组，元素各不相同且按升序排列，编写一个算法，创建一棵高度最小的二叉搜索树。

题目解法

要创建一棵高度最小的树，就必须让左右子树的节点数量尽量接近，也就是说，要让数组中间的值成为根节点。

继续以类似方式构造整棵树。数组每一区段的中间元素成为子树的根节点，左半部分成为左子树，右半部分成为右子树。

一种实现方式是使用简单的 `root.insertNode(int v)`方法，从根节点开始，以递归方式将值 v 插入树中。每次插入操作都要遍历整棵树，用时为 $O(N \log N)$。

另一种做法是以递归方式运用 `createMinimalBST` 方法，传入数组的一个区段，并返回最小树的根节点。这样可以减少遍历，提升效率。

该算法简述如下。

(1) 将数组中间位置的元素插入树中。

(2) 将数组左半边元素插入左子树。

(3) 将数组右半边元素插入右子树。

(4) 递归处理。

下面是该算法的实现代码。

```
1  TreeNode createMinimalBST(int array[]) {
2    return createMinimalBST(array, 0, array.length - 1);
3  }
4
5  TreeNode createMinimalBST(int arr[], int start, int end) {
6    if (end < start) {
7      return null;
8    }
9    int mid = (start + end) / 2;
10   TreeNode n = new TreeNode(arr[mid]);
11   n.left = createMinimalBST(arr, start, mid - 1);
12   n.right = createMinimalBST(arr, mid + 1, end);
13   return n;
14 }
```

编写注意差一（off-by-one）错误，并重点测试。

4.3　检查平衡性

实现一个函数，检查二叉树是否平衡。在这个问题中，平衡树的定义如下：任意一个节点，其两棵子树的高度差不超过 1。

题目解法

根据该定义可以得到一种解法，即直接递归访问整棵树，计算每个节点两棵子树的高度。

```
1  int getHeight(TreeNode root) {
2    if (root == null) return -1; // 基本情况
3    return Math.max(getHeight(root.left), getHeight(root.right)) + 1;
```

```
4  }
5
6  boolean isBalanced(TreeNode root) {
7    if (root == null) return true; // 基本情况
8
9    int heightDiff = getHeight(root.left) - getHeight(root.right);
10   if (Math.abs(heightDiff) > 1) {
11     return false;
12   } else { // 递归
13     return isBalanced(root.left) && isBalanced(root.right);
14   }
15 }
```

这个算法的时间复杂度为 $O(N \log N)$。代码会递归访问每个节点的整棵子树，getHeight 会被反复调用计算同一个节点的高度。

getHeight 不仅可以检查高度，还能检查这棵树是否平衡。子树不平衡时直接返回一个错误代码即可。

改进过的算法会从根节点递归向下检查每棵子树的高度。我们会通过 checkHeight 方法，以递归方式获取每个节点左右子树的高度。若子树是平衡的，则 checkHeight 返回该子树的实际高度。若子树不平衡，则 checkHeight 返回一个错误代码。checkHeight 会立即中断执行，并返回一个错误代码。

> 我们可以将 Integer.MIN_VALUE 作为错误代码。

下面是该算法的实现代码。

```
1  int checkHeight(TreeNode root) {
2    if (root == null) return -1;
3
4    int leftHeight = checkHeight(root.left);
5    if (leftHeight == Integer.MIN_VALUE) return Integer.MIN_VALUE; // 向上传递错误
6
7    int rightHeight = checkHeight(root.right);
8    if (rightHeight == Integer.MIN_VALUE) return Integer.MIN_VALUE; // 向上传递错误
9
10   int heightDiff = leftHeight - rightHeight;
11   if (Math.abs(heightDiff) > 1) {
12     return Integer.MIN_VALUE; // 发现错误，把它传回来
13   } else {
14     return Math.max(leftHeight, rightHeight) + 1;
15   }
16 }
17
18 boolean isBalanced(TreeNode root) {
19   return checkHeight(root) != Integer.MIN_VALUE;
20 }
```

这段代码需要 $O(N)$的时间和 $O(H)$的空间，其中 H 为树的高度。

4.4 合法二叉搜索树

实现一个函数，检查一棵二叉树是否为二叉搜索树。

题目解法

此题有两种不同的解法:第一种是利用中序遍历,第二种则建立在 left <= current < right 这项特性之上。

解法 1：中序遍历

看到此题，首先想到的可能是中序遍历，即将所有元素复制到数组中，然后检查该数组是否有序。这种解法会占用一点儿额外的内存，但大部分情况下都奏效。

唯一的问题在于，它无法正确处理树中的重复值。例如，该算法无法区分下面这两棵树（其中一棵是无效的），因为两者的中序遍历结果相同。

不过，要是假定这棵树不得包含重复值，那么这种做法还是行之有效的。该方法的伪码大致如下。

```
int index = 0;
void copyBST(TreeNode root, int[] array) {
  if (root == null) return;
  copyBST(root.left, array);
  array[index] = root.data;
  index++;
  copyBST(root.right, array);
}

boolean checkBST(TreeNode root) {
  int[] array = new int[root.size];
  copyBST(root, array);
  for (int i = 1; i < array.length; i++) {
    if (array[i] <= array[i - 1]) return false;
  }
  return true;
}
```

注意，这里必须记录数组在逻辑上的“尾部”，用它来分配空间以存储所有元素。

在进行比较时，直接记下最后的元素，就可不使用数组。

下面是该算法的实现代码。

```
Integer last_printed = null;
boolean checkBST(TreeNode n) {
  if (n == null) return true;

  // 对左子树递归、检查
  if (!checkBST(n.left)) return false;

  // 检查当前节点
  if (last_printed != null && n.data <= last_printed) {
    return false;
  }
  last_printed = n.data;

  // 对右子树递归、检查
  if (!checkBST(n.right)) return false;

  return true; // 完成
}
```

我们使用了 `Integer` 而非 `int` 从而了解 `last_printed` 是否已经被赋值。

使用包裹类存放这个整数值，就可不使用静态变量如下所示。

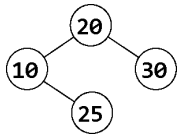
```
class WrapInt {
  public int value;
}
```

或者若用 C++或其他支持按引用传值的语言实现，就可以这么做。

解法 2：最小与最大法

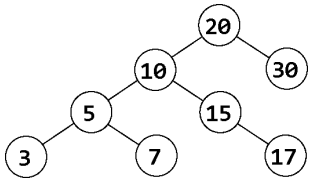
第二种解法利用的是二叉搜索树的定义。成为二叉搜索树的条件是：所有左边的节点必须小于或等于当前节点，而当前节点必须小于所有右边的节点。利用这一点，我们可以通过自上而下传递最小和最大值来解决这个问题。在迭代遍历整个树的过程中，会用逐渐变窄的范围来检查各个节点。

试看下面这棵小树。



尽管每个节点都比左子节点大，比右子节点小，但这显然不是一棵二叉搜索树，其中 25 的位置不对。

以下面这棵树为例。



首先，从(min = NULL, max = NULL)这个范围开始，根节点显然落在其中（NULL 表示没有最小值或最大值）。然后处理左子树，检查这些节点是否落在(min = NULL, max = 20)范围内。接下来处理（值为 10 的节点）右子树，检查节点是否落在(min = 20, max = NULL)的范围内。

之后，继续以此遍历整棵树。进入左子树时，更新 max。进入右子树时，更新 min。只要有任一节点不能通过检查，则停止并返回 false。这种解法的时间复杂度为 $O(N)$，其中 N 为整棵树的节点数为目前最优解。

因为用了递归法，对于平衡树，空间复杂度为 $O(\log N)$。在调用栈上，共有 $O(\log N)$个递归调用，最大递归深度等于树的深度。

该解法的递归实现代码如下：

```
boolean checkBST(TreeNode n) {
  return checkBST(n, null, null);
}

boolean checkBST(TreeNode n, Integer min, Integer max) {
  if (n == null) {
    return true;
  }
  if ((min != null && n.data <= min) || (max != null && n.data > max)) {
    return false;
  }
```

```
12
13    if (!checkBST(n.left, min, n.data) || !checkBST(n.right, n.data, max)) {
14      return false;
15    }
16    return true;
17  }
```

注意在递归算法中，一定要确保基线条件以及节点为空的情况得到妥善处理。

第 5 章

位 操 作

5.1 翻转数位

给定一个整数，你可以将一个数位从 0 变为 1。请编写一个程序，找出你能够获得的最长的一串 1 的长度。

示例：

输入：1775（或者：11011101111）

输出：8

题目解法

假设每个整数数值都由 0 序列和 1 序列交替构成。每当发现一个 0 序列的长度为 1，即有可能将相邻的两个 1 序列合并。

1. 蛮力法

一种解法是将一个整数数值转化为一个数组，该数组中的元素表示其对应的 0 序列和 1 序列的长度。比如，11011101111 将会被转化为（以从右向左的顺序）$[0_0, 4_1, 1_0, 3_1, 1_0, 2_1, 21_1]$，其中的下角标表示长度对应的是 0 序列还是 1 序列，因为该序列是一个严格的从 0 开始的交替序列，所以它们不需要出现在真正的算法中。

有了如上数组之后，我们只需要对其进行遍历。对于每一个 0 序列，如果它的长度为 1，就可以试图合并相邻的两个 1 序列。

```
1  int longestSequence(int n) {
2    if (n == -1) return Integer.BYTES * 8;
3    ArrayList<Integer> sequences = getAlternatingSequences(n);
4    return findLongestSequence(sequences);
5  }
6
7  /* 返回所有序列的尺寸组成的链表。序列由 0 的个数开始（或许为 0），之后是每个值的个数 */
8  ArrayList<Integer> getAlternatingSequences(int n) {
9    ArrayList<Integer> sequences = new ArrayList<Integer>();
10
11   int searchingFor = 0;
12   int counter = 0;
13
14   for (int i = 0; i < Integer.BYTES * 8; i++) {
15     if ((n & 1) != searchingFor) {
16       sequences.add(counter);
17       searchingFor = n & 1; // 将 1 翻转为 0 或者 0 翻转为 1
18       counter = 0;
19     }
20     counter++;
21     n >>>= 1;
22   }
```

```
  sequences.add(counter);

  return sequences;
}

/* 给定由 0 和 1 交替组成的序列的大小，找出我们可以构造的最长的序列 */
int findLongestSequence(ArrayList<Integer> seq) {
  int maxSeq = 1;

  for (int i = 0; i < seq.size(); i += 2) {
    int zerosSeq = seq.get(i);
    int onesSeqRight = i - 1 >= 0 ? seq.get(i - 1) : 0;
    int onesSeqLeft = i + 1 < seq.size() ? seq.get(i + 1) : 0;

    int thisSeq = 0;
    if (zerosSeq == 1) { // 可以合并
      thisSeq = onesSeqLeft + 1 + onesSeqRight;
    } if (zerosSeq > 1) { // 将 0 加入到其中一端
      thisSeq = 1 + Math.max(onesSeqRight, onesSeqLeft);
    } else if (zerosSeq == 0) { // 无 0，因此尝试另一端
      thisSeq = Math.max(onesSeqRight, onesSeqLeft);
    }
    maxSeq = Math.max(thisSeq, maxSeq);
  }

  return maxSeq;
}
```

该解法时间复杂度和空间复杂度均为 $O(b)$，其中 b 为序列的长度。

此案例中因 n 的含义有可能造成歧义时，引起你和面试官的误解。建议选择不同的变量命名。这里我们使用了 b，用于指代比特位的数量同时，正确描述 O 为“该算法的时间复杂度是 O(该整数的比特位数)”。

此算法运行时间已是最优，但可以在减少算法所用的内存空间上进行优化。

2. 优化算法

为了减少内存空间的使用，需要注意我们并不需要从始至终都保留与序列长度相等的空间。必需保留能够比较前后相邻的两个 1 序列的长度即可。

我们可以在遍历的过程中，追踪当前 1 序列的长度和上一段 1 序列的长度。当发现一个比特位为 0 时，更新 previousLength 的值。

- 如果下一个比特位是 1，那么 previousLength 应被置为 currentLength 的值。
- 如果下一个比特位是 0，我们则不能合并这两个 1 序列。因此，需要将 previousLength 的值置为 0。遍历的同时需要更新 maxLength 的值。

```
int flipBit(int a) {
  /* 如果都是 1，那么这已经是最长的序列了 */
  if (~a == 0) return Integer.BYTES * 8;

  int currentLength = 0;
  int previousLength = 0;
  int maxLength = 1; // 我们总能找到包含至少一个 1 的序列
  while (a != 0) {
    if ((a & 1) == 1) { // 当前位为 1
      currentLength++;
    } else if ((a & 1) == 0) { // 当前位为 0
      /* 更新为 0（若下一位是 0）或 currentLength（若下一位是 1）*/
```

```
13        previousLength = (a & 2) == 0 ? 0 : currentLength;
14        currentLength = 0;
15      }
16      maxLength = Math.max(previousLength + currentLength + 1, maxLength);
17      a >>>= 1;
18    }
19    return maxLength;
20  }
```

该算法的时间复杂度保持为为 $O(b)$的同时，只使用了 $O(1)$的额外存储空间。

5.2 调试

解释代码((n & (n-1)) == 0)的具体含义。

题目解法

可以由外而内来解决这个问题。

1. (A & B) == 0 的含义

(A & B) == 0 的含义是，A 和 B 二进制表示的同一位置绝不会同时为 1。因此，如果(n & (n - 1)) == 0，则 n 和 n - 1 就不会有共同的 1。

2. 相比 n，n - 1 长什么样

尝试一下减法（二进制或十进制）。

```
  1101011000 [base 2]                593100 [base 10]
-          1                     -       1
= 1101010111 [base 2]             = 593099 [base 10]
```

当要将一个数减去 1 时，需要注意最低有效位。如果最低有效位为 1，则变为 0，如果是 0，你就必须从高位“借”1。因此，要逐一前往更高的位，将每个位从 0 改为 1，直至找到 1 为止，并将这个 1 翻转成 0。

综上所述，n-1 形似 n，只不过 n 中低位的 0 在 n-1 中变为 1，n 中最低有效位的 1 在 n-1 中变为 0，示例如下。

```
if     n = abcde1000
then n-1 = abcde0111
```

3. 那么，(n & (n-1)) == 0 究竟表示什么

n 和 n-1 不存在同一位均为 1 的情况，因为两者的二进制表示如下：

```
if     n = abcde1000
then n-1 = abcde0111
```

abcde 必定全为 0，也就是说，n 必须形如 00001000，因此，n 的值是 2 的某次方。

问题的答案为：((n & (n-1)) == 0)检查 n 是否为 2 的某次方（或者检查 n 是否为 0）。

5.3 整数转换

编写一个函数，确定需要改变几个位才能将整数 *A* 转成整数 *B*。

示例：

输入：29（或者：11101），15（或者：01111）

输出：2

题目解法

要解决这个问题，得设法找出两个数之间有哪些位不同。使用异或（XOR）操作即可。

在异或操作的结果中，每个 1 代表 *A* 和 *B* 相应位不同。因此，要找出 *A* 和 *B* 有多少个不同的位，只要数一数 *A*^*B* 有几个位为 1。

```
int bitSwapRequired(int a, int b) {
  int count = 0;
  for (int c = a ^ b; c != 0; c = c >>> 1) {
    count += c & 1;
  }
  return count;
}
```

上面的做法是不断对 c 执行移位操作，然后检查最低有效位，但如果不断翻转最低有效位，计算要多少次 c 才会变成 0。操作 `c = c & (c - 1)`会清除 c 的最低有效位。

下面的代码运用了这个方法。

```
int bitSwapRequired(int a, int b) {
  int count = 0;
  for (int c = a ^ b; c != 0; c = c & (c-1)) {
    count++;
  }
  return count;
}
```

这段代码涉及面试中偶尔会出现的位操作问题。记住这个技巧。

5.4　绘制直线

有个单色屏幕存储在一个一维字节数组中，使得 8 个连续像素可以存放在一个字节里。屏幕宽度为 *w*，且 *w* 可被 8 整除（即一个字节不会分布在两行上），屏幕高度可由数组长度及屏幕宽度推算得出。请实现一个函数，绘制从点(*x*1, *y*)到点(*x*2, *y*)的水平线。

该方法的签名应形似于 `drawLine(byte[] screen, int width, int x1, int x2, int y)`。

题目解法

这个问题有个简单解法：用 for 循环迭代，从 *x*1 到 *x*2，一路设定每个像素，但效率不高。

更好的做法是，如果 *x*1 和 *x*2 相距甚远，其间包含几个完整字节，只要使用 `screen[byte_pos] = 0xFF`，一次就能设定一整个字节。这条线起点和终点剩余部分的位，可用掩码设定。

```
void drawLine(byte[] screen, int width, int x1, int x2, int y) {
  int start_offset = x1 % 8;
  int first_full_byte = x1 / 8;
  if (start_offset != 0) {
    first_full_byte++;
  }

  int end_offset = x2 % 8;
  int last_full_byte = x2 / 8;
  if (end_offset != 7) {
    last_full_byte--;
  }

  // 设置完整的字节
  for (int b = first_full_byte; b <= last_full_byte; b++) {
    screen[(width / 8) * y + b] = (byte) 0xFF;
```

```
17    }
18
19    // 创建开始行和结束行的掩码
20    byte start_mask = (byte) (0xFF >> start_offset);
21    byte end_mask = (byte) ~(0xFF >> (end_offset + 1));
22
23    // 设置开始与结束行
24    if ((x1 / 8) == (x2 / 8)) { // x1 和 x2 在同一字节
25      byte mask = (byte) (start_mask & end_mask);
26      screen[(width / 8) * y + (x1 / 8)] |= mask;
27    } else {
28      if (start_offset != 0) {
29        int byte_number = (width / 8) * y + first_full_byte - 1;
30        screen[byte_number] |= start_mask;
31      }
32      if (end_offset != 7) {
33        int byte_number = (width / 8) * y + last_full_byte + 1;
34        screen[byte_number] |= end_mask;
35      }
36    }
37  }
```

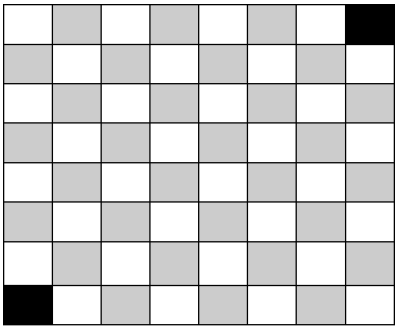
处理这个问题要小心，“陷阱”和特殊情况。例如，你必须考虑到 $x1$ 和 $x2$ 位于同一字节的情况。

第 6 章

数学与逻辑题

6.1 多米诺骨牌

有个 8 × 8 棋盘，其中对角的角落上，两个方格被切掉了。给定 31 块多米诺骨牌，一块骨牌恰好可以覆盖两个方格。用这 31 块骨牌能否盖住整个棋盘？请证明你的答案（提供范例或证明为什么不能）。



题目解法

棋盘大小为 8 × 8，共有 64 个方格，但其中两个方格已被切掉，因此只剩 62 个方格。31 块骨牌应该刚好能盖住整个棋盘，但实际上用骨牌盖住第 1 行，而第 1 行只有 7 个方格，因此有一块骨牌必须铺至第 2 行。而用骨牌盖住第 2 行时，又必须将一块骨牌铺至第 3 行。

要盖住每一行，总有一块骨牌必须铺至下一行。无论怎样都无法成功铺下所有骨牌。

更严谨的证法是。棋盘原本有 32 个黑格和 32 个白格。将对角角落上的两个方格（相同颜色）切掉，棋盘只剩下 30 个同色的方格和 32 个另一种颜色的方格。此时我们假定棋盘上剩下 30 个黑格和 32 个白格。那么，放在棋盘上的每块骨牌必定会盖住一个白格和一个黑格。因此，31 块骨牌正好盖住 31 个白格和 31 个黑格。然而，这个棋盘只有 30 个黑格和 32 个白格，所以，31 块骨牌盖不住整个棋盘。

6.2 三角形上的蚂蚁

三角形的三个顶点上各有一只蚂蚁。如果蚂蚁开始沿着三角形的边爬行，两只或三只蚂蚁撞在一起的概率有多大？假定每只蚂蚁会随机选一个方向，每个方向被选到的概率相等，而且三只蚂蚁的爬行速度相同。

类似问题：在 *n* 个顶点的多边形上有 *n* 只蚂蚁，求出这些蚂蚁发生碰撞的概率。

题目解法

两只蚂蚁互相朝着对方而行，就会发生碰撞。朝着同一方向爬行（顺时针或逆时针），则不会碰撞。

每只蚂蚁可以朝两个方向爬行，一共有三只蚂蚁，它们不发生碰撞的概率如下。

$P(\text{顺时针}) = (1/2)^3$

$P(\text{逆时针}) = (1/2)^3$

$P(\text{同方向}) = (1/2)^3 + (1/2)^3 = 1/4$

因此，发生碰撞的概率就是蚂蚁不朝着同方向爬行的概率。

$P(\text{碰撞}) = 1 - P(\text{同方向}) = 1 - 1/4 = 3/4$

若要将这个方法推广至 n 个顶点的多边形，不相撞的情况一致，但总共有 $2n$ 种爬行方式。综上所述，发生碰撞的概率如下。

$P(\text{顺时针}) = (1/2)^n$

$P(\text{逆时针}) = (1/2)^n$

$P(\text{同方向}) = 2(1/2)^n = (1/2)^{n-1}$

$P(\text{碰撞}) = 1 - P(\text{同方向}) = 1 - (1/2)^{n-1}$

6.3 水壶问题

有两个水壶，容量分别为 3 升和 5 升，若水的供应不限量（但没有量杯），怎么用这两个水壶得到刚好的水？注意，这两个水壶呈不规则状，无法精准地装满“半壶”水。

题目解法

根据题意，我们只能使用这两个水壶，不妨随意把玩一番，把水倒来倒去，可以得到如下顺序组合。

5 升水壶	3 升水壶	操　作
5	0	装满5升水壶
2	3	用5升水壶里的水装满3升水壶
2	0	将3升水壶里的水倒掉
0	2	将5升水壶里的水倒入3升水壶
5	2	装满5升水壶
4	3	用5升水壶里的水装满3升水壶
4		搞定！准确量得4升

只要这两个水壶的容量互质，我们就能找出一种倒水的顺序组合，量出一到两个水壶容量总和（含）之间的任意水量。

6.4 蓝眸岛

有个岛上住着一群人，有一天来了个游客，定了一条奇怪的规矩：所有棕色眼睛的人都必须尽快离开这个岛。每晚 8 点会有一个航班离岛。每个人都看得见别人眼睛的颜色，但不知道自己的（别人也不可以告知）。此外，他们不知道岛上到底有多少人有棕色眼睛，只知道至少有一个人的眼睛是棕色的。所有棕色眼睛的人要花几天才能离开这个岛？

题目解法

下面将采用简单构造法。假定这个岛上一共有 n 人，其中 c 人有棕色眼睛。由题目可知，$c > 0$。

1. 情况 $c = 1$：只有一人眼睛是棕色的

假设岛上所有人都智力超群，棕色眼睛的人四处观察之后，发现没有人的眼睛是棕色的。但他知道至少有一人眼睛是棕色的，于是就推导出自己的眼睛一定是棕色的。因此，他会搭乘当晚的飞机离开。

2. 情况 $c = 2$：只有两人眼睛是棕色的

两个棕色眼睛的人看到对方，并不确定 c 是 1 还是 2，但是由上一种情况得知，如果 $c = 1$，那个棕色眼睛的人第一晚就会离岛。因此，发现另一个棕色眼睛的人仍在岛上，他一定能推断出 $c = 2$，也就意味着他自己的眼睛也是棕色的。于是，两个棕色眼睛的人都会在第二晚离岛。

3. 情况 $c > 2$：一般情况

逐步增加 c 时，我们可以看出上述公式仍旧适用。如果 $c = 3$，那么，这三个人会立即意识到有两到三人的眼睛是棕色的。如果有两人眼睛是棕色的，那么这两人会在第二晚离岛。因此，如果过了第二晚另外两人还在岛上，每个棕色眼睛的人都能推断出 $c = 3$，因此这三人都有棕色眼睛。他们会在第三晚离岛。

无论 c 为何值，都可套用这个公式。所以，如果有 c 人有棕色眼睛，则所有棕色眼睛的人要用 c 晚才能离岛，且都在同一晚离开。

6.5　100 个储物柜

走廊上有 100 个关上的储物柜。有个人先是将 100 个柜子全都打开。接着，每数两个柜子关上一个。然后，在第三轮时，再每隔两个就切换第三个柜子的开关状态（也就是将关上的柜子打开，将打开的关上）。照此规律反复操作 100 次，在第 i 轮，这个人会每数 i 个就切换第 i 个柜子的状态。当第 100 轮经过走廊时，只切换第 100 个柜子的开关状态，此时有几个柜子是开着的?

题目解法

首先，必须弄清楚切换储物柜开关状态是什么意思。这有助于后续推断最终哪些柜子是开着的。

1. 问题：柜子会在哪几轮切换状态（开或关）

柜子 n 会在 n 的每个因子（包括 1 和 n）对应的那一轮切换状态，也就是说，柜子 15 会在第 1、3、5 和 15 轮开或关一次。

2. 问题：柜子什么时候还是开着的

如果因子个数（记作 x）为奇数，则这个柜子是开着的。可以把一对因子比作开和关，若还剩一个因子，则柜子就是开着的。

3. 问题：x 什么时候为奇数

若 n 为完全平方数，则 x 的值为奇数。理由如下：将 n 的两个互补因子配对。例如，如 n 为 36，则因子配对情况为：(1, 36)、(2, 18)、(3, 12)、(4, 9)、(6, 6)。注意，(6, 6)其实只有一个因子，因此 n 的因子个数为奇数。

4. 问题：有多少个完全平方数

一共有 10 个完全平方数，你可以数一数（1, 4, 9, 16, 25, 36, 49, 64, 81, 100），或者直接列出 1 到 10 的平方。

$$1 \times 1, 2 \times 2, 3 \times 3, \cdots, 10 \times 10$$

因此，最后共有 10 个柜子是开着的。

第 7 章

面向对象设计

7.1 电台点播系统

运用面向对象原则，设计一款音乐点唱机。

题目解法

但凡遇到面向对象设计的问题，一开始就要向面试官问几个问题，以清晰设计时有哪些限制条件。这电台点播系统放的是 CD，是黑胶，还是 MP3？它是计算机模拟软件，还是代表一台实体设备？

假设这电台点播系统为计算机模拟软件，类似于实体设备。

下面将列出基本的系统组件：

- 点唱系统（jukebox）；
- CD；
- 歌曲（song）；
- 艺术家（artist）；
- 播放列表（playlist）；
- 显示屏（display，在屏幕上显示详细信息）。

接下来，进一步分解上述组件，考虑可能的动作：

- 新建播放列表（包括新增、删除和随机播放）；
- CD 选择器；
- 歌曲选择器；
- 将歌曲放进播放队列；
- 获取播放列表中的下一首歌曲。

另外，还可引入用户：

- 添加；
- 删除；
- 信用信息。

每个主要系统组件大致都会转换成一个对象，每个动作则转换为一个方法。下面将介绍一种可行的设计。

`Jukebox` 类代表此题的主体，系统各个组件之间或系统与用户间的大量交互都是通过这个类实现的。

```
1  public class Jukebox {
2    private CDPlayer cdPlayer;
3    private User user;
```

```
    private Set<CD> cdCollection;
    private SongSelector ts;

    public Jukebox(CDPlayer cdPlayer, User user, Set<CD> cdCollection,
                   SongSelector ts) { ... }

    public Song getCurrentSong() { return ts.getCurrentSong(); }
    public void setUser(User u) { this.user = u; }
}
```

跟实体设备一样，CDPlayer 类一次只能放一张 CD。不再播放的 CD 都存放在电台点播系统里。

```
public class CDPlayer {
  private Playlist p;
  private CD c;

  /* 构造函数 */
  public CDPlayer(CD c, Playlist p) { ... }
  public CDPlayer(Playlist p) { this.p = p; }
  public CDPlayer(CD c) { this.c = c; }

  /* 播放歌曲 */
  public void playSong(Song s) { ... }

  /* getter 和 setter */
  public Playlist getPlaylist() { return p; }
  public void setPlaylist(Playlist p) { this.p = p; }

  public CD getCD() { return c; }
  public void setCD(CD c) { this.c = c; }
}
```

Playlist 类管理当前播放的歌曲和待播放的下一首歌曲。它本质上是播放队列的包裹类，还提供了一些操作起来更方便的方法。

```
public class Playlist {
  private Song song;
  private Queue<Song> queue;
  public Playlist(Song song, Queue<Song> queue) {
    ...
  }
  public Song getNextSToPlay() {
    return queue.peek();
  }
  public void queueUpSong(Song s) {
    queue.add(s);
  }
}
```

CD、Song 和 User 这几个类都相当简单，主要由成员变量、getter（访问）和 setter（设置）方法组成。

```
public class CD { /* 识别码、艺术家、歌曲等数据 */ }

public class Song { /* 识别码、CD（可能为空）、曲名、长度等数据 */ }

public class User {
  private String name;
  public String getName() { return name; }
  public void setName(String name) {  this.name = name; }
  public long getID() { return ID; }
```

```
  public void setID(long iD) { ID = iD; }
  private long ID;
  public User(String name, long iD) { ... }
  public User getUser() { return this; }
  public static User addUser(String name, long iD) { ... }
}
```

7.2　停车场

运用面向对象原则，设计一个停车场。

题目解法

首先问清楚允许哪些车辆进入停车场，停车场是不是多层的，等细节。假设

- 停车场是多层的。每一层有好几排停车位。
- 停车场可停放摩托车、轿车和大巴。
- 停车场有摩托车车位、小车位和大车位。
- 摩托车可停在任意车位上。
- 轿车可停在单个小车位或大车位上。
- 大巴可停在同一排五个连续的大车位上，但不能停在小车位上。

在下面的实现中，我们创建了抽象类 `Vehicle`，`Car`、`Bus` 和 `Motorcycle` 都继承自这个类。为处理不同大小的车位，我们用了一个 `ParkingSpot` 类，并以它的成员变量表示车位大小。

```
public enum VehicleSize { Motorcycle, Compact, Large }

public abstract class Vehicle {
  protected ArrayList<ParkingSpot> parkingSpots = new ArrayList<ParkingSpot>();
  protected String licensePlate;
  protected int spotsNeeded;
  protected VehicleSize size;

  public int getSpotsNeeded() { return spotsNeeded; }
  public VehicleSize getSize() { return size; }

  /* 将车辆停在这个车位里（也可能包含其他车位）*/
  public void parkInSpot(ParkingSpot s) { parkingSpots.add(s); }

  /* 从车位移除车辆，并通知车位车辆已离开 */
  public void clearSpots() { ... }

  /* 检查车位是否够大以停放该车辆（且车位是空的），
   * 这只会检查车位大小，并不检查是否有足够多的车位 */
  public abstract boolean canFitInSpot(ParkingSpot spot);
}

public class Bus extends Vehicle {
  public Bus() {
    spotsNeeded = 5;
    size = VehicleSize.Large;
  }

  /* 检查车位是否为大车位，不会检查车位的数目 */
  public boolean canFitInSpot(ParkingSpot spot) { ... }
}

public class Car extends Vehicle {
```

```
  public Car() {
    spotsNeeded = 1;
    size = VehicleSize.Compact;
  }

  /* 检查车位是小车位还是大车位 */
  public boolean canFitInSpot(ParkingSpot spot) { ... }
}

public class Motorcycle extends Vehicle {
  public Motorcycle() {
    spotsNeeded = 1;
    size = VehicleSize.Motorcycle;
  }

  public boolean canFitInSpot(ParkingSpot spot) { ... }
}
```

ParkingLot 类本质上就是 Level 数组的包裹类。以这种方式实现，我们就能将真正寻找空车位和泊车的处理逻辑从 ParkingLot 里更为广泛的动作中抽取出来。否则就需要将车位放在某种双数组中或将车位位于所在楼层的编号对应到车位列表的散列表。将 ParkingLot 与 Level 分离开来，整个设计更显清晰。

```
public class ParkingLot {
  private Level[] levels;
  private final int NUM_LEVELS = 5;

  public ParkingLot() { ... }

  /* 将该车辆停在一个车位或多个车位，失败则返回 false */
  public boolean parkVehicle(Vehicle vehicle) { ... }
}

/* 代表停车场里的一层 */
public class Level {
  private int floor;
  private ParkingSpot[] spots;
  private int availableSpots = 0; // 空闲车位的数量
  private static final int SPOTS_PER_ROW = 10;

  public Level(int flr, int numberSpots) { ... }

  public int availableSpots() { return availableSpots; }

  /* 找地方停这辆车，失败则返回 false */
  public boolean parkVehicle(Vehicle vehicle) { ... }

  /* 停放该车辆，从车位编号 spotNumber 开始，直到 vehicle.spotsNeeded */
  private boolean parkStartingAtSpot(int num, Vehicle v) { ... }

  /* 寻找车位停放这辆车。返回车位索引号，失败则返回-1 */
  private int findAvailableSpots(Vehicle vehicle) { ... }

  /* 当有车辆从车位移除时，增加可用车位数 availableSpots */
  public void spotFreed() { availableSpots++; }
}
```

ParkingSpot 类只用一个变量表示车位的大小。

```
public class ParkingSpot {
```

```
    private Vehicle vehicle;
    private VehicleSize spotSize;
    private int row;
    private int spotNumber;
    private Level level;

    public ParkingSpot(Level lvl, int r, int n, VehicleSize s) {...}

    public boolean isAvailable() { return vehicle == null; }

    /* 检查车位是否够大、可用 */
    public boolean canFitVehicle(Vehicle vehicle) { ... }

    /* 将车辆停在该车位 */
    public boolean park(Vehicle v) { ... }

    public int getRow() { return row; }
    public int getSpotNumber() { return spotNumber; }

    /* 从车位移除车辆，并通知楼层，有新的车位可用 */
    public void removeVehicle() { ... }
}
```

电子版可以找到上述代码的完整实现，包括可执行的测试代码。

7.3　线上图书馆

请设计线上图书馆系统的数据结构。

题目解法

假设系统功能要求。

- ❑ 用户成员资格的建立和延长期限。
- ❑ 搜索图书数据库。
- ❑ 阅读书籍。
- ❑ 同一时间只能有一个活跃用户。
- ❑ 该用户一次只能看一本书。

以上操作需提供许多其他函数，比如 `get`、`set`、`update` 等。系统的对象可能包括 `User`、`Book` 和 `Library`。

`OnlineReaderSystem` 类为程序的主体，存放所有图书的信息，管理用户，刷新显示画面，整个类会变得臃肿不堪。因此，我们将这些组件拆分成 `Library`、`UserManager` 和 `Display` 等几个类。

```
public class OnlineReaderSystem {
  private Library library;
  private UserManager userManager;
  private Display display;

  private Book activeBook;
  private User activeUser;

  public OnlineReaderSystem() {
    userManager = new UserManager();
    library = new Library();
    display = new Display();
```

```
  }

  public Library getLibrary() { return library; }
  public UserManager getUserManager() { return userManager; }
  public Display getDisplay() { return display; }

  public Book getActiveBook() { return activeBook; }
  public void setActiveBook(Book book) {
    activeBook = book;
    display.displayBook(book);
  }

  public User getActiveUser() { return activeUser; }
  public void setActiveUser(User user) {
    activeUser = user;
    display.displayUser(user);
  }
}
```

然后，我们实现这几个类，用于处理用户管理器、图书库和显示组件。

```
public class Library {
  private HashMap<Integer, Book> books;

  public Book addBook(int id, String details) {
    if (books.containsKey(id)) {
      return null;
    }
    Book book = new Book(id, details);
    books.put(id, book);
    return book;
  }

  public boolean remove(Book b) { return remove(b.getID()); }
  public boolean remove(int id) {
    if (!books.containsKey(id)) {
      return false;
    }
    books.remove(id);
    return true;
  }

  public Book find(int id) {
    return books.get(id);
  }
}

public class UserManager {
  private HashMap<Integer, User> users;

  public User addUser(int id, String details, int accountType) {
    if (users.containsKey(id)) {
      return null;
    }
    User user = new User(id, details, accountType);
    users.put(id, user);
    return user;
  }

  public User find(int id) { return users.get(id); }
  public boolean remove(User u) { return remove(u.getID()); }
  public boolean remove(int id) {
```

```
    if (!users.containsKey(id)) {
      return false;
    }
    users.remove(id);
    return true;
  }
}

public class Display {
  private Book activeBook;
  private User activeUser;
  private int pageNumber = 0;

  public void displayUser(User user) {
    activeUser = user;
    refreshUsername();
  }

  public void displayBook(Book book) {
    pageNumber = 0;
    activeBook = book;

    refreshTitle();
    refreshDetails();
    refreshPage();
  }

  public void turnPageForward() {
    pageNumber++;
    refreshPage();
  }

  public void turnPageBackward() {
    pageNumber--;
    refreshPage();
  }

  public void refreshUsername() { /* 更新显示的用户名 */ }
  public void refreshTitle() { /* 更新显示的书名 */ }
  public void refreshDetails() { /* 更新显示的详细信息 */ }
  public void refreshPage() { /* 更新显示的页数 */ }
}
```

User 和 Book 类只是存放数据，并没有什么真正的功能。

```
public class Book {
  private int bookId;
  private String details;

  public Book(int id, String det) {
    bookId = id;
    details = det;
  }

  public int getID() { return bookId; }
  public void setID(int id) { bookId = id; }
  public String getDetails() { return details; }
  public void setDetails(String d) { details = d; }
}

public class User {
  private int userId;
```

```
18    private String details;
19    private int accountType;
20
21    public void renewMembership() { }
22
23    public User(int id, String details, int accountType) {
24      userId = id;
25      this.details = details;
26      this.accountType = accountType;
27    }
28
29    /* Getter和setter方法 */
30    public int getID() { return userId; }
31    public void setID(int id) { userId = id; }
32    public String getDetails() {
33      return details;
34    }
35
36    public void setDetails(String details) {
37      this.details = details;
38    }
39    public int getAccountType() { return accountType; }
40    public void setAccountType(int t) { accountType = t; }
41  }
```

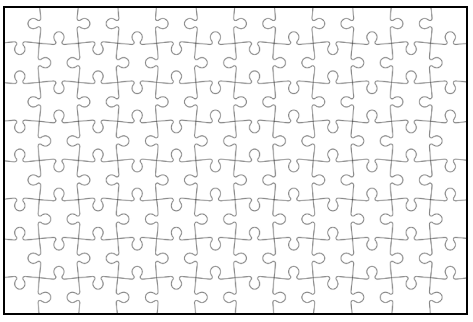
将信息拆分至不同的类中，可以避免这个主类变得臃肿不堪。但不适用于小系统，不扩展的小系统统放在一起即可。

7.4 拼图游戏

实现一个 $N \times N$ 的拼图程序。设计相关数据结构并提供一种拼图算法。假设你有一种 `fitsWith` 方法，传入两块拼图，若两块拼图能拼在一起，则返回 `true`。

题目解法

假设有一套传统的拼图游戏，按行和列划分为网格，每块拼图都落在某一行和某一列中，有 4 条边，每条边分为 3 种：内凹、外凸和平直。例如，角落的拼图块有两条边是平直的，另外两条边可能是内凹或外凸。



在玩拼图游戏（手动或借助算法）时，我们需要存储每块拼图的位置，位置可以是绝对的或相对的。

- 绝对位置："这块拼图的位置是(12, 23)。"
- 相对位置："我不知道这块拼图的实际位置，但知道它与另一块拼图相邻。"

我们的解法只使用绝对位置。

我们需要一些表示 Puzzle、Piece 和 Edge 的类。另外，需要表示不同形状（inner、outer、flat）的 4 条边（left、top、right、bottom）的枚举类型。

开始时，Puzzle 类应包含一个链表，链表的元素为 Piece。当拼图游戏结束时，会得到一个由 Piece 组成的 $N \times N$ 的矩阵。

Piece 类应包含一个散列表，该散列表以边的方向为键，边为值。请注意，有时我们需要旋转一块拼图，这种情况下散列表的值也会发生变化。边的方向在开始时会被赋予一个任意值。

Edge 类只包含形状和其所属拼图的指针，其本身不保存边的方向。

下面是一种可能的面向对象设计。

```
public enum Orientation {
  LEFT, TOP, RIGHT, BOTTOM; // 保持有序

  public Orientation getOpposite() {
    switch (this) {
      case LEFT: return RIGHT;
      case RIGHT: return LEFT;
      case TOP: return BOTTOM;
      case BOTTOM: return TOP;
      default: return null;
    }
  }
}

public enum Shape {
  INNER, OUTER, FLAT;

  public Shape getOpposite() {
    switch (this) {
      case INNER: return OUTER;
      case OUTER: return INNER;
      default: return null;
    }
  }
}

public class Puzzle {
  private LinkedList<Piece> pieces; /* 剩余拼图 */
  private Piece[][] solution;
  private int size;

  public Puzzle(int size, LinkedList<Piece> pieces) { ... }


  /* 将拼图放入解决方案中，进行恰当的旋转并从链表中移除 */
  private void setEdgeInSolution(LinkedList<Piece> pieces, Edge edge, int row,
                                 int column, Orientation orientation) {
    Piece piece = edge.getParentPiece();
    piece.setEdgeAsOrientation(edge, orientation);
    pieces.remove(piece);
    solution[row][column] = piece;
  }

  /* 在 piecesToSearch 中找到匹配的拼图并插入到当前列和行的位置 */
  private boolean fitNextEdge(LinkedList<Piece> piecesToSearch, int row, int col);

  /* 解决拼图问题 */
  public boolean solve() { ... }
```

```
}

public class Piece {
  private HashMap<Orientation, Edge> edges = new HashMap<Orientation, Edge>();

  public Piece(Edge[] edgeList) { ... }

  /* 按照 numberRotations 旋转拼图的边 */
  public void rotateEdgesBy(int numberRotations) { ... }

  public boolean isCorner() { ... }
  public boolean isBorder() { ... }
}

public class Edge {
  private Shape shape;
  private Piece parentPiece;
  public Edge(Shape shape) { ... }
  public boolean fitsWith(Edge edge) { ... }
}
```

拼拼图的算法

首先将所有拼图分成 4 个角落的拼图、4 边的拼图和内部的拼图。

分类结束之后，任意选择一块角落的拼图将其置于左上角。然后，我们按顺序遍历所有的拼图，一块接一块地将拼图摆放在合适的位置上。在每一个拼图所处的位置，我们在对应的拼图类别中搜索合适的拼图。当将一块拼图放入图中后，将其旋转至合适的方向。

下面的代码勾勒出了该算法。

```
/* 在 piecesToSearch 中找到匹配的拼图并插入到当前列和行的位置 */
boolean fitNextEdge(LinkedList<Piece> piecesToSearch, int row, int column) {
  if (row == 0 && column == 0) { // 在左上角直接放置一块拼图
    Piece p = piecesToSearch.remove();
    orientTopLeftCorner(p);
    solution[0][0] = p;
  } else {
    /* 获取右侧以及匹配的链表 */
    Piece pieceToMatch = column == 0 ? solution[row - 1][0] :
                                       solution[row][column - 1];
    Orientation orientationToMatch = column == 0 ? Orientation.BOTTOM :
                                                   Orientation.RIGHT;
    Edge edgeToMatch = pieceToMatch.getEdgeWithOrientation(orientationToMatch);

    /* 获取匹配的边 */
    Edge edge = getMatchingEdge(edgeToMatch, piecesToSearch);
    if (edge == null) return false; // 无法解决

    /* 插入边和拼图 */
    Orientation orientation = orientationToMatch.getOpposite();
    setEdgeInSolution(piecesToSearch, edge, row, column, orientation);
  }
  return true;
}

boolean solve() {
  /* 将拼图分组 */
  LinkedList<Piece> cornerPieces = new LinkedList<Piece>();
  LinkedList<Piece> borderPieces = new LinkedList<Piece>();
  LinkedList<Piece> insidePieces = new LinkedList<Piece>();
  groupPieces(cornerPieces, borderPieces, insidePieces);
```

```
32
33    /* 遍历所有拼图，找到和前一个拼图匹配的拼图 */
34    solution = new Piece[size][size];
35    for (int row = 0; row < size; row++) {
36      for (int column = 0; column < size; column++) {
37        LinkedList<Piece> piecesToSearch = getPieceListToSearch(cornerPieces,
38          borderPieces, insidePieces, row, column);
39        if (!fitNextEdge(piecesToSearch, row, column)) {
40          return false;
41        }
42      }
43    }
44
45    return true;
46  }
```

电子版代码附件中查看。

7.5　黑白棋

“黑白棋的玩法如下：每一枚棋子的一面为白，一面为黑。游戏双方各执黑、白棋子对决，当一枚棋子的左右或上下同时被对方棋子夹住，这枚棋子就算是被吃掉了，随即翻面为对方棋子的颜色。轮到你落子时，必须至少吃掉对方一枚棋子。任意一方无子可落时，游戏即告结束。最后，棋盘上棋子较多的一方获胜。请运用面向对象设计方法，实现“奥赛罗棋”。

题目解法

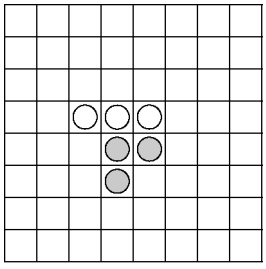
我们先来举个例子。假设在一盘奥赛罗棋中，有如下棋步。

(1) 初始化棋盘，在中心位置布下两枚黑子和两枚白子。两枚黑子分别落在中心点的左上方和右下方。

(2) 在 6 行 4 列处落黑子，则 5 行 4 列的白子翻面变为黑子。

(3) 在 4 行 3 列处落白子，则 4 行 4 列的黑子翻面变为白子。

经过上面的棋步，棋盘布局如下。



在黑白棋中，核心对象大致有游戏（game）、棋盘（board）、棋子（piece，黑子或白子）和玩家（player）。该如何用面向对象设计优雅地表示这些对象？

1. 该不该创建 `BlackPiece` 和 `WhitePiece` 类

起先，我们如果从 `Piece` 抽象类派生出 `BlackPiece` 类和 `WhitePiece` 类。但因为，每颗棋子都可以来回翻面，黑变白，白变黑，所以，连续不断地销毁和创建完全相同的对象并不明智。因此，更好的做法是只创建 `Piece` 类，并用标记指示棋子当前的颜色。

2. 需要 Board 和 Game 两个独立的类吗

虽然没有必要既创建 Game 对象又引入 Board 对象。但是分别创建这两个对象可以从逻辑上划分棋盘（只含涉及落子的逻辑处理）和游戏（含计时、游戏流程等）。不过缺点是，程序在多层处理时，个别函数会因为要调用 Board 里的方法，而先调用 Game 的方法让处理复杂化。下面我们会将 Game 和 Board 分开创建，不过面试时最好跟面试官讨论一下。

3. 谁来记录分数

我们需要某种记分方式来记录黑子和白子的数目。可是不管是由 Game 还是 Board 又或者 Piece（在静态方法中）维护这些信息，各有各的理由。我们暂定由 Board 保存这部分信息，分数在逻辑上可以算是棋盘的一部分，由 Piece 或 Board 调用 Board 类的 colorChanged 和 colorAdded 方法进行更新。

4. Game 该不该实现成单态类

将 Game 实现为单态类，优点在于 Game 的方法调用起来很容易，不用将 Game 对象的引用传来传去。

不过，将 Game 实现成单态类也意味着它只能实例化一次，这个假设条件是否成立需要在面试时，与面试官交流一下。

下面是黑白棋的一种可能设计。

```
public enum Direction {
  left, right, up, down
}

public enum Color {
  White, Black
}

public class Game {
  private Player[] players;
  private static Game instance;
  private Board board;
  private final int ROWS = 10;
  private final int COLUMNS = 10;

  private Game() {
    board = new Board(ROWS, COLUMNS);
    players = new Player[2];
    players[0] = new Player(Color.Black);
    players[1] = new Player(Color.White);
  }

  public static Game getInstance() {
    if (instance == null) instance = new Game();
    return instance;
  }

  public Board getBoard() {
    return board;
  }
}
```

Board 类负责管理棋子本身，但并不处理游戏玩法的部分，而是交由 Game 类处理。

```
public class Board {
  private int blackCount = 0;
  private int whiteCount = 0;
```

```
  private Piece[][] board;

  public Board(int rows, int columns) {
    board = new Piece[rows][columns];
  }

  public void initialize() {
    /* 初始化棋盘中心的白子和黑子 */
  }

  /* 试着将颜色为 color 的棋子放在(row, column)位置，成功则返回 true */
  public boolean placeColor(int row, int column, Color color) {
    ...
  }

  /* 从(row, column)开始，顺着方向 d，将棋子翻面 */
  private int flipSection(int row, int column, Color color, Direction d) { ... }

  public int getScoreForColor(Color c) {
    if (c == Color.Black) return blackCount;
    else return whiteCount;
  }

  /* 更新棋盘，有 newPieces 个棋子变为 newColor 颜色，减少另一种颜色的分数*/
  public void updateScore(Color newColor, int newPieces) { ... }
}
```

如前所述，我们会用 Piece 类实现黑白棋子，该类有个简单的 Color 变量，表示棋子是黑子还是白子。

```
public class Piece {
  private Color color;
  public Piece(Color c) { color = c; }

  public void flip() {
    if (color == Color.Black) color = Color.White;
    else color = Color.Black;
  }

  public Color getColor() { return color; }
}
```

Player 存放的信息非常有限，甚至不会保存自己的分数，但可以通过 Player.getScore() 会调用 GameManager 的方法取得分数。

```
public class Player {
  private Color color;
  public Player(Color c) { color = c; }

  public int getScore() { ... }

  public boolean playPiece(int r, int c) {
    return Game.getInstance().getBoard().placeColor(r, c, color);
  }

  public Color getColor() { return color; }
}
```

在处理很多问题时，相比你做了些什么，你**为什么**这么做反而更显重要。面试官也许不会在意你是否选择将 Game 类实现为单态类，但可能真的在乎你有没有花时间思考，有没有跟她讨论各种做法的优劣。

7.6 文件系统

设计一种内存文件系统（in-memory file system）的数据结构和算法，并说明其具体做法。如若可行，请用代码举例说明。

题目解法

本题主要考查组件考虑是否周全，注意不要因为简单就轻敌。

一个最简单的文件系统由 File（文件）和 Directory（目录）组成。每个 Directory 包含一组 File 和 Directory。File 和 Directory 特征相似，因此我们创建了 Entry 类，前面两个类则继承自这个类。

```
public abstract class Entry {
  protected Directory parent;
  protected long created;
  protected long lastUpdated;
  protected long lastAccessed;
  protected String name;

  public Entry(String n, Directory p) {
    name = n;
    parent = p;
    created = System.currentTimeMillis();
    lastUpdated = System.currentTimeMillis();
    lastAccessed = System.currentTimeMillis();
  }

  public boolean delete() {
    if (parent == null) return false;
    return parent.deleteEntry(this);
  }

  public abstract int size();

  public String getFullPath() {
    if (parent == null) return name;
    else return parent.getFullPath() + "/" + name;
  }

  /* getter 和 setter */
  public long getCreationTime() { return created; }
  public long getLastUpdatedTime() { return lastUpdated; }
  public long getLastAccessedTime() { return lastAccessed; }
  public void changeName(String n) { name = n; }
  public String getName() { return name; }
}

public class File extends Entry {
  private String content;
  private int size;

  public File(String n, Directory p, int sz) {
    super(n, p);
    size = sz;
  }

  public int size() { return size; }
  public String getContents() { return content; }
  public void setContents(String c) { content = c; }
```

```
48 }
49
50 public class Directory extends Entry {
51   protected ArrayList<Entry> contents;
52
53   public Directory(String n, Directory p) {
54     super(n, p);
55     contents = new ArrayList<Entry>();
56   }
57
58   public int size() {
59     int size = 0;
60     for (Entry e : contents) {
61       size += e.size();
62     }
63     return size;
64   }
65
66   public int numberOfFiles() {
67     int count = 0;
68     for (Entry e : contents) {
69       if (e instanceof Directory) {
70         count++; // 目录也算作文件
71         Directory d = (Directory) e;
72         count += d.numberOfFiles();
73       } else if (e instanceof File) {
74         count++;
75       }
76     }
77     return count;
78   }
79
80   public boolean deleteEntry(Entry entry) {
81     return contents.remove(entry);
82   }
83
84   public void addEntry(Entry entry) {
85     contents.add(entry);
86   }
87
88   protected ArrayList<Entry> getContents() { return contents; }
89 }
```

另外，我们还可以这样实现 Directory：为文件和子目录创建不同的链表。如此一来，numberOfFiles()方法就不需要再用 instanceof 运算符了，更为简洁，缺点是我们就无法轻易按日期或名称对文件和目录进行排序了。

第 8 章

递归与动态规划

8.1 迷路的机器人

设想有个机器人坐在一个网格的左上角，网格 *r* 行 *c* 列。机器人只能向下或向右移动，但不能走到一些被禁止的网格。设计一种算法，寻找机器人从左上角移动到右下角的路径。

题目解法

如果把网格画出来，你会发现移动到位置(*r*, *c*)的唯一方式，就是先移动到它的相邻点，即(*r*–1, *c*)或(*r*, *c*–1)。因此，我们需要找到一条移至(*r*–1, *c*)或(*r*, *c*–1)的路径。

怎么才能找出前往这些位置的路径呢？要找出前往(*r*–1, *c*)或(*r*, *c*–1)的路径，我们需要先移至其中一个相邻点。因此，要找到一条路径移动到(*r*–1, *c*)的相邻点，坐标为(*r*–2, *c*)和(*r*–1, *c*–1)或(*r*, *c*–1)的相邻点，其坐标为(*r*–1, *c*–1)和(*r*, *c*–2)。注意，坐标(*r*–1, *c*–1)一共出现了两次，稍候再作讨论。

> 小技巧：很多人处理二维数组时喜欢用 *x* 和 *y* 当作下标值。但有时 bug 就是因此而来。人们通常认为 *x* 是矩阵中的第一维，*y* 是第二维（比如 `matrix[x][y]`）。但事实。第一维通常作为列，也就是 *y* 的值（它是垂直的）。你应该写成 `matrix[y][x]`。或者，直接使用 `r`（row）和 `c`（column）代替。

因此，要找到一条从原点出发的路径，我们只需像上面那样从终点往回走。从最后一点开始，试着找出一条到其相邻点的路径。下面是该算法的递归实现代码。

```
1  ArrayList<Point> getPath(boolean[][] maze) {
2    if (maze == null || maze.length == 0) return null;
3    ArrayList<Point> path = new ArrayList<Point>();
4    if (getPath(maze, maze.length - 1, maze[0].length - 1, path)) {
5      return path;
6    }
7    return null;
8  }
9
10 boolean getPath(boolean[][] maze, int row, int col, ArrayList<Point> path) {
11   /* 如果越界或无效，则直接返回 */
12   if (col < 0 || row < 0 || !maze[row][col]) {
13     return false;
14   }
15
16   boolean isAtOrigin = (row == 0) && (col == 0);
17
18     /* 如果有一条路径从起点通向这里，把它添加到我的位置 */
19     if (isAtOrigin || getPath(maze, row, col - 1, path) ||
20         getPath(maze, row - 1, col, path)) {
```

```
      Point p = new Point(row, col);
      path.add(p);
      return true;
    }

    return false;
  }
```

这个解法的时间复杂度是 $O(2^{r+c})$，因为每个路径都有 $r+c$ 步，每步都有两种选择。

可以通过优化指数级算法去掉重复性工作来实现更高效率。上面算法都有哪些重复工作？

完整过一遍算法就会发现，我们多次访问方格。事实上，每一个方格都被访问了一遍又一遍。毕竟格子才 rc 个，我们的算法却要访问 $O(2^{r+c})$次。假如能做到每个格子都只访问一次，算法的时间复杂度可能接近 $O(rc)$。

那么目前的算法是如何工作的？为了找到一条到(r, c)的路径，算法先去找到通往相邻点的路径，即$(r–1, c)$或$(r, c–1)$。在这个过程中，会忽略禁止访问的点。接下来寻找这两个点的相邻节点， 即$(r–2, c)$、$(r–1, c–1)$、$(r–1, c–1)$和$(r, c–2)$，其中$(r–1, c–1)$出现了两次，也就是我们想要寻找的重复性工作。理想情况下，我们应该能记住访问过$(r–1, c–1)$节点以节省时间。

下面的动态规划算法正是这样做的。

```
ArrayList<Point> getPath(boolean[][] maze) {
  if (maze == null || maze.length == 0) return null;
  ArrayList<Point> path = new ArrayList<Point>();
  HashSet<Point> failedPoints = new HashSet<Point>();
  if (getPath(maze, maze.length - 1, maze[0].length - 1, path, failedPoints)) {
    return path;
  }
  return null;
}

boolean getPath(boolean[][] maze, int row, int col, ArrayList<Point> path,
                HashSet<Point> failedPoints) {
  /* 如果越界或无效，则直接返回 */
  if (col < 0 || row < 0 || !maze[row][col]) {
    return false;
  }

  Point p = new Point(row, col);

  /* 如果已经访问过该点，则返回 */
  if (failedPoints.contains(p)) {
    return false;
  }

  boolean isAtOrigin = (row == 0) && (col == 0);

  /* 如果找到一条路径从起点通往当前位置，把它放到结果里 */
  if (isAtOrigin || getPath(maze, row, col - 1, path, failedPoints) ||
      getPath(maze, row - 1, col, path, failedPoints)) {
    path.add(p);
    return true;
}

  failedPoints.add(p); // 缓存结果
  return false;
}
```

改变虽小，却大大提升了算法执行速度。现在这个算法运行时间是 $O(XY)$，因为每个格子仅访问一次。

8.2 巧合索引

在数组 A[0...n-1]中，存在所谓的巧合索引，满足条件 A[i] = i。给定一个有序整数数组，已知元素值各不相同，请你编写一种方法找出巧合索引，如果数组 A 中存在这类索引的话。

进阶：如果数组元素有重复值，又该如何处理呢?

题目解法

看到这个问题，第一个想到的应该是蛮力法，蛮力法只需迭代访问整个数组，找出符合条件的元素即可。

```
int magicSlow(int[] array) {
  for (int i = 0; i < array.length; i++) {
    if (array[i] == i) {
      return i;
    }
  }
  return -1;
}
```

既然给定数组是有序的，我们理应充分利用这个条件。

你可能会发现这个问题与经典的二分查找问题大同小异，我们又该怎么运用二分查找法呢?

在二分查找中，要找出元素 k，我们会先拿它跟数组中间的元素 x 比较，确定 k 位于 x 的左边还是右边。

以此为基础，是否通过检查中间元素就能确定巧合索引的位置？下面来看一个样例数组。

-40	-20	-1	1	2	3	5	7	9	12	13
0	1	2	3	4	5	6	7	8	9	10

看到中间元素 A[5] = 3，我们可以断定巧合索引一定在数组右侧，因为 A[mid] < mid。

继续运用这个递归算法，就会写出与二分查找极为相似的代码。

```
int magicFast(int[] array) {
  return magicFast(array, 0, array.length - 1);
}

int magicFast(int[] array, int start, int end) {
  if (end < start) {
    return -1;
  }
  int mid = (start + end) / 2;
  if (array[mid] == mid) {
    return mid;
  } else if (array[mid] > mid){
    return magicFast(array, start, mid - 1);
  } else {
    return magicFast(array, mid + 1, end);
  }
}
```

进阶：如果数组元素有重复值，又该如何处理

如果数组元素有重复值，前面的算法就会失效。以下面的数组为例。

-10	-5	2	2	2	3	4	7	9	12	13
0	1	2	3	4	5	6	7	8	9	10

看到 A[mid] < mid，我们无法断定巧合索引位于数组哪一边。它可能在数组右侧，跟前面一样。或者，也可能在左侧（在本例中的确在左侧）。

它有没有可能在左侧的**任意位置**？未必。由 A[5] = 3 可知，A[4]不可能是巧合索引。A[4]必须等于 4，其索引才能成为巧合索引，但数组是有序的，故 A[4]必定小于 A[5]。

其实，看到 A[5] = 3 时，按照前面的做法，我们需要递归搜索右半部分。不过，若搜索左半部分，我们可以跳过一些元素，只递归搜索 A[0]到 A[3]的元素。A[3]是第一个可能成为巧合索引的元素。

综上所述，我们得到一般模式：先比较 midIndex 和 midValue 是否相同。然后，若两者不同，则按如下方式递归搜索左半部分和右半部分。

- 左半部分：搜索索引从 start 到 Math.min(midIndex - 1, midValue)的元素。
- 右半部分：搜索索引从 Math.max(midIndex + 1, midValue)到 end 的元素。

下面是该算法的实现代码。

```
1  int magicFast(int[] array) {
2    return magicFast(array, 0, array.length - 1);
3  }
4
5  int magicFast(int[] array, int start, int end) {
6    if (end < start) return -1;
7
8    int midIndex = (start + end) / 2;
9    int midValue = array[midIndex];
10   if (midValue == midIndex) {
11     return midIndex;
12   }
13
14   /* 搜索左半部分 */
15   int leftIndex = Math.min(midIndex - 1, midValue);
16   int left = magicFast(array, start, leftIndex);
17   if (left >= 0) {
18     return left;
19   }
20
21   /* 搜索右半部分 */
22   int rightIndex = Math.max(midIndex + 1, midValue);
23   int right = magicFast(array, rightIndex, end);
24
25   return right;
26 }
```

注意，在上面的代码中，如果数组元素各不相同，这个方法的执行动作与第一个解法几近相同。

8.3　汉诺塔问题

在经典汉诺塔问题中，有 3 根柱子及 *N* 个不同大小的穿孔圆盘，盘子可以滑入任意一根柱子。一开始，所有盘子自上而下按升序依次套在第一根柱子上（即每一个盘子只能放在更大的盘子上面）。移动圆盘时受到以下限制：

⑴ 每次只能移动一个盘子；

⑵ 盘子只能从柱子顶端滑出移到下一根柱子；

⑶ 盘子只能叠在比它大的盘子上。

请编写程序，用栈将所有盘子从第一根柱子移到最后一根柱子。

题目解法

我们先从最简单的例子 $n = 1$ 开始。

当 $n = 1$ 时，能否将盘子 1 从柱 1 移至柱 3？可以。

直接将盘子 1 从柱 1 移至柱 3。

当 $n = 2$ 时，能否将盘子 1 和盘子 2 从柱 1 移至柱 3？可以。

(1) 将盘子 1 从柱 1 移至柱 2。

(2) 将盘子 2 从柱 1 移至柱 3。

(3) 将盘子 1 从柱 2 移至柱 3。

注意，上述步骤将柱 2 用作缓冲区，在我们将其他盘子移至柱 3 时，柱 2 会暂存一个盘子。

当 $n = 3$ 时，能否将盘子 1、2、3 从柱 1 移至柱 3？可以。

(1) 从上面可知，我们可以将上面的两个盘子从一根柱子移至另一根柱子，假定已经这么做了，只不过，这里是将这两个盘子移至柱 2。

(2) 将盘子 3 移至柱 3。

(3) 将盘子 1、2 移至柱 3。重复步骤(1)即可。

当 $n = 4$ 时，能否将盘子 1、2、3、4 从柱 1 移至柱 3？可以。

(1) 将盘子 1、2、3 移至柱 2。具体做法参见前面的例子。

(2) 将盘子 4 移至柱 3。

(3) 将盘子 1、2、3 移至柱 3。

注意，柱 2 和柱 3 之间并无多大区别，只是叫法不一样，实则是等价的。把柱 2 作为缓冲将盘子移至柱 3，与把柱 3 用作缓冲将盘子移至柱 2，两者并无区别。

根据上述做法，很自然地就可以导出递归算法。在每一部分，我们都会执行以下步骤，用伪码简述如下。

```
moveDisks(int n, Tower origin, Tower destination, Tower buffer) {
  /* 基线条件 */
  if (n <= 0) return;

  /* 将顶端 n - 1 个盘子从 origin 移至 buffer，将 destination 用作缓冲区 */
  moveDisks(n - 1, origin, buffer, destination);

  /* 将 origin 顶端的盘子移至 destination */
  moveTop(origin, destination);

  /* 将顶部 n - 1 个盘子从 buffer 移至 destination，将 origin 用作缓冲区 */
  moveDisks(n - 1, buffer, destination, origin);
}
```

下面的代码比较详细地给出了这个算法的实现方式，其中还用到了面向对象设计这一概念。

```
void main(String[] args)
  int n = 3;
```

```
    Tower[] towers = new Tower[n];
    for (int i = 0; i < 3; i++) {
      towers[i] = new Tower(i);
    }

    for (int i = n - 1; i >= 0; i--) {
      towers[0].add(i);
    }
    towers[0].moveDisks(n, towers[2], towers[1]);
  }

  class Tower {
    private Stack<Integer> disks;
    private int index;
    public Tower(int i) {
      disks = new Stack<Integer>();
      index = i;
    }

    public int index() {
      return index;
    }

    public void add(int d) {
      if (!disks.isEmpty() && disks.peek() <= d) {
        System.out.println("Error placing disk " + d);
      } else {
        disks.push(d);
      }
    }

    public void moveTopTo(Tower t) {
      int top = disks.pop();
      t.add(top);
    }

    public void moveDisks(int n, Tower destination, Tower buffer) {
      if (n > 0) {
        moveDisks(n - 1, buffer, destination);
        moveTopTo(destination);
        buffer.moveDisks(n - 1, destination, this);
      }
    }
  }
```

8.4　颜色填充

编写函数，实现许多图片编辑软件都支持的“颜色填充”功能。给定一个屏幕（以二维数组表示，元素为颜色值）、一个点和一个新的颜色值，将新颜色值填入这个点的周围区域，直到原来的颜色值全都改变。

题目解法

首先，想象一下这个方法是怎么回事。假设要对一个像素（比如绿色）调用 `paintFill`（也即点击图片编辑软件的填充颜色），我们希望颜色向四周“渗出”。我们会对周围的像素逐一调用 `paintFill`，向外扩张，一旦碰到非绿色的像素就停止填充。

```
enum Color { Black, White, Red, Yellow, Green }
```

```
2
3  boolean PaintFill(Color[][] screen, int r, int c, Color ncolor) {
4    if (screen[r][c] == ncolor) return false;
5    return PaintFill(screen, r, c, screen[r][c], ncolor);
6  }
7
8  boolean PaintFill(Color[][] screen, int r, int c, Color ocolor, Color ncolor) {
9    if (r < 0 || r >= screen.length || c < 0 || c >= screen[0].length) {
10     return false;
11   }
12
13   if (screen[r][c] == ocolor) {
14     screen[r][c] = ncolor;
15     PaintFill(screen, r - 1, c, ocolor, ncolor); // 上
16     PaintFill(screen, r + 1, c, ocolor, ncolor); // 下
17     PaintFill(screen, r, c - 1, ocolor, ncolor); // 左
18     PaintFill(screen, r, c + 1, ocolor, ncolor); // 右
19   }
20   return true;
21 }
```

如果你用变量 x 和 y 来表示，要特别注意 screen[y][x]中 x 和 y 的顺序，碰到图像问题时切记这一点。因为 x 表示**水平轴**（即自左向右），实际上对应列数而非行数。y 的值等于行数。在面试以及平时写代码时，这个地方也很容易犯错。通常使用“行”和“列”来代替会更加清晰。

这个算法它本质上是图的深度优先遍历。对于每个格子，我们都向外搜索环绕着它的格子，直到这个颜色的格子周围的每个格子都被遍历过才终止。这个问题也可以用广度优先遍历来做。

8.5 硬币

给定数量不限的硬币，币值为 25 美分、10 美分、5 美分和 1 美分，编写代码计算 *n* 美分有几种表示法。

题目解法

这是个递归问题，我们要找出如何利用较早的答案（也就是子问题的答案）计算出 makeChange(n)。

假设 $n = 100$，我们想要算出 100 美分有几种换零方式。

已知 100 美分换零后会包含 0、1、2、3 或 4 个 25 美分硬币（quarter），由此得出如下算法。

```
makeChange(100) = makeChange(100，使用 0 个 25 美分硬币) +
                  makeChange(100，使用 1 个 25 美分硬币) +
                  makeChange(100，使用 2 个 25 美分硬币) +
                  makeChange(100，使用 3 个 25 美分硬币) +
                  makeChange(100，使用 4 个 25 美分硬币)
```

仔细观察，可以看出其中有些问题简化了。举个例子，makeChange(100，使用 1 个 25 美分硬币)与 makeChange(75，使用 0 个 25 美分硬币)等价。这是因为，如果给 100 美分换零时只准用 1 个 25 美分硬币，那么，我们就只能选择给余下的 75 美分换零。

同样地，这也适用于 makeChange(100，使用 2 个 25 美分硬币)、makeChange(100，使用 3 个 25 美分硬币)和 makeChange(100，使用 4 个 25 美分硬币)。综上所述，前面的算式可简化如下。

```
makeChange(100) = makeChange(100，使用 0 个 25 美分硬币) +
                  makeChange(75，使用 0 个 25 美分硬币) +
```

```
makeChange(50，使用 0 个 25 美分硬币) +
makeChange(25，使用 0 个 25 美分硬币) +
1
```

注意最后一行，makeChange(100，使用 4 个 25 美分硬币)等于 1。我们把这叫作“完全简化”。

接下来呢？我们已经用完了 25 美分硬币，现在开始使用下一个币值最大的硬币：10 美分硬币（dime）。

前面使用 25 美分硬币的做法同样可以套用在 10 美分硬币上，但需要套用在上面算式 5 部美分中的 4 个部美分上，且每一部美分都要套用。第一部美分的套用结果如下。

```
makeChange(100，使用 0 个 25 美分硬币) = makeChange(100，使用 0 个 25 美分硬币、0 个 10 美分硬币) +
                                         makeChange(100，使用 0 个 25 美分硬币、1 个 10 美分硬币) +
                                         makeChange(100，使用 0 个 25 美分硬币、2 个 10 美分硬币) +
                                         ...
                                         makeChange(100，使用 0 个 25 美分硬币、10 个 10 美分硬币)

makeChange(75，使用 0 个 25 美分硬币) = makeChange(75，使用 0 个 25 美分硬币、0 个 10 美分硬币) +
                                        makeChange(75，使用 0 个 25 美分硬币、1 个 10 美分硬币) +
                                        makeChange(75，使用 0 个 25 美分硬币、2 个 10 美分硬币) +
                                        ...
                                        makeChange(75，使用 0 个 25 美分硬币、7 个 10 美分硬币)

makeChange(50，使用 0 个 25 美分硬币) = makeChange(50，使用 0 个 25 美分硬币、0 个 10 美分硬币) +
                                        makeChange(50，使用 0 个 25 美分硬币、1 个 10 美分硬币) +
                                        makeChange(50，使用 0 个 25 美分硬币、2 个 10 美分硬币) +
                                        ...
                                        makeChange(50，使用 0 个 25 美分硬币、5 个 10 美分硬币)

makeChange(25，使用 0 个 25 美分硬币) = makeChange(25，使用 0 个 25 美分硬币、0 个 10 美分硬币) +
                                        makeChange(25，使用 0 个 25 美分硬币、1 个 10 美分硬币) +
                                        makeChange(25，使用 0 个 25 美分硬币、2 个 10 美分硬币)
```

开始使用 5 美分硬币（nickel）时，上面算式的每一部美分都要逐一展开，最终会得到一个树状递归结构，其中每次调用都会展开为 4 个或更多调用。

递归的基线条件就是完全简化的算式。举个例子，makeChange(50，使用 0 个 25 美分硬币、5 个 10 美分硬币)会被完全简化为 1，因为 5 个 10 美分硬币就等于 50 美分。

综上所述，可导出类似下面这样的递归算法。

```
1   int makeChange(int amount, int[] denoms, int index) {
2     if (index >= denoms.length - 1) return 1; // 最后一种币值
3     int denomAmount = denoms[index];
4     int ways = 0;
5     for (int i = 0; i * denomAmount <= amount; i++) {
6       int amountRemaining = amount - i * denomAmount;
7       ways += makeChange(amountRemaining, denoms, index + 1);
8     }
9     return ways;
10  }
11
12  int makeChange(int n) {
13    int[] denoms = {25, 10, 5, 1};
14    return makeChange(n, denoms, 0);
15  }
```

这样的解法虽然正确，却不怎么高效。因为即使对于相同的 amount 和 index，也会多次调用 makeChange 方法。

优化时，只要把计算过的值存起来就可以了。我们需要存储的是每一对(amout, index)和

对应结果的映射。

```
int makeChange(int n) {
  int[] denoms = {25, 10, 5, 1};
  int[][] map = new int[n + 1][denoms.length]; // 预处理值
  return makeChange(n, denoms, 0, map);
}

int makeChange(int amount, int[] denoms, int index, int[][] map) {
  if (map[amount][index] > 0) { // 检索对应值
    return map[amount][index];
  }
  if (index >= denoms.length - 1) return 1; // 还剩一个面值
  int denomAmount = denoms[index];
  int ways = 0;
  for (int i = 0; i * denomAmount <= amount; i++) {
    // 继续求下一个面值，假设面值为 denomAmount 的硬币有 i 个
    int amountRemaining = amount - i * denomAmount;
    ways += makeChange(amountRemaining, denoms, index + 1, map);
  }
  map[amount][index] = ways;
  return ways;
}
```

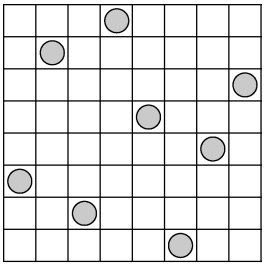
请注意，我们使用了一个二维的整数数组来存储计算过的值。这样简单一些，但占用了额外空间。也可以用真正的散列表，把币值映射成一个新的散列表，新散列表是面值（denom）到预计算值的映射。或者选其他的数据结构也可以。

8.6 国际象棋八皇后

在 8 × 8 格的国际象棋盘上摆 8 个皇后，使其不能互相攻击，即任意两个皇后都不能处在同一行、同一列或同一斜线上，问有多种解法（斜线指棋盘上所有斜线）。

题目解法

拆分题目可知，每一行、每一列以及对角线只能使用一次。



解决八皇后的一个棋局

想象一下最后放到棋盘上的那个皇后，这里假设是在第 8 行。这个皇后要摆在第 8 行的哪一格呢？一共有 8 种选择，每一列代表一种可能。

具体算法如下。

```
八皇后在 8 × 8 棋盘上的摆法 =
    八皇后在 8 × 8 棋盘上的摆法，且其中一个皇后位于(7, 0) +
    八皇后在 8 × 8 棋盘上的摆法，且其中一个皇后位于(7, 1) +
    八皇后在 8 × 8 棋盘上的摆法，且其中一个皇后位于(7, 2) +
    八皇后在 8 × 8 棋盘上的摆法，且其中一个皇后位于(7, 3) +
```

```
八皇后在 8×8 棋盘上的摆法，且其中一个皇后位于(7, 4) +
八皇后在 8×8 棋盘上的摆法，且其中一个皇后位于(7, 5) +
八皇后在 8×8 棋盘上的摆法，且其中一个皇后位于(7, 6) +
八皇后在 8×8 棋盘上的摆法，且其中一个皇后位于(7, 7)
```

接着，运用相似的方法计算其中的每一项。

```
八皇后在 8×8 棋盘上的摆法，且其中一个皇后位于(7, 3) =
    八皇后在……的摆法，且其中两个皇后位于(7, 3)和(6, 0) +
    八皇后在……的摆法，且其中两个皇后位于(7, 3)和(6, 1) +
    八皇后在……的摆法，且其中两个皇后位于(7, 3)和(6, 2) +
    八皇后在……的摆法，且其中两个皇后位于(7, 3)和(6, 4) +
    八皇后在……的摆法，且其中两个皇后位于(7, 3)和(6, 5) +
    八皇后在……的摆法，且其中两个皇后位于(7, 3)和(6, 6) +
    八皇后在……的摆法，且其中两个皇后位于(7, 3)和(6, 7)
```

注意，我们不必考虑皇后位于格子(7, 3)和(6, 3)的组合情况，因为这与所有皇后不同行、不同列且不在斜线上的要求不符。

```
int GRID_SIZE = 8;

void placeQueens(int row, Integer[] columns, ArrayList<Integer[]> results) {
  if (row == GRID_SIZE) { // 找到有效摆法
    results.add(columns.clone());
  } else {
    for (int col = 0; col < GRID_SIZE; col++) {
      if (checkValid(columns, row, col)) {
        columns[row] = col; // 摆放皇后
        placeQueens(row + 1, columns, results);
      }
    }
  }
}

/* 检查(row1, column1)可否摆放皇后，做法是检查
 * 有无其他皇后位于同一列或对角线。不必检查是否
 * 在同一行上，因为调用 placeQueen 时，一次只会
 * 摆放一个皇后。由此可知，这一行是空的 */
boolean checkValid(Integer[] columns, int row1, int column1) {
  for (int row2 = 0; row2 < row1; row2++) {
    int column2 = columns[row2];
    /* 检查摆放在(row2, column2)是否会
     * 让(row1, column1)变成无效 */

    /* 检查同一列是否有其他皇后 */
    if (column1 == column2) {
      return false;
    }

    /* 检查对角线：若两列的距离等于两行的
     * 距离，就表示两个皇后在同一对角线上 */
    int columnDistance = Math.abs(column2 - column1);

    /* row1 > row2，不用取绝对值 */
    int rowDistance = row1 - row2;
    if (columnDistance == rowDistance) {
      return false;
    }
  }
  return true;
}
```

注意，每一行只能摆放一个皇后，因此不需要将棋盘存储为完整的 8×8 矩阵，只需一维数组，其中 `columns[r] = c` 表示有个皇后位于行 `r` 列 `c`。

8.7 堆箱子

给你一堆 n 个箱子，箱子宽 w_i、高 h_i、深 d_i。箱子不能翻转，将箱子堆起来时，下面箱子的宽度、高度和深度必须大于上面的箱子。实现一种方法，搭出最高的一堆箱子。箱堆的高度为每个箱子高度的总和。

题目解法

解法 1

假设我们有以下这些箱子：$b_1, b_2, \cdots, b_n$。能够堆出的最高箱堆的高度等于 `max`（底部为 b_1 的最高箱堆，底部为 b_2 的最高箱堆，……，底部为 b_n 的最高箱堆），这样，只要试着用每个箱子作为箱堆底部并搭出可能的最高高度，就能找出箱堆的最高高度。

但要怎么找出以那个箱子为底箱堆最高呢？我们需要试着在第二层以不同的箱子为底继续堆箱子，如此反复。

因为问题已经明确规定了下面的箱子在三维上必须严格大于上面的箱子。所以，我们可以先按照任意维度降序排列箱子，这样就不必往列表后面寻找。比方说，b_1 不可能在 b_5 的上面，因为它的高度比 b_5 高。

下面代码是该算法的递归版本。

```
int createStack(ArrayList<Box> boxes) {
  /* 基于高度降序排序 */
  Collections.sort(boxes, new BoxComparator());
  int maxHeight = 0;
  for (int i = 0; i < boxes.size(); i++) {
    int height = createStack(boxes, i);
    maxHeight = Math.max(maxHeight, height);
  }
  return maxHeight;
}

int createStack(ArrayList<Box> boxes, int bottomIndex) {
  Box bottom = boxes.get(bottomIndex);
  int maxHeight = 0;
  for (int i = bottomIndex + 1; i < boxes.size(); i++) {
    if (boxes.get(i).canBeAbove(bottom)) {
      int height = createStack(boxes, i);
      maxHeight = Math.max(height, maxHeight);
    }
  }
  maxHeight += bottom.height;
  return maxHeight;
}

class BoxComparator implements Comparator<Box> {
  @Override
  public int compare(Box x, Box y){
    return y.height - x.height;
  }
}
```

上述代码的缺点是效率太低，以上述方法为基础，运用制表法可以优化效率问题。

```
int createStack(ArrayList<Box> boxes) {
  Collections.sort(boxes, new BoxComparator());
  int maxHeight = 0;
  int[] stackMap = new int[boxes.size()];
  for (int i = 0; i < boxes.size(); i++) {
    int height = createStack(boxes, i, stackMap);
    maxHeight = Math.max(maxHeight, height);
  }
  return maxHeight;
}

int createStack(ArrayList<Box> boxes, int bottomIndex, int[] stackMap) {
  if (bottomIndex < boxes.size() && stackMap[bottomIndex] > 0) {
    return stackMap[bottomIndex];
}

  Box bottom = boxes.get(bottomIndex);
  int maxHeight = 0;
  for (int i = bottomIndex + 1; i < boxes.size(); i++) {
    if (boxes.get(i).canBeAbove(bottom)) {
      int height = createStack(boxes, i, stackMap);
      maxHeight = Math.max(height, maxHeight);
    }
  }
  maxHeight += bottom.height;
  stackMap[bottomIndex] = maxHeight;
  return maxHeight;
}
```

我们仅需要把索引映射到高度，就可以用一个整数数组来充当“散列表”。

但需要注意散列表中每个位置所代表的意义。在上面的代码中，`stackMap[i]`代表以箱子 i 为底的最大箱子堆高度。所以在从散列表中取值之前，你得保证箱子 i 可以放在当前底部箱子的上面。

这样保持回调链是线性的，让散列表中的取出和插入保持对称。例如，我们在该方法的起始处回调散列表的 `bottomIndex`，在方法的结尾处插入 `bottomIndex` 位置的值。

解法 2

我们也可以考虑用递归算法做抉择，在每一步选择是否把箱子放在堆上。首先依旧在任一维度把箱子降序排序。

面临的第一个问题是否把 0 位置上的箱子放到堆里。这样就划分了两条递归路径，一个以箱子 0 为底，一个不以箱子 0 为底。然后直接返回这两种选择中较好的那个。

其次，我们选择是否把 1 位置上的箱子放入堆里。同上一个箱子一样，选择是否放入箱子 1。然后返回两条递归路径中高度的最大值。 同样，我们再次使用制表法缓存以每个箱子为底的箱子堆最大高度。

```
int createStack(ArrayList<Box> boxes) {
  Collections.sort(boxes, new BoxComparator());
  int[] stackMap = new int[boxes.size()];
  return createStack(boxes, null, 0, stackMap);
}

int createStack(ArrayList<Box> boxes, Box bottom, int offset, int[] stackMap) {
  if (offset >= boxes.size()) return 0; // 基本情况

  /* 以这个箱子为底的高度 */
  Box newBottom = boxes.get(offset);
```

```
12    int heightWithBottom = 0;
13    if (bottom == null || newBottom.canBeAbove(bottom)) {
14      if (stackMap[offset] == 0) {
15        stackMap[offset] = createStack(boxes, newBottom, offset + 1, stackMap);
16        stackMap[offset] += newBottom.height;
17      }
18      heightWithBottom = stackMap[offset];
19    }
20
21    /* 不以这个箱子为底 */
22    int heightWithoutBottom = createStack(boxes, bottom, offset + 1, stackMap);
23
24    /* 返回最佳选择 */
25    return Math.max(heightWithBottom, heightWithoutBottom);
26  }
```

要格外注意回调及往散列表中插入值的时机。如第 15 行和第 16~18 行那种对称通常就是最佳的。

第 9 章

系统设计与可扩展性

9.1 股票数据

假设你正在搭建某种服务，有多达 1000 个客户端软件会调用该服务，取得每天盘后股票价格信息（开盘价、收盘价、最高价与最低价）。同时你手里已有这些数据，存储格式可自行定义。你会如何设计这套服务以向客户端软件提供信息？你需要负责该服务的研发、部署、持续监控和维护。请描述你想到的各种实现方案，以及为何推荐采用你的方案。（该服务的实现技术和信息分发机制可任选）。

题目解法

此题的重点是如何真正地将信息分发给客户端。首先假设已有收集信息的脚本、其次方案还需要满足什么诉求。

- **客户端软件易用性**。我们希望这套服务对客户端实现起来又容易又好用。
- **实现难度不高**。我们需要考虑的不仅有研发成本，还有维护成本不要太高。
- **灵活应对未来需求**。此题的问法是“在现实世界中你会怎么做”，因此，我们应该从解决实际问题的角度来思考方法要能适应更多情景。
- **扩展性和效率**。关注实现方案的效率，才不会让服务负担过重。

方案 1

一种选择是，将数据直接保存在纯文本文件中，让客户端通过某种 FTP 服务器下载。虽然这么做容易维护，但需要更复杂的文件解析才能实现各种查询。而且，若这些文件有新增数据，可能会打乱客户端的解析机制。

方案 2

我们可以使用标准的 SQL 数据库，让客户端直接接入。这么做有如下优点。

- 如需支持新功能，这种做法提供了一种让客户端查询和处理数据的简单方式。例如，返回开盘价高于 N 且收盘价低于 M 的所有股票。
- 利用标准的数据库功能就能提供数据回滚、数据备份和各种安全保障。避免重复性劳动。
- 客户端可以很容易地整合现有应用。在各种软件开发环境中，SQL 整合是标准功能。

缺点呢？

- 有冗余功能造成额外负担。
- 为保障数据的查看与维护需要额外开发。
- 数据安全有潜在风险。

方案 3

就分发信息而言，XML 也是一种不错的选择。采用 XML 时，数据有固定的格式和大小：company name（公司名）、open（开盘价）、high（最高价）、low（最低价）、closingPrice（收盘价），下面是一个 XML 格式的数据样例。

```
<root>
  <date value="2008-10-12">
    <company name="foo">
      <open>126.23</open>
      <high>130.27</high>
      <low>122.83</low>
      <closingPrice>127.30</closingPrice>
    </company>
    <company name="bar">
       <open>52.73</open>
       <high>60.27</high>
       <low>50.29</low>
       <closingPrice>54.91</closingPrice>
    </company>
  </date>
  <date value="2008-10-11"> . . . </date>
</root>
```

优点如下：

- 容易分发，也容易为机器和人类所识别。（这也是 XML 成为分享和分发数据的标准数据模型的原因之一。）
- 大多数语言都有执行 XML 解析的库，因此客户端实现起来也很容易。
- 在 XML 文件中增加新节点就可以添加新数据。不会打乱客户端解析器。
- 数据以 XML 文件格式存储，我们可以利用现有工具备份数据，不必自己重新做一套。

缺点是：

- 这种做法会向客户端发送所有信息，（有用的 + 没用的）效率很低。
- 进行数据查询时，必须解析整个文件。

此外我们还可以提供 Web 服务（比如 SOAP）以供客户端存取数据。虽然这会在工作中多加一层，但它能够增加安全保障，还能使客户更易整合系统。

但这也有利也有弊，客户端将只能按我们预设的方式获取数据。相比在纯 SQL 实现中，即使我们没有预料到客户端需要查询最高股价，它们还是可以进行查询的。

至于选择哪种方案，全看你如何设计一个系统，怎么权衡利弊。

9.2 社交网络

你会如何设计诸如 Facebook 或 LinkedIn 的超大型社交网站的数据结构？请设计一种算法，展示两人之间最短的社交路径（比如，我 → 小明 → 王总 → 梅姐 → 你）。

题目解法

步骤 1：简化问题——先忘记有几百万用户

首先，我们可以构造一个图，把每个人看作一个节点，两个节点之间若有连线，则表示这两个用户为朋友。

要找到两个人之间的连接，可以从其中一人开始，直接进行广度优先搜索。

为什么深度优先搜索效果不彰呢？因为它只能找到一条连接，还不一定是最短的。其次，即使任一连接都可以，它效率也很低，两个用户可能只有一度之隔，却可能要在他们的“子树”中搜索几百万个节点后，才能找到这条非常简单而直接的连接。

也可以双向广度优先搜索，即做两个广度优先搜索，一个是来源，另一个是目的地。当搜索相遇时，我们就找到了一条连接。

在实现的过程中，使用两个类会更优。`BFSData` 保存我们需要进行广度优先搜索的数据，比如 `isVisited` 散列表和 `toVisit` 队列。`PathNode` 代表着我们正在搜索的连接，存储每个 `Person` 和我们在这个连接中访问的 `previousNode`。

```
LinkedList<Person> findPathBiBFS(HashMap<Integer, Person> people, int source,
                                 int destination) {
  BFSData sourceData = new BFSData(people.get(source));
  BFSData destData = new BFSData(people.get(destination));

  while (!sourceData.isFinished() && !destData.isFinished()) {
    /* 从出发点开始搜索 */
    Person collision = searchLevel(people, sourceData, destData);
    if (collision != null) {
      return mergePaths(sourceData, destData, collision.getID());
    }

    /* 从目的地开始搜索 */
    collision = searchLevel(people, destData, sourceData);
    if (collision != null) {
      return mergePaths(sourceData, destData, collision.getID());
    }
  }
  return null;
}

/* 搜索一层，若有碰撞则返回 */
Person searchLevel(HashMap<Integer, Person> people, BFSData primary,
                   BFSData secondary) {
  /* 我们每次只想搜索一个级别。计算当前主节点中有多少个节点，只搜索那么多，
   * 随后将这些节点加到末尾 */
  int count = primary.toVisit.size();
  for (int i = 0; i < count; i++) {
    /* 取出第一个节点 */
    PathNode pathNode = primary.toVisit.poll();
    int personId = pathNode.getPerson().getID();

    /* 检查是否已经访问过 */
    if (secondary.visited.containsKey(personId)) {
      return pathNode.getPerson();
    }

    /* 把朋友添加到队列中 */
    Person person = pathNode.getPerson();
    ArrayList<Integer> friends = person.getFriends();
    for (int friendId : friends) {
      if (!primary.visited.containsKey(friendId)) {
        Person friend = people.get(friendId);
        PathNode next = new PathNode(friend, pathNode);
        primary.visited.put(friendId, next);
        primary.toVisit.add(next);
      }
    }
```

```
  }
  return null;
}

/* 在搜索碰撞地方合并连接 */
LinkedList<Person> mergePaths(BFSData bfs1, BFSData bfs2, int connection) {
  PathNode end1 = bfs1.visited.get(connection); // end1 -> 起点
  PathNode end2 = bfs2.visited.get(connection); // end2 -> 目的地
  LinkedList<Person> pathOne = end1.collapse(false);
  LinkedList<Person> pathTwo = end2.collapse(true); // 反转
  pathTwo.removeFirst(); // 移除连接
  pathOne.addAll(pathTwo); // 添加第二个连接
  return pathOne;
}

class PathNode {
  private Person person = null;
  private PathNode previousNode = null;
  public PathNode(Person p, PathNode previous) {
    person = p;
    previousNode = previous;
  }

  public Person getPerson() { return person; }

  public LinkedList<Person> collapse(boolean startsWithRoot) {
    LinkedList<Person> path = new LinkedList<Person>();
    PathNode node = this;
    while (node != null) {
      if (startsWithRoot) {
        path.addLast(node.person);
      } else {
        path.addFirst(node.person);
      }
      node = node.previousNode;
    }
    return path;
  }
}

class BFSData {
  public Queue<PathNode> toVisit = new LinkedList<PathNode>();
  public HashMap<Integer, PathNode> visited =
    new HashMap<Integer, PathNode>();

  public BFSData(Person root) {
    PathNode sourcePath = new PathNode(root, null);
    toVisit.add(sourcePath);
    visited.put(root.getID(), sourcePath);
  }

  public boolean isFinished() {
    return toVisit.isEmpty();
  }
}
```

为什么这种方式更优。

假设每个人有 k 个朋友，节点 S 和 D 有一个共同的朋友 C。

- 从 S 到 D 的传统的广度优先搜索：我们大概会经过 $k + k \times k$ 个节点，分别来自 S 的 k 个朋友以及他们各自的 k 个朋友。

❑ 双向的广度优先搜索：只需要经过 $2k$ 个节点，即 S 的 k 个朋友和 D 的 k 个朋友。

$2k$ 和 $k + k \times k$ 相比，显而易见，$2k$ 更小些。

把它推广到一个长度为 q 的路径，可以由此得出下面两种情况。

❑ 广度优先搜索：$O(k^q)$。

❑ 双向广度优先搜索：$O(k^{q/2} + k^{q/2})$，即 $O(k^{q/2})$。

想象一个像 A -> B -> C -> D -> E 这样的路径，每个人有 100 个朋友，观察两者的表现。BFS 需要查看 1 亿（100^4）个节点，而双向 BFS 只需要查看 2 万个节点（100^2）。双向 BFS 一般会比传统的 BFS 更快。但它除了访问源节点外还需要访问目标节点，这个要求并非每次能满足。

步骤 2：处理数百万的用户

处理 LinkedIn 或 Facebook 这种规模的服务时，不可能将所有数据存放在一台机器上。之前的简单数据结构 Person 就不管用了，因为朋友的资料和我们的资料可能不在同一台机器上。我们需要，将朋友列表改为他们 ID 的列表，并按如下方式追踪。

(1) 针对每个朋友 ID，找出所在机器的位置：int machine_index = getMachineIDForUser (personID);。

(2) 转到编号为#machine_index 的机器。

(3) 在那台机器上，执行：Person friend = getPersonWithID(person_id);。

我们定义了一个 Server 类，包含一份所有机器的列表，还有一个 Machine 类，代表一台单独的机器。这两个类都用了散列表，从而有效地查找数据。下面的代码描绘了这一过程。

```
1  class Server {
2    HashMap<Integer, Machine> machines = new HashMap<Integer, Machine>();
3    HashMap<Integer, Integer> personToMachineMap = new HashMap<Integer, Integer>();
4
5    public Machine getMachineWithId(int machineID) {
6      return machines.get(machineID);
7    }
8
9    public int getMachineIDForUser(int personID) {
10     Integer machineID = personToMachineMap.get(personID);
11     return machineID == null ? -1 : machineID;
12   }
13
14   public Person getPersonWithID(int personID) {
15     Integer machineID = personToMachineMap.get(personID);
16     if (machineID == null) return null;
17
18     Machine machine = getMachineWithId(machineID);
19     if (machine == null) return null;
20
21     return machine.getPersonWithID(personID);
22   }
23 }
24
25 class Person {
26   private ArrayList<Integer> friends = new ArrayList<Integer>();
27   private int personID;
28   private String info;
29
30   public Person(int id) { this.personID = id; }
31   public String getInfo() { return info; }
32   public void setInfo(String info) { this.info = info; }
33   public ArrayList<Integer> getFriends() { return friends; }
```

```
34    public int getID() { return personID; }
35    public void addFriend(int id) { friends.add(id); }
36  }
```

其他优化可能。

减少机器间跳转的次数

从一台机器跳转到另一台机器的花费过高。不要单个跳转，试着批处理这些跳转动作。举例来说，如果有 5 个朋友都在同一台机器上，那就应该一次性找出来。

优化：智能划分用户和机器

人们跟生活在同一国家的人成为朋友的可能性更大。因此，不要随意将用户划分到不同机器上，而应该尽量按国家、城市、州等进行划分。这样一来，就可以减少跳转的次数。

问题：广度优先搜索通常要求“标记”访问过的节点。在这种情况下，你会怎么做

在广度优先搜索中，通常我们会设定节点类的 `visited` 标志，以标记访问过的节点。但对这道题来说不太好。因为同一时间可能会执行很多搜索操作，所以直接编辑数据的做法并不妥当。我们可以利用散列表模仿节点的标记动作，以查询节点 id，看它是否访问过。

其他扩展问题

- 思考服务器会出故障。会对你造成什么影响？
- 如何利用缓存？
- 如果一直搜索，直到图的终点（无限）吗？该如何判断何时放弃？

这些扩展问题，你在面试时可能会遇到，可以想想。

9.3 文本分享

设计一个类似于 Pastebin 的系统，用户输入一段文本，就可以得到一个随机生成的 URL 来访问该系统。

题目解法

步骤 1：确定问题的范围

- 系统不支持用户账户或编辑文档。
- 系统可以跟踪分析每个页面访问次数。
- 旧文档在长时间不被访问后会被删除。
- 虽然在访问文档时没有真实的身份验证，但用户不应该轻松猜到文档的 URL。
- 该系统有前端和 API 接口。
- 每个 URL 的分析可以通过对应页面上的“统计”链接访问。但是，默认情况下不显示。

步骤 2：作出合理的假设

- 系统流量大，包含数百万个文档。
- 文档的访问量不是均匀分布的。一些文档更会被多次访问。

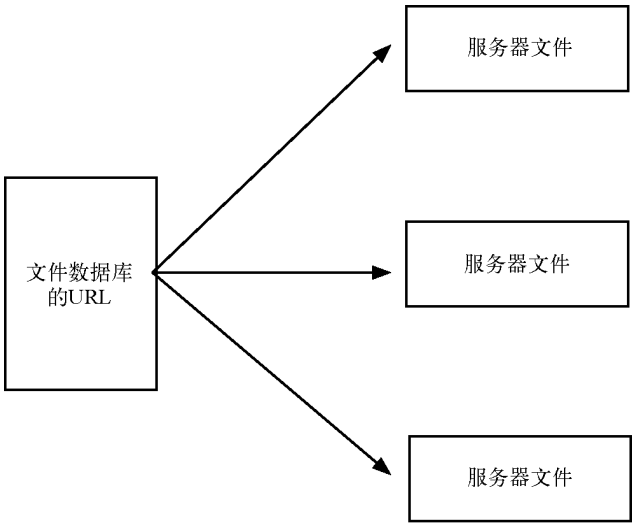
步骤 3：绘制主要组件

我们可以先勾勒出一个简单的设计。我们需要跟踪 URL 和与之对应的文件，以及关于文件访问频率的分析。

存储文件有如下两种选择：可以将它们存储在数据库中，也可以将它们存储在文件中。由

于文件可能很大，而且我们不太需要搜索功能，因此将它们存储在文件中更好。

下图这样一个简单的设计可能就合适。



在这里我们有一个简单的数据库，用于查找每个文件的位置，即服务器和路径。当我们请求一个 URL 时，先在数据存储中查找 URL 的位置，然后访问文件。

另外，我们需要一个跟踪分析的数据库。一个简单的数据存储即可，它将每次访问的时间戳、IP 地址和位置作为一行添加到数据库中。当需要访问这些统计信息时，便从该数据库中拉取相关数据。

步骤 4：确定关键问题

第一个问题就是，如何保证热门文本的访问速度。与从内存中读取数据相比，从文件系统读取数据相对较慢。因此，我们需要使用缓存来存储最近访问的文档。这会确保热门文件的访问速度较快。由于文件不能编辑，我们不需要担心缓存失效。

我们也应该考虑分解数据库。我们可以依据 URL 的映射来分割它，比如用 URL 的散列码（hash code）以某个整数为模，这将使我们能够快速定位包含该文件的数据库。

我们甚至可以作进一步优化。跳过数据库，只是让 URL 的散列值表示哪个服务器包含文档。URL 本身可以代表文档的位置。但缺点是如果我们需要添加服务器时，可能很难重新分配文档。

- 生成 URL

我们并不希望它是个单调递增的整数值，因为这样用户很容易就猜到规律。

一条简单的路径是生成一个随机 GUID，例如，5d50e8ac-57cb-4a0d-8661-bcdee2548979。这是一个 128 位的值，尽管不能保证是唯一的，但它发生冲突的概率极低，我们可以将其视为“唯一”。该方案的缺点是这样的 URL 对用户来说并不是很“漂亮”。我们可以将它散列到一个更小的值，但那会增加发生冲突的概率。

不过，我们同样可以这样做：生成一个 10 个字符的字母和数字序列，这给我们提供了 36^{10} 个可能的字符串。即使有 10 亿个 URL，任何特定 URL 的发生冲突的概率都很低。

如果不能接受出现数据牧的情况，我们可以检查数据存储库以查看 URL 是否存在，或者如果 URL 映射到特定服务器，则只需检测目标位置是否存在文件。

发生冲突时，我们可以生成一个新的 URL。因为有 36^{10} 个可用的 URL，而冲突概率极低，

所以使用检测冲突和重试这种省事的方法就足够了。

- 分析

最后要讨论的部分是分析。我们如果想要显示访问次数，并按时间或位置划分。会面临如下两种选择。

(1) 存储每行的原始数据。

(2) 只存储需要用到的数据，比如访问次数。

你可以与面试官讨论这个问题，但建议存储原始数据。原始数据可以让我们更灵活地来应对未来分析的扩展需求。

但这并不意味着原始数据要易于搜索甚至可以被访问。 我们可以将访问日志存储在文件中，并将其备份到其他服务器。

如果数据量很大，怎么办?

我们可以通过存概率的方式存储数据，这样可以显著降低空间使用。每个 URL 都有一个关联的 `storage_probability`。随着网站热度的提升，`storage_probability` 会降低。例如：一个热门文档每 10 次访问记录 1 次数据，我们查看访问次数时，根据概率乘以 10 显示（误差范围可接受）。

日志文件适合频繁使用。我们还希望将预先计算的数据存储到数据库中。如果前端的分析栏仅需要显示随访问次数和固定时间段的数据图表，则可以将其保存在单独的数据库中。

链　接	日　期	访问次数
12ab31b92p	2013 年 12 月	242119
12ab31b92p	2014 年 1 月	429918
……	……	……

每次访问 URL 时，我们都可以增加相应的行和列。该数据存储也可以通过 URL 进行分片。

统计数据没有在常规页面上列出且关注度低，所以，应该不必担心过载的情况。我们仍可以将生成的 HTML 缓存在前端服务器上，这样保障热门 URL 的访问效率。

扩展思考

- ❑ 你将如何支持用户账户?
- ❑ 如何将新的分析（例如，推荐来源）添加到统计信息页面?
- ❑ 如果统计信息要关联文档一起显示，那你的设计要如何更改?

第 10 章

排序与查找

10.1 合并排序的数组

给定两个排序后的数组 A 和 B，其中 A 的末端有足够的缓冲空间容纳 B。编写一个方法，将 B 合并入 A 并排序。

题目解法

已知数组 A 末端有足够的缓冲区域，不需要再分配额外空间。只需逐一比较 A 和 B 中的元素，然后将 A 和 B 中的所有元素按顺序插入数组。

缺点是，如果将元素插入数组 A 的前端，就必须将原有的元素往后移动，以腾出空间。建议将元素插入数组 A 的末端，毕竟那里都是空闲的可用空间。从数组 A 和 B 的末端元素开始，将最大的元素放到数组 A 的末端。

下面的代码就实现了上述做法。

```
1  void merge(int[] a, int[] b, int lastA, int lastB) {
2    int indexA = lastA - 1; /* 数组 a 最后元素的索引 */
3    int indexB = lastB - 1; /* 数组 b 最后元素的索引 */
4    int indexMerged = lastB + lastA - 1; /* 合并后数组的最后元素索引 */
5
6    /* 合并 a 和 b，从这两个数组的最后元素开始 */
7    while (indexB >= 0) {
8      /* 数组 a 最后元素 > 数组 b 最后元素 */
9      if (indexA >= 0 && a[indexA] > b[indexB]) {
10       a[indexMerged] = a[indexA]; // 复制元素
11       indexA--;
12     } else {
13       a[indexMerged] = b[indexB]; // 复制元素
14       indexB--;
15     }
16     indexMerged--; // 更新索引
17   }
18 }
```

注意，处理完 B 的剩余元素后，不要复制 A 的剩余元素，它们已经在里面了。

10.2 稀疏数组搜索

有个排好序的字符串数组，其中散布着一些空字符串，编写一种方法，找出给定字符串的位置。

示例：

输入：在字符串数组{"at", "", "", "", "ball", "", "", "car", "", "", "dad", "", ""}中查找"ball"

输出：4

题目解法

如果没有空字符串，可以直接使用二分查找法。比较待查找字符串 str 和数组的中间元素，然后继续搜索下去。

通过向对数组中插入一些空字符串的方式，实现对二分查找法的修改，修改位置就是与 mid 进行比较的地方，如果 mid 为空字符串，就将 mid 换到离它最近的非空字符串的位置。

下面以递归方式解决此题，稍加修改，就可以用迭代法实现。

```
int search(String[] strings, String str, int first, int last) {
  if (first > last) return -1;
  /* 将 mid 移到中间 */
  int mid = (last + first) / 2;

  /* 若 mid 为空字符串，就找出离它最近的非空字符串 */
  if (strings[mid].isEmpty()) {
    int left = mid - 1;
    int right = mid + 1;
    while (true) {
      if (left < first && right > last) {
        return -1;
      } else if (right <= last && !strings[right].isEmpty()) {
        mid = right;
        break;
      } else if (left >= first && !strings[left].isEmpty()) {
        mid = left;
        break;
      }
      right++;
      left--;
    }
  }

  /* 检查字符串，如有必要则继续递归 */
  if (str.equals(strings[mid])) { // 找到了
    return mid;
  } else if (strings[mid].compareTo(str) < 0) { // 搜索右半边
    return search(strings, str, mid + 1, last);
  } else { // 搜索左半边
    return search(strings, str, first, mid - 1);
  }
}

int search(String[] strings, String str) {
  if (strings == null || str == null || str == "") {
    return -1;
  }
  return search(strings, str, 0, strings.length - 1);
}
```

在最坏情况下，该算法的运行时间是 $O(n)$。毕竟，我们可能需要查看数组中的每个元素以发现非空字符。

关于空字符的处理方式，你可以与面试官进行讨论。

10.3　大文件排序

设想你有个 20 GB 的文件，每行有一个字符串，请阐述一下将如何对这个文件进行排序。

题目解法

当面试官给出 20 GB 大小的限制时，他们的言下之意是不希望你将数据全部载入内存。

我们可以把整个文件划分成许多块，每个块大小为 x MB，其中 x 是可用的内存大小。每个块各自进行排序，然后存回文件系统。

各个块一旦完成排序，我们便将这些块逐一合并在一起，最终就能得到全都排好序的文件。

这个算法被称为外部排序（external sort）。

10.4　寻找重复数

给定一个数组，包含 1 到 N 的整数，N 最大为 32 000，数组可能含有重复的值，且 N 的取值不定。若只有 4 KB 内存可用，该如何打印数组中所有重复的元素。

题目解法

我们有 4 KB 内存可用，也就是最多可寻址 $8 \times 4 \times 2^{10}$ 个比特。注意，32×2^{10} 要比 32 000 大。我们可以创建含有 32 000 个比特的位向量，其中每个比特代表一个整数。

利用这个位向量，就可以迭代访问整个数组，发现数组元素 v 时，就将位 v 设定为 1。碰到重复元素时，就打印出来。

```
1   void checkDuplicates(int[] array) {
2     BitSet bs = new BitSet(32000);
3     for (int i = 0; i < array.length; i++) {
4       int num = array[i];
5       int num0 = num - 1; // bitset 从 0 开始，数字从 1 开始
6       if (bs.get(num0)) {
7         System.out.println(num);
8       } else {
9         bs.set(num0);
10      }
11    }
12  }
13
14  class BitSet {
15    int[] bitset;
16
17    public BitSet(int size) {
18      bitset = new int[(size >> 5) + 1]; // 除以 32
19    }
20
21    boolean get(int pos) {
22      int wordNumber = (pos >> 5); // 除以 32
23      int bitNumber = (pos & 0x1F); // 除以 32 取余数
24      return (bitset[wordNumber] & (1 << bitNumber)) != 0;
25    }
26
27    void set(int pos) {
28      int wordNumber = (pos >> 5); // 除以 32
29      int bitNumber = (pos & 0x1F); // 除以 32 取余数
30      bitset[wordNumber] |= 1 << bitNumber;
31    }
32  }
```

注意，虽然此题不太难，但重要的是实现代码要写得干净利落。这也是为什么要定义位向量类来保存大型的位向量。要是面试官允许（也可能不会），那就可以使用 Java 内置的 `BitSet` 类。

10.5 数字流的秩

假设你正在读取一串整数。每隔一段时间，你希望能找出数字 x 的秩（小于或等于 x 的值的个数）。请实现数据结构和算法来支持这些操作，也就是说，实现 `track(int x)`方法，每读入一个数字都会调用该方法；实现 `getRankOfNumber(int x)`方法，返回小于或等于 x（x 除外）的值的个数。

示例：

```
数据流为（按出现的先后顺序）：5，1，4，4，5，9，7，13，3
getRankOfNumber(1) = 0
getRankOfNumber(3) = 1
getRankOfNumber(4) = 3
```

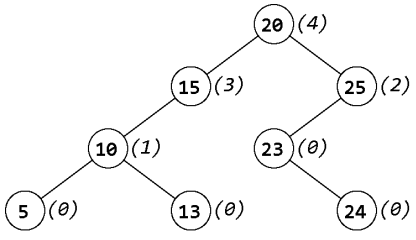
题目解法

有种相对简单的实现方式是用一个数组存放所有已排好序的元素。当有新元素进来时，我们需要搬移其他元素以腾出空间。这样，`getRankOfNumber` 实现起来就很容易，只需执行二分查找，返回索引。

但是，插入元素（也就是 `track(int x)`函数）将会非常低效，我们需要一种数据结构，不仅能在插入新元素时加以更新，还能维持相对排列顺序。二叉搜索树正好可派上用场。之前是要把元素插入数组，现在则要将元素插入二叉搜索树。`track(int x)`方法的时间复杂度为 $O(\log n)$，其中 n 为树的大小（当然，前提为这棵树是平衡的）。

要找出某个数的秩，可以执行中序遍历，并在访问节点时利用计数器记录数量。目标是找到 x 时，计数器变量将会是小于 x 的元素的数量。在查找 x 期间，只要向左移动，计数器变量就不会变，这是因为右边跳过的所有值都比 x 大。可是当向右移动时，我们跳过了左边的一堆元素。这些元素都比 x 小，因此，必须增加计数器的值，这个值等于左子树的元素个数。

我们不会去计算左子树的大小（效率低），而是在加入新元素时，记录相关信息。接下来，我们将以下面的树为例进一步阐述。在下图中，括号内的数字代表左子树的节点数量（或者，换句话说，该节点的秩与子树的节点数量有关）。



假设我们想知道 24 在上面这棵树中的秩，会先将 24 与根节点 20 比较，发现 24 位于右边。根节点的左子树有 4 个节点，再加上根节点本身，总共有 5 个节点小于 24，因此我们会将计数器变量 `counter` 设为 5。

然后，将 24 与节点 25 进行比较，发现 24 必定位于左边。`counter` 变量的值不会更新，因为我们并未“跳过”任何较小的节点，`counter` 变量的值仍为 5。

接着，将 24 与节点 23 进行比较，发现 24 必定位于右边。`counter` 变量会增加 1（变为 6），

因为 23 没有左边的节点。最后，我们找到 24 并返回 counter 的值，即 6。

这个递归算法如下。

```
int getRank(Node node, int x) {
  if x is node.data, return node.leftSize()
  if x is on left of node, return getRank(node.left, x)
  if x is on right of node, return node.leftSize() + 1 + getRank(node.right, x)
}
```

下面是完整的代码。

```
ankNode root = null;

void track(int number) {
  if (root == null) {
    root = new RankNode(number);
  } else {
    root.insert(number);
  }
}

int getRankOfNumber(int number) {
  return root.getRank(number);
}


public class RankNode {
  public int left_size = 0;
  public RankNode left, right;
  public int data = 0;
  public RankNode(int d) {
    data = d;
  }

  public void insert(int d) {
    if (d <= data) {
    if (left != null) left.insert(d);
      else left = new RankNode(d);
      left_size++;
    } else {
      if (right != null) right.insert(d);
      else right = new RankNode(d);
    }
  }

  public int getRank(int d) {
    if (d == data) {
      return left_size;
    } else if (d < data) {
      if (left == null) return -1;
      else return left.getRank(d);
    } else {
      int right_rank = right == null ? -1 : right.getRank(d);
      if (right_rank == -1) return -1;
      else return left_size + 1 + right_rank;
    }
  }
}
```

track 方法和 getRankOfNumber 方法在平衡树中的运行时间为 $O(\log n)$，在不平衡树中为 $O(n)$。

注意上面的代码是怎么处理 d 不在树里的情况的。我们会检查返回值是否为–1，当发现为–1 时，将它往上返回。实操时这类情况后处理很重要。

第 11 章

数 据 库

问题 11.1 至 11.2 用到了以下数据库模式。

Apartments	
AptID	int
UnitNumber	varchar(10)
BuildingID	int

Buildings	
BuildingID	int
ComplexID	int
BuildingName	varchar(100)
Address	varchar(500)

Requests	
RequestID	int
Status	varchar(100)
AptID	int
Description	varchar(500)

Complexes	
ComplexID	int
ComplexName	varchar(100)

AptTenants	
TenantID	int
AptID	int

Tenants	
TenantID	int
TenantName	varchar(100)

注意，每套公寓可能有多位承租人，而每位承租人可能租住多套公寓。每套公寓隶属于一栋大楼，而每栋大楼属于一个综合体。

11.1 “open”的申请数量

编写 SQL 查询，列出所有建筑物，并取得状态为“Open”的申请数量（Requests 表中 Status 为“Open”的条目）。

题目解法

此题直接将 Requests 和 Apartments 连接起来，就能列出建筑物 ID，并取得 Open 申请的数量。取得这份列表后，再将它与 Buildings 表进行连接。

```
SELECT BuildingName, ISNULL(Count, 0) as 'Count'
FROM Buildings
LEFT JOIN
   (SELECT Apartments.BuildingID, count(*) as 'Count'
    FROM Requests INNER JOIN Apartments
    ON Requests.AptID = Apartments.AptID
    WHERE Requests.Status = 'Open'
    GROUP BY Apartments.BuildingID) ReqCounts
ON ReqCounts.BuildingID = Buildings.BuildingID
```

诸如这种有子查询的查询，务必要经过全面测试，最好先测试查询的内层，然后再测试外层部分。

11.2 关闭所有请求

11 号建筑物正在进行大翻修。编写 SQL 查询，关闭这栋建筑物里所有公寓的入住申请。

题目解法

跟 SELECT 查询一样，UPDATE 查询也可以有 WHERE 子句。要实现这个查询，我们会获取 11 号建筑物里所有公寓的 ID，然后从这些公寓取得入住申请列表。

```
UPDATE Requests
SET Status = 'Closed'
WHERE AptID IN (SELECT AptID FROM Apartments WHERE BuildingID = 11)
```

11.3　画一个实体关系图

有个数据库，里面有公司（companies）、人（people）和在职专业人员（professional），请绘制实体关系图。

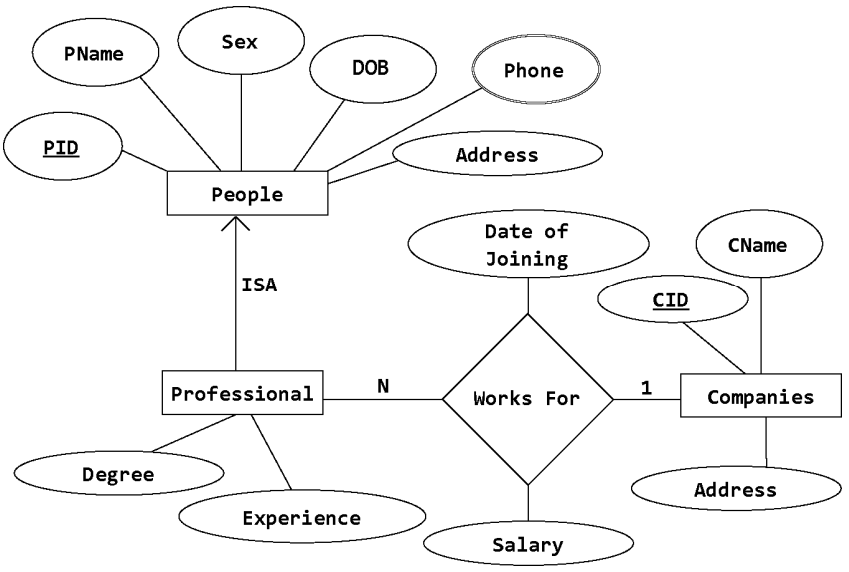
题目解法

在公司（Companies）上班的人（People）称作专业人员（Professional）。因此，People 和 Professional 之间是 ISA（“is a”）关系（或者说 Professional 派生自 People）。

除了从 People 派生的属性，Professional 还有一些附加信息，包括学历（degree）和工作经验（experience）等。

每位 Professional 同一时间只能为一家 Company 工作（也许你可能想验证这一假设），Companies 则可以同时雇佣多位 Professional，因此 Professional 和 Companies 之间是多对一的关系。“Works For”关系可以存放员工的入职时间和薪资等属性。这些属性只有在将 Professional 与 Company 相关联时才会定义。

一个 People 可能拥有多个电话号码，所以 Phone 是个多值属性。



第 12 章

C 和 C++

12.1 虚函数原理

C++虚函数的工作原理是什么?

题目解法

虚函数（virtual function）需要虚函数表（vtable，virtual table）才能实现。如果一个类有函数声明是虚函数，就会生成一个 vtable，存放这个类的虚函数地址。此外，编译器还会在类里加入隐藏的 `vptr` 变量。若子类没有覆写虚函数，该子类的 vtable 就会存放父类的函数地址。调用这个虚函数时，就会通过 vtable 解析函数的地址。在 C++里，动态绑定（dynamic binding）就是通过 vtable 机制实现的。

因此，将子类对象赋值给基类指针时，`vptr` 变量就会指向子类的 vtable。这样一来，就能确保继承关系最末端的子类虚函数会被调用到。

请看以下代码。

```
class Shape {
  public:
  int edge_length;
  virtual int circumference () {
    cout << "Circumference of Base Class\n";
    return 0;
  }
};

class Triangle: public Shape {
  public:
  int circumference () {
    cout<< "Circumference of Triangle Class\n";
    return 3 * edge_length;
  }
};

void main() {
  Shape * x = new Shape();
  x->circumference(); // "Circumference of Base Class"
  Shape *y = new Triangle();
  y->circumference(); // "Circumference of Triangle Class"
}
```

在上述代码中，`circumference` 是 `Shape` 类的虚函数，因此所有继承 `Shape` 类的子类（`Triangle` 等）都为虚函数。在 C++里，非虚函数的调用是在编译期通过静态绑定确定的，虚函数的调用则是在运行期通过动态绑定确定的。

12.2　虚基类

基类的析构函数为何要声明为 virtual?

题目解法

让我们先想想为何会有虚函数，假设有如下代码。

```
class Foo {
 public:
  void f();
};

class Bar : public Foo {
 public:
  void f();
}

Foo * p = new Bar();
p->f();
```

调用 p->f()最后将会调用 Foo::f()，这是因为 p 是指向 Foo 的指针，而 f()不是虚拟的。

为确保 p->f()会调用继承关系最末端的子类的 f()实现，我们需要将 f()声明为虚函数。

现在，回到前面的析构函数。析构函数用于释放内存和资源。Foo 的析构函数若不是虚拟的，那么，即使 p 实际上是 Bar 类型的，还是会调用 Foo 的析构函数。这就是要将析构函数声明为虚拟的原因所在，为了确保正确调用继承关系最末端的子类的析构函数。

12.3　复制节点

编写一种方法，传入参数为指向 Node 结构的指针，返回传入数据结构的完整副本，其中，Node 数据结构含有两个指向其他 Node 的指针。

题目解法

下面的算法将记录一份映射关系，从原先结构中的节点地址对应到新结构中相应的节点。利用该映射关系，在这个结构的深度优先遍历中，就能判断某个节点是不是复制过了。遍历时通常会标记访问过的节点，标记可以有多种形式，不一定要存放在节点里。

综上所述，可以得出一个简单的递归算法。

```
typedef map<Node*, Node*> NodeMap;

Node * copy_recursive(Node * cur, NodeMap & nodeMap) {
  if (cur == NULL) {
    return NULL;
  }

  NodeMap::iterator i = nodeMap.find(cur);
  if (i != nodeMap.end()) {
    // 已访问过这里，返回复制
    return i->second;
  }

  Node * node = new Node;
  nodeMap[cur] = node; // 在遍历链接之前，建立映射关系
  node->ptr1 = copy_recursive(cur->ptr1, nodeMap);
  node->ptr2 = copy_recursive(cur->ptr2, nodeMap);
```

```
  return node;
}

Node * copy_structure(Node * root) {
  NodeMap nodeMap; // 需要一个空的 map
  return copy_recursive(root, nodeMap);
}
```

12.4 智能指针

编写一个智能指针类。智能指针是一种数据类型，一般用模板实现，模拟指针行为的同时还提供自动垃圾回收机制。它会自动记录 SmartPointer<T*>对象的引用计数，一旦 T 类型对象的引用计数为 0，就会释放该对象。

题目解法

智能指针跟普通指针一样，但它借由自动化内存管理保证了安全性，避免了诸如悬挂指针、内存泄漏和分配失败等问题。智能指针必须为给定对象的所有引用维护单一引用计数。

第一次看到这类问题，可能会觉得太难而不知所措，特别是当你并非 C++专家时。此题有个解决之道，分两步走：(1) 以伪码勾勒出做法；(2) 实现具体代码。

按照这种做法，我们需要一个引用计数变量，每新增一个对象的引用，该变量会加 1，移除一个引用则减 1。实现代码与下面的伪码类似。

```
template <class T> class SmartPointer {
  /* 智能指针类需要指向对象本身及引用计数两者的指针。这些都必须是指针，
   * 而不是真实的对象或引用计数值，因为智能指针的目的就在于，
   * 可以跨多个指向某一对象的智能指针，来追踪同一个引用计数 */
  T * obj;
  unsigned * ref_count;
}
```

这个类还需要若干构造函数和一个析构函数，下面先加上这些函数。

```
SmartPointer(T * object) {
  /* 想要设定 T * obj 的值，并将引用计数设为 1 */
}

SmartPointer(SmartPointer<T>& sptr) {
  /* 这个构造函数会新建一个指向已有对象的智能指针。我们需要先设定 obj 和 ref_count，
   * 设为指向 sptr 的 obj 和 ref_count。然后，因为我们新建了一个 obj 的引用，
   * 所以需要增加 ref_count */
}

~SmartPointer(SmartPointer<T> sptr) {
  /* 销毁该对象的引用，减少 ref_count 的值。若 ref_count 为 0，
   * 则释放为存放整数而申请的内存，并销毁对象 */
}
```

还有一种方式也可以创建引用：将一个 SmartPointer 赋值给另一个。处理这种情况需要覆写=操作符，不过这里先略述一二。

```
onSetEquals(SmartPointer<T> ptr1, SmartPointer<T> ptr2) {
  /* 若 ptr1 已有值，减小其引用计数。然后，复制指向 obj 和 ref_count 的指针。
   * 最后，因为创建了新引用，所以需要增加 ref_count 的值 */
}
```

虽未填入 C++语法，但大体框架已完成。接下来，要完成所有代码，只需填补好细节即可。

```
template <class T> class SmartPointer {
 public:
  SmartPointer(T * ptr) {
    ref = ptr;
    ref_count = (unsigned*)malloc(sizeof(unsigned));
    *ref_count = 1;
  }

  SmartPointer(SmartPointer<T> & sptr) {
    ref = sptr.ref;
    ref_count = sptr.ref_count;
    ++(*ref_count);
  }

  /* 覆写=运算符，这样才能将一个旧的智能指针赋值给另一指针，
   * 旧的引用计数减一，新的智能指针的引用计数则加一 */
  SmartPointer<T> & operator=(SmartPointer<T> & sptr) {
    if (this == &sptr) return *this;

    /* 若已赋值为某个对象，则移除引用 */
    if (*ref_count > 0) {
       remove();
    }

    ref = sptr.ref;
    ref_count = sptr.ref_count;
    ++(*ref_count);
    return *this;
  }

  ~SmartPointer() {
    remove(); // 移除一个对象引用
  }

  T getValue() {
    return *ref;
  }

protected:
  void remove() {
    --(*ref_count);
    if (*ref_count == 0) {
       delete ref;
       free(ref_count);
       ref = NULL;
       ref_count = NULL;
    }
  }

  T * ref;
  unsigned * ref_count;
};
```

此题的代码复杂难懂，错漏在所难免，面试官也不会强求代码写得完美无缺。

第 13 章

Java

13.1 异常处理中的返回

在 Java 中，若在 try-catch-finally 的 try 语句块中插入 return 语句，finally 语句块是否还会执行?

题目解法

会执行。当退出 try 语句块时，finally 语句块将会执行。即使我们试图从 try 语句块（通过 return 语句、continue 语句、break 语句或任意异常）里跳出，finally 语句块仍将得以执行。

注意，有些情况下 finally 语句块将不会执行，比如下列情形。

- 如果虚拟机在 try/catch 语句块执行期间退出。
- 如果执行 try/catch 语句块的线程被杀死终止了。

13.2 lambda 随机数

使用 lambda 表达式写一种名为 getRandomSubset(List<Integer> list)的方法，返回值类型为 List<Integer>，返回一个任意大小的随机子集，所有子集（包括空子集）选中的概率都一样。

题目解法

先从 0 至 N 中选取子集的数量，然后生成此数量的随机子集，从而解决这个问题。这个方法值得一试。

但会产生如下两个问题。

(1) 我们必须加权这些概率。如果 $N>1$，那么容量为 $N/2$ 的子集比容量为 N 的子集（其中总是只有一个）更多。

(2) 实际上生成受限大小（特别是 10）的子集比生成任意大小的子集更困难。

与其基于容量生成子集，不如考虑基于元素的情况。其实，该题要求使用 lambda 表达式也表明，我们应该考虑通过元素进行某种迭代或处理。

想象一下，我们正在迭代{1, 2, 3}生成一个子集，1 应该在其中吗?

我们有两种选择：是或否。我们需要根据子集中包含 1 的百分比来衡量“是”与“否”的概率。那么，包含 1 的子集占比多少?

对于任何特定的元素,包含某个元素的子集和不包含该元素的子集数量一样多。考虑下列情况。

```
{}          {1}
{2}         {1, 2}
{3}         {1, 3}
{2, 3}      {1, 2, 3}
```

请注意左边的子集和右边的子集之间的差异在于是否存在 1。左右两边肯定具有相同数量的子集，因为我们只需添加一个元素即可将其从一个转换为另一个。

这意味着我们可以通过遍历列表和抛出一枚硬币（即决定 50 / 50 的概率）来生成一个随机子集，以选择每个元素是否在其中。不用 lambda 表达式，我们可以写出如下所示的代码。

```
List<Integer> getRandomSubset(List<Integer> list) {
  List<Integer> subset = new ArrayList<Integer>();
  Random random = new Random();
  for (int item : list) {
    /* 翻转硬币 */
    if (random.nextBoolean()) {
      subset.add(item);
    }
  }
  return subset;
}
```

要用 lambda 实现这个方法，我们可以执行如下操作。

```
List<Integer> getRandomSubset(List<Integer> list) {
  Random random = new Random();
  List<Integer> subset = list.stream().filter(
    k -> { return random.nextBoolean(); /* 翻转硬币 */
  }).collect(Collectors.toList());
  return subset;
}
```

该实现方法的一大益处是，现在我们可以在其他地方使用 `flipCoin` 谓词了。

```
Random random = new Random();
Predicate<Object> flipCoin = o -> {
  return random.nextBoolean();
};

List<Integer> getRandomSubset(List<Integer> list) {
  List<Integer> subset = list.stream().filter(flipCoin).
    collect(Collectors.toList());
  return subset;
}
```

第 14 章

线程与锁

14.1 哲学家用餐

在著名的哲学家用餐问题中，一群哲学家围坐在圆桌周围，每两位哲学家之间有一根筷子。每位哲学家需要两根筷子才能用餐，并且一定会先拿起左手边的筷子，然后才会去拿右手边的筷子。如果所有哲学家在同一时间拿起左手边的筷子，就有可能造成死锁。请使用线程和锁，编写代码模拟哲学家用餐问题，避免出现死锁。

题目解法

首先，先不管死锁，让我们写些代码简单模拟哲学家用餐问题。具体实现时，从 `Thread` 派生 `Philosopher`，拿起 `Chopstick` 时会调用 `lock.lock()`，放下时则调用 `lock.unlock()`。

```
1  class Chopstick {
2    private Lock lock;
3
4    public Chopstick() {
5      lock = new ReentrantLock();
6    }
7
8    public void pickUp() {
9      void lock.lock();
10   }
11
12   public void putDown() {
13     lock.unlock();
14   }
15 }
16
17 class Philosopher extends Thread {
18   private int bites = 10;
19   private Chopstick left, right;
20
21   public Philosopher(Chopstick left, Chopstick right) {
22     this.left = left;
23     this.right = right;
24   }
25
26   public void eat() {
27     pickUp();
28     chew();
29     putDown();
30   }
31
32   public void pickUp() {
33     left.pickUp();
```

```
    right.pickUp();
  }

  public void chew() { }

  public void putDown() {
    right.putDown();
    left.putDown();
  }

  public void run() {
    for (int i = 0; i < bites; i++) {
      eat();
    }
  }
}
```

如果所有哲学家都拿起左手边的一根筷子，并都等着拿右手边的另一根筷子，运行上面的代码就可能造成死锁。

解法 1：全部或无

为了防止发生死锁，我们的实现可以采用如下策略：如有哲学家拿不到右手边的筷子，就让他放下已拿到的左手边的筷子。

```
public class Chopstick {
  /* 同前 */

  public boolean pickUp() {
    return lock.tryLock();
  }
}

public class Philosopher extends Thread {
  /* 同前 */

  public void eat() {
    if (pickUp()) {
      chew();
      putDown();
    }
  }

  public boolean pickUp() {
    /* 试着拿起筷子 */
    if (!left.pickUp()) {
      return false;
    }
    if (!right.pickUp()) {
      left.putDown();
      return false;
    }
    return true;
  }
}
```

在上面的代码中，要确保拿不到右手边的筷子时就要放下左手边的筷子。如果手上根本没有筷子，就不该调用 `putDown()`。

一个问题是，如果所有的哲学家都完全同步，他们可以同时拿起左手边筷子，无法拿起右手边筷子，然后把筷子放回左手边，会不断重复这个过程。

解法 2：区分筷子优先级

或者我们可以用数字 0 到 $N-1$ 来标记筷子。每位哲学家首先尝试拿起较低编号的筷子。这基本上意味着每位哲学家都会在选择右手边筷子之前选择左手边筷子（假设这是你标记它的方式），除了最后一位哲学家以相反的方式做这件事。 这将打破这个循环。

```
1  public class Philosopher extends Thread {
2    private int bites = 10;
3    private Chopstick lower, higher;
4    private int index;
5    public Philosopher(int i, Chopstick left, Chopstick right) {
6      index = i;
7      if (left.getNumber() < right.getNumber()) {
8        this.lower = left;
9        this.higher = right;
10     } else {
11       this.lower = right;
12       this.higher = left;
13     }
14   }
15
16   public void eat() {
17     pickUp();
18     chew();
19     putDown();
20   }
21
22   public void pickUp() {
23     lower.pickUp();
24     higher.pickUp();
25   }
26
27   public void chew() { ... }
28
29   public void putDown() {
30     higher.putDown();
31     lower.putDown();
32   }
33
34   public void run() {
35     for (int i = 0; i < bites; i++) {
36       eat();
37     }
38   }
39 }
40
41 public class Chopstick {
42   private Lock lock;
43   private int number;
44
45   public Chopstick(int n) {
46     lock = new ReentrantLock();
47     this.number = n;
48   }
49
50   public void pickUp() {
51     lock.lock();
52   }
53
54   public void putDown() {
55     lock.unlock();
```

```
  }

  public int getNumber() {
    return number;
  }
}
```

有了这个解决方案，一位哲学家就不能拿着较大编号的筷子而不拿着那个较小编号的筷子。这也就阻止了循环的发生，因为循环意味着更高优先级的筷子会“指向”优先级更低的筷子。

14.2　同步方法

给定一个类，内含同步方法 A 和普通方法 B。在同一个程序实例中，有两个线程，能否同时执行 A？两者能否同时执行 A 和 B？

题目解法

在方法前加上关键字 synchronized，即可保证两个线程无法同时执行某个对象的同步方法。

因此，第一个子问题的答案要视具体情况而定。如果两个线程拥有该对象的同一实例，那么，答案就是否定的，它们不能同时执行方法 A。不过，要是这两个线程拥有该对象的不同实例，就能同时执行方法 A。

在概念上，你可以从“锁”的角度来考虑答案。同步方法会对所属对象特定实例的所有同步方法上锁，从而阻止任何其他线程执行那个实例的同步方法。

第二个子问题问的是，thread2 在执行非同步方法 B 时，thread1 能否执行同步方法 A。既然 B 不是同步方法，在 thread2 执行方法 B 时，也就无从阻止 thread1 执行方法。不管 thread1 和 thread2 是否拥有该对象的同一实例，这一点都成立。

说到底，此题强调的关键概念是，那个对象的每个实例只能执行一个同步方法。其他线程可以执行该实例的非同步方法，或者它们可以执行该对象不同实例的任意方法。

第 15 章

测　试

15.1　找错

找出以下代码中的错误（可能不止一处）。

```
unsigned int i;
for (i = 100; i >= 0; --i)
  printf("%d\n", i);
```

题目解法

这段代码有两处错误。

首先，根据定义，unsigned int 类型的变量一定会大于或等于 0。因此，for 循环的测试条件一直为真，将陷入无限循环。

要打印 100 到 1 之间的所有整数，正确的做法是测试 i > 0。如果真的想打印 0，可以在 for 循环之后加一条 printf 语句。

```
1  unsigned int i;
2  for (i = 100; i > 0; --i)
3    printf("%d\n", i);
```

另一个需要修正的地方是用%u 代替%d，因为这里打印的是 unsigned int 型变量。

```
1  unsigned int i;
2  for (i = 100; i > 0; --i)
3    printf("%u\n", i);
```

现在，这段代码会正确地打印 100 到 1 的整数序列（按降序排列）。

15.2　测试国际象棋

有个国际象棋游戏程序使用了 boolean canMoveTo(int x, int y)方法，这个方法是 Piece 类的一部分，可以判断某个棋子能否移动到位置 (*x*, *y*)。请阐述你会如何测试该方法。

题目解法

这个问题主要涉及两大类测试：极限情况测试（确保有错误输入时程序不会崩溃）和一般情况测试。我们先从第一类测试着手。

测试类型 1：极限情况测试

确保程序会妥善处理错误或异常输入，这意味着要检查以下情况。

- 测试 x 和 y 为负数的情况。
- 测试 x 大于棋盘宽度的情况。
- 测试 y 大于棋盘高度的情况。

- ❑ 测试一个满是棋子的棋盘。
- ❑ 测试一个空的或接近空的棋盘。
- ❑ 测试白子远多于黑子的情况。
- ❑ 测试黑子远多于白子的情况。

对于上面的错误情况，我们应该询问面试官，是要返回 false 还是抛出异常，然后有针对性地进行测试。

测试类型 2：一般情况测试

一般情况测试的涉及面要广泛得多。理想的做法是测试每一种可能的棋盘布局，但是棋局实在太多了。不过，我们还是可以合理地执行测试，尽量涵盖不同的棋局。

国际象棋一共有 6 种棋子，我们可以测试每一种棋子，在所有可能的方向上，向其他所有棋子移动的情况，大致如下面的代码所示。

```
对每一种棋子 a：
  对其他每一种棋子 b（6 种及空白）
    对每一个方向 d
      创建有 a 的棋盘
      将 b 放在方向 d 上
      试着移动——检查返回值
```

此题的关键在于能认识到我们不可能测试每一种可能的场景，即使有心也无力办到。因此，好钢要用在刀刃上，我们必须专攻最重要的部分。

15.3　测试一支笔

如何测试一支笔？

题目解法

解出此题的关键在于能理解限制条件以及解题时能做到有条不紊。

为了理解有哪些限制条件，你应该抛出一系列诸如“谁、什么、何地、何时、如何以及为什么”（只要是与该问题相关的就行）之类的问题。一个好的测试人员在着手测试之前，会先准确了解自己要测试的是什么。

让我们通过下面的模拟对话来理解上述技巧。

面试官：你会如何测试一支笔？

求职者：我想先了解一下这支笔。谁会使用这支笔？

面试官：可能是小孩。

求职者：嗯，有意思。他们会用这支笔做什么？写字、画画还是干别的？

面试官：画画。

求职者：好的，谢谢。画在哪里呢？纸张、布料还是墙壁上？

面试官：画在布料上。

求职者：那么，这支笔的笔头是什么样的？签字笔还是圆珠笔？要洗得掉的，还是洗不掉的？

面试官：要求洗得掉。

基于以上问题，你可以得出如下结论。

求职者：好的。综上所述，我理解如下：这支笔主要面向 5 至 10 岁的小孩，为签字笔头，有红、绿、蓝、黑四色，用来画画。画在布料上并且要求洗得掉。我的理解对吗？

此时，求职者面对的问题与乍看上去的问题截然不同，这种情况并不少见。事实上，许多面试官会故意抛出一个看似再清楚不过的问题（谁不知道笔是什么呢！），实则在考查你能否明察秋毫，找出问题的关键所在。他们相信用户也会这么做，但用户多半是无意之举。

至此，你已经知道自己要测试的是什么，接下来该提出测试计划了。这里的关键是**结构**。

想想测试对象或问题会涉及哪些方面，并以此为基础展开测试。这个问题可能会涉及以下几个方面。

- **事实核查**。核实这是一支签字笔以及墨水颜色为要求的四种颜色之一。
- **预期用途**。绘图。这支笔在布料上画得出来吗？
- **预期用途**。水洗。画在布料上的墨迹洗得掉吗（哪怕已经过了一段时间）？是用热水、温水还是冷水才能洗掉？
- **安全性**。这支笔对小孩是否安全（无毒）？
- **非预期用途**。小孩还会怎么使用这支笔？他们可能在其他物体表面上涂鸦，因此，还需检查他们的行为是否正确。他们还可能踩踏、乱扔这支笔，等等。你需要确认这支笔是否经受得住这些使用条件。

记住，对于任何测试问题，你都必须测试预期和非预期的场景。因为用户并不一定会按照你预想的方式使用产品。

15.4 测试 ATM

在一个分布式银行系统中，该如何测试一台自动柜员机（ATM）？

题目解法

对于这个问题，第一要务是厘清若干假设条件，请提出以下问题。

- 谁会使用 ATM 机？答案可能是“任何人”，或是“盲人”，或任意其他可能的答案。
- 他们会用 ATM 机来做什么？答案可能是“取款”、“转账”、“查询余额”，等等。
- 我们有什么工具来测试呢？我们可以查看代码吗？还是只能访问 ATM 机？

切记：好的测试人员会先确定自己要测试的是什么。

一旦了解系统是什么样的，我们就会想着将问题分解成可测试的子部分，如下所示。

- 登录；
- 取款；
- 存款；
- 查询余额；
- 转账。

我们可能要搭配使用手动和自动测试。

手动测试会检查上述步骤的每一个环节，确保涵盖所有错误情况（余额不足、新开账户、不存在的账户，等等）。

自动测试稍微复杂一些。我们会希望自动处理上述所有标准流程，还要找一些较具体的问题，比如竞争条件。理想情况下，我们会设法建立一套有假账户的封闭系统，以确保即使有人从不同地点快速取款和存款，也不会多得不应得的钱或者损失应得的钱。

最重要的是，我们必须优先考虑安全性和可靠性。客户的账户无时无刻都要处于被保护的状态，我们必须确保账目得到正确处理。没有人会希望自己的钱不翼而飞。优秀的测试人员深谙整个系统里哪些事项是最重要的。

第 16 章

中等难题

16.1 单词频率

设计一个方法，找出任意指定单词在一本书中的出现频率。如果我们多次使用此方法，应该怎么办?

题目解法

让我们从简单的用例开始。

解法 1：单次查询

在这种情况下,我们会直接逐字逐句地扫描整本书,数一数某个单词出现的次数,用时 $O(n)$。可以确定这是最短用时，因为不管怎么样，我们必须查看过书中的每个单词。

```
int getFrequency(String[] book, String word) {
  word = word.trim().toLowerCase();
  int count = 0;
  for (String w : book) {
    if (w.trim().toLowerCase().equals(word)) {
      count++;
    }
  }
  return count;
}
```

我们同时将字符串转化为了小写字符，并对其两端的空白字符进行了移除。你可以与面试官讨论是否有必要（是否应该）进行如上操作。

解法 2：重复查询

如果是要重复执行查询操作，那么，或许值得我们多花些时间，多分配内存，对全书进行预处理。我们可以构造一个散列表，将单词映射到该单词的出现频率，这么一来，任意单词的频率都能在 $O(1)$时间内找到。具体实现代码如下。

```
HashMap<String, Integer> setupDictionary(String[] book) {
  HashMap<String, Integer> table =
    new HashMap<String, Integer>();
  for (String word : book) {
    word = word.toLowerCase();
    if (word.trim() != "") {
      if (!table.containsKey(word)) {
        table.put(word, 0);
      }
      table.put(word, table.get(word) + 1);
    }
  }
```

```
  return table;
}

int getFrequency(HashMap<String, Integer> table, String word) {
  if (table == null || word == null) return -1;
  word = word.toLowerCase();
  if (table.containsKey(word)) {
    return table.get(word);
  }
  return 0;
}
```

注意，相对而言，这类问题还是比较容易的。因此，面试官会更看重你的心思有多缜密，有没有检查错误条件？

16.2 井字游戏

设计一个算法，判断玩家是否赢了井字游戏。

题目解法

乍一看，可能会觉得此题很简单，不就是直接检查井字棋盘，这会有多难呢？细一想，此题还是有点复杂的，而且没有唯一的“完美”答案。你的喜好不同，最佳解法也会不一样。

解决此题，有几个重要的设计决策需要考虑。

(1) hasWon 只会调用一次还是很多次（比如，放在网站上的井字游戏）？如果答案是后者，我们可能会增加一些预处理，以优化 hasWon 的运行时间。

(2) 我们知道最后一步吗？

(3) 井字游戏通常是 3 × 3 棋盘。我们只是针对 3 × 3 大小的棋盘进行设计，还是要实现一个 $N \times N$ 的解法？

(4) 对于程序大小、执行速度和代码清晰度，一般如何区分它们的优先级呢？记住：最高效的代码不一定是最好的。代码是否易于理解且易维护也很重要。

解法 1：如果 hasWon 会被多次调用

总共只有 3^9，大约 20 000 种井字游戏棋盘（假设为 3 × 3 的棋盘）。因此，用一个 int 就能表示，其中每个数位代表棋盘中的一格（0 为空、1 为红、2 为蓝）。我们会事先设定好一个散列表或数组，将所有可能的棋盘作为键，值则代表谁赢了。这么一来，hasWon 函数就很简单了。

```
Piece hasWon(int board) {
  return winnerHashtable[board];
}
```

要将一个棋盘（以字符数组表示）转成一个 int，可以运用“3 进位”表示法，每个棋盘可表示为 $3^0v_0 + 3^1v_1 + 3^2v_2 + \cdots + 3^8v_8$，若格子为空则 v_i 为 0，格子为蓝色则 v_i 为 1，格子为红色则 v_i 为 2。

```
enum Piece { Empty, Red, Blue };

int convertBoardToInt(Piece[][] board) {
  int sum = 0;
  for (int i = 0; i < board.length; i++) {
    for (int j = 0; j < board[i].length; j++) {
      /* 每个枚举类型的值都有整型数值与之对应，我们可以直接使用 */
      int value = board[i][j].ordinal();
```

```
        sum = sum * 3 + value;
      }
    }
    return sum;
  }
```

至此，要判断谁是赢家，只需查询散列表即可。

当然，如果每次判断谁赢了都要将棋盘转成这种格式，那么跟其他解法相比，其实并没有节省多少时间。但是，如果一开始就以这种格式存储棋盘，那么查询操作将会非常高效。

解法 2：如果我们知道最后一步

如果我们知道最后一步（并且至此为止都在不断地检查是否有人胜出），那么只需要检查与最后一步所走的位置相重叠的行、列和对角线即可。

```
Piece hasWon(Piece[][] board, int row, int column) {
  if (board.length != board[0].length) return Piece.Empty;

  Piece piece = board[row][column];

  if (piece == Piece.Empty) return Piece.Empty;

  if (hasWonRow(board, row) || hasWonColumn(board, column)) {
    return piece;
  }

  if (row == column && hasWonDiagonal(board, 1)) {
    return piece;
  }

  if (row == (board.length - column - 1) && hasWonDiagonal(board, -1)) {
    return piece;
  }

  return Piece.Empty;
}

boolean hasWonRow(Piece[][] board, int row) {
  for (int c = 1; c < board[row].length; c++) {
    if (board[row][c] != board[row][0]) {
      return false;
    }
  }
  return true;
}

boolean hasWonColumn(Piece[][] board, int column) {
  for (int r = 1; r < board.length; r++) {
    if (board[r][column] != board[0][column]) {
      return false;
    }
  }
  return true;
}

boolean hasWonDiagonal(Piece[][] board, int direction) {
  int row = 0;
  int column = direction == 1 ? 0 : board.length - 1;
  Piece first = board[0][column];
  for (int i = 0; i < board.length; i++) {
```

```
46      if (board[row][column] != first) {
47        return false;
48      }
49      row += 1;
50      column += direction;
51    }
52    return true;
53  }
```

实际上有一种方法可以清理上述代码中一些重复的部分。我们在后面的函数中会看到该方法。

解法 3：专为 3 × 3 棋盘设计

如果只想为 3 × 3 棋盘设计一种解法，代码就会比较简短且简单。复杂的地方只剩下如何写得清晰而有条理，并且不要写出太多重复代码。

下面的代码对每一行、每一列和每条对角线都进行了检查，以便确认是否有人胜出。

```
1   Piece hasWon(Piece[][] board) {
2     for (int i = 0; i < board.length; i++) {
3       /* 检查行 */
4       if (hasWinner(board[i][0], board[i][1], board[i][2])) {
5         return board[i][0];
6       }
7
8       /* 检查列 */
9       if (hasWinner(board[0][i], board[1][i], board[2][i])) {
10        return board[0][i];
11      }
12    }
13
14    /* 检查对角线 */
15    if (hasWinner(board[0][0], board[1][1], board[2][2])) {
16      return board[0][0];
17    }
18
19    if (hasWinner(board[0][2], board[1][1], board[2][0])) {
20      return board[0][2];
21    }
22
23    return Piece.Empty;
24  }
25
26  boolean hasWinner(Piece p1, Piece p2, Piece p3) {
27    if (p1 == Piece.Empty) {
28      return false;
29    }
30    return p1 == p2 && p2 == p3;
31  }
```

该算法可行，这是因为理解起来相对容易。问题是，代码中的值是以硬编码的方式实现的，很容易会不小心写错索引的值。

另外，该算法也不易于被扩展到 $N \times N$ 的棋盘上。

解法 4：面向 $N \times N$ 棋盘进行设计

有很多种方法在 $N \times N$ 的棋盘上实现该算法。

- *嵌套 for 循环法*

最明显的方法是通过几层嵌套的 `for` 循环实现该算法。

```
Piece hasWon(Piece[][] board) {
  int size = board.length;
  if (board[0].length != size) return Piece.Empty;
  Piece first;

  /* 检查行 */
  for (int i = 0; i < size; i++) {
    first = board[i][0];
    if (first == Piece.Empty) continue;
    for (int j = 1; j < size; j++) {
      if (board[i][j] != first) {
        break;
      } else if (j == size - 1) { // 最后一个元素
        return first;
      }
    }
  }

  /* 检查列 */
  for (int i = 0; i < size; i++) {
    first = board[0][i];
    if (first == Piece.Empty) continue;
    for (int j = 1; j < size; j++) {
      if (board[j][i] != first) {
      break;
    } else if (j == size - 1) { // 最后一个元素
      return first;
    }
  }
}

  /* 检查对角线 */
  first = board[0][0];
  if (first != Piece.Empty) {
    for (int i = 1; i < size; i++) {
      if (board[i][i] != first) {
        break;
      } else if (i == size - 1) { // 最后一个元素
        return first;
      }
    }
  }

  first = board[0][size - 1];
  if (first != Piece.Empty) {
    for (int i = 1; i < size; i++) {
      if (board[i][size - i - 1] != first) {
        break;
      } else if (i == size - 1) { // 最后一个元素
        return first;
      }
    }
  }

  return Piece.Empty;
}
```

退一步讲，该代码实现得非常粗糙。我们基本上每次都在做相同的事情，应该能够找到重用代码的方式。

● 增加（increment）和减少（decrement）函数法

一种进行代码重用的较好方式是：将值传入一个函数中，该函数可以对行和列进行增加或减少。这样一来，hasWon 函数只需要起始位置以及行和列的增量即可。

```
class Check {
  public int row, column;
  private int rowIncrement, columnIncrement;
  public Check(int row, int column, int rowI, int colI) {
    this.row = row;
    this.column = column;
    this.rowIncrement = rowI;
    this.columnIncrement = colI;
  }

  public void increment() {
    row += rowIncrement;
    column += columnIncrement;
  }

  public boolean inBounds(int size) {
    return row >= 0 && column >= 0 && row < size && column < size;
  }
}

Piece hasWon(Piece[][] board) {
  if (board.length != board[0].length) return Piece.Empty;
  int size = board.length;

  /* 创建一个链表 */
  ArrayList<Check> instructions = new ArrayList<Check>();
  for (int i = 0; i < board.length; i++) {
    instructions.add(new Check(0, i, 1, 0));
    instructions.add(new Check(i, 0, 0, 1));
  }
  instructions.add(new Check(0, 0, 1, 1));
  instructions.add(new Check(0, size - 1, 1, -1));

  /* 检查每个元素 */
  for (Check instr : instructions) {
    Piece winner = hasWon(board, instr);
    if (winner != Piece.Empty) {
        return winner;
    }
  }
  return Piece.Empty;
}

Piece hasWon(Piece[][] board, Check instr) {
  Piece first = board[instr.row][instr.column];
  while (instr.inBounds(board.length)) {
    if (board[instr.row][instr.column] != first) {
      return Piece.Empty;
    }
    instr.increment();
  }
  return first;
}
```

Check 函数本质上承担了迭代器的任务。

- 迭代器法

当然，另外一种方式是创建一个真正的迭代器。

```
Piece hasWon(Piece[][] board) {
  if (board.length != board[0].length) return Piece.Empty;
  int size = board.length;

  ArrayList<PositionIterator> instructions = new ArrayList<PositionIterator>();
  for (int i = 0; i < board.length; i++) {
    instructions.add(new PositionIterator(new Position(0, i), 1, 0, size));
    instructions.add(new PositionIterator(new Position(i, 0), 0, 1, size));
  }
  instructions.add(new PositionIterator(new Position(0, 0), 1, 1, size));
  instructions.add(new PositionIterator(new Position(0, size - 1), 1, -1, size));

  for (PositionIterator iterator : instructions) {
    Piece winner = hasWon(board, iterator);
    if (winner != Piece.Empty) {
      return winner;
    }
  }
  return Piece.Empty;
}

Piece hasWon(Piece[][] board, PositionIterator iterator) {
  Position firstPosition = iterator.next();
  Piece first = board[firstPosition.row][firstPosition.column];
  while (iterator.hasNext()) {
    Position position = iterator.next();
    if (board[position.row][position.column] != first) {
      return Piece.Empty;
    }
  }
  return first;
}

class PositionIterator implements Iterator<Position> {
  private int rowIncrement, colIncrement, size;
  private Position current;

  public PositionIterator(Position p, int rowIncrement,
                          int colIncrement, int size) {
    this.rowIncrement = rowIncrement;
    this.colIncrement = colIncrement;
    this.size = size;
    current = new Position(p.row - rowIncrement, p.column - colIncrement);
  }

  @Override
  public boolean hasNext() {
    return current.row + rowIncrement < size &&
           current.column + colIncrement < size;
  }

  @Override
  public Position next() {
    current = new Position(current.row + rowIncrement,
                           current.column + colIncrement);
    return current;
  }
}
```

```
59
60 public class Position {
61   public int row, column;
62   public Position(int row, int column) {
63     this.row = row;
64     this.column = column;
65   }
66 }
```

写出上述所有方法对于此题来说似乎有些大材小用，但是有必要和面试官讨论一下这些选项。该题目的重点在于考查你对于代码整洁性和可维护性的理解。

16.3 阶乘尾数

设计一个算法，算出 *n* 阶乘有多少个尾随零。

题目解法

简单的做法是先算出阶乘，然后不断地除以 10，数一数有几个尾随零（trailing zero）。但这种做法的问题是，使用 `int` 很快就会越界。为了避开这个限制，我们可以从数学上来分析这个问题。

下面以阶乘 19!为例进行说明。

19! = 1 × 2 × 3 × 4 × 5 × 6 × 7 × 8 × 9 × 10 × 11 × 12 × 13 × 14 × 15 × 16 × 17 × 18 × 19

10 倍数就会形成尾随零，而 10 倍数又可分解为一组组 5 倍数和 2 倍数。

例如，在 19!中，下列几项会形成尾随零。

19! = 2 × … × 5 ×… × 10 × … × 15 × 16 × …

因此，为了算出尾随零的数量，只需计算有几对 5 和 2 倍数。不过，2 倍数始终要比 5 倍数多，最后只要数出 5 倍数就可以了。

这里有个陷阱，就是 15 只能算一个 5 倍数（因此会形成一个尾随零），而 25 算两个 5 倍数（25 = 5 × 5）。

编写代码时，相关代码有两种写法。

第一种写法是迭代访问所有 2 到 *n* 的数字，计算每个数字中有几个 5。

```
1  /* 如果该数字是 5 的幂，返回其幂次。
2   * 例如：5 返回 1，25 返回 2 */
3  int factorsOf5(int i) {
4    int count = 0;
5    while (i % 5 == 0) {
6      count++;
7      i /= 5;
8    }
9    return count;
10 }
11
12 int countFactZeros(int num) {
13   int count = 0;
14   for (int i = 2; i <= num; i++) {
15     count += factorsOf5(i);
16   }
17   return count;
18 }
```

更高效的方法是：直接数一数 5 的因数。采用这种做法，我们会先数一数 1 到 *n* 之间有几个 5 的倍数（数量为 *n*/5），然后数一数 25 的倍数有几个（*n*/25），接着是 125，以此类推。

要算出 n 中有几个 m 的倍数，直接将 n 除以 m 即可。

```
int countFactZeros(int num) {
  int count = 0;
  if (num < 0) {
    return -1;
  }
  for (int i = 5; num / i > 0; i *= 5) {
    count += num / i;
  }
  return count;
}
```

解此类题只要思考一下到底有哪些条件会形成尾随零，就能得到解法。你必须从一开始就透彻地理解相关规则才能正确地实现出来。

16.4　最大数值

编写一个方法，找出两个数字中最大的那一个。不得使用 `if-else` 或其他比较运算符。

题目解法

`max` 函数的常见实现方法是检查 $a-b$ 的正负号。但这里不能使用比较运算符检查正负情况，不过我们可以使用乘法。

假定 k 代表 $a-b$ 的正负号，如果 $a-b \geqslant 0$，则 k 为 1，否则 k 为 0。令 q 为 k 的反数。

那么，我们可以实现如下代码。

```
/* 将 1 翻转为 0，0 翻转为 1  */
int flip(int bit) {
  return 1^bit;
}

/* 如果是正数，就返回 1；如果是负数，则返回 0 */
int sign(int a) {
  return flip((a >> 31) & 0x1);
}

int getMaxNaive(int a, int b) {
  int k = sign(a - b);
  int q = flip(k);
  return a * k + b * q;
}
```

这段代码看似可行，实则不济。要是 $a-b$ 溢出，这段代码就行不通。例如，假设 a 为 `INT_MAX - 2`，b 为–15。此时，$a-b$ 将大于 `INT_MAX` 并且会溢出，最终变为负值。

运用同样的方法，我们可以实现此题的解法，目标是当 $a>b$ 时维持 k 为 1 的条件。为此，我们需要使用更为复杂的逻辑方法。

$a-b$ 什么时候会溢出呢？它只会在 a 为正且 b 为负时溢出，或者反过来也有可能。专门检测溢出条件可能比较困难，不过，我们可以检测 a 和 b 何时会有不同的正负号。注意，如果 a 和 b 的正负号不同，就让 k 等于 `sign(a)`。

具体逻辑方法如下。

```
if a and b have different signs:
  // 如果 a > 0，则 b < 0，且 k = 1
  // 如果 a < 0，则 b > 0，且 k = 0
```

```
    // 所以，无论如何，k = sign(a)
    let k = sign(a)
  else
    let k = sign(a - b) // 不可能出现溢出
```

上述逻辑方法的实现代码如下，其中使用了乘法而不是 `if` 语句。

```
int getMax(int a, int b) {
  int c = a - b;

  int sa = sign(a); // 如果 a ≥ 0 则返回 1，否则返回 0
  int sb = sign(b); // 如果 b ≥ 0 则返回 1，否则返回 0
  int sc = sign(c); // 取决于 a-b 是否溢出

  /*目标：定义 k，如果 a > b 则为 1，如果 a < b 则为 0。如果 a = b，k 为何值并不重要 */

  // 如果 a 和 b 有不同的符号，则 k = sign(a)
  int use_sign_of_a = sa ^ sb;

  // 如果 a 和 b 有 x 相同的符号，则 k = sign(a-b)
  int use_sign_of_c = flip(sa ^ sb);

  int k = use_sign_of_a * sa + use_sign_of_c * sc;
  int q = flip(k); // k 的相反值

  return a * k + b * q;
}
```

注意，为清晰起见，我们将代码拆分成多个方法和变量。很显然，这不是最紧凑或最有效的写法，但这么写代码要清晰许多。

16.5 跳水板

你正在使用一堆木板建造跳水板。有两种类型的木板，其中一种长度较短（长度记为 `shorter`），一种长度较长（长度记为 `longer`）。你必须正好使用 K 块木板。编写一个方法，生成跳水板所有可能的长度。

题目解法

该题其中一种解法是仔细分析建造跳水板时的决策过程。该过程将会引导我们使用递归算法。

1. 递归解法

使用递归解法时，我们可以想象自己正在建造一个跳水板。我们需要进行 K 次决策，每次决策时需要决定下一步使用哪块木板。当我们使用了 K 块木板之后，即完成了跳水板的建造，随即可以将其加入到列表中。（假设从未建造过相同长度的跳水板。）

基于此，我们可以编写递归代码。请注意，我们不需要记录木板的顺序，只需了解当前跳水板的长度和剩余可用的木板数量。

```
HashSet<Integer> allLengths(int k, int shorter, int longer) {
  HashSet<Integer> lengths = new HashSet<Integer>();
  getAllLengths(k, 0, shorter, longer, lengths);
  return lengths;
}

void getAllLengths(int k, int total, int shorter, int longer,
                   HashSet<Integer> lengths) {
```

```
9     if (k == 0) {
10      lengths.add(total);
11      return;
12    }
13    getAllLengths(k - 1, total + shorter, shorter, longer, lengths);
14    getAllLengths(k - 1, total + longer, shorter, longer, lengths);
15  }
```

我们将所有的长度都加入到了散列集合中。这样，就可以自动防止增加重复长度的木板了。

该算法会花费 $O(2^k)$的时间，这是因为每次递归调用时，都有两个不同的选择，而我们总共完成 K 次递归。

2. 记忆化解法

和许多递归算法（特别是具有指数型时间复杂度的递归算法）一样，我们可以通过记忆化的方式优化该算法（该优化即动态编程的形式之一）。

请注意，很多递归调用本质上是相同的。例如，“先选择木板 1 再选择木板 2”完全等同于“先选择木板 2 再选择木板 1”。

因此，如果我们之前已经见到过(total, plank count)这样的组合（plank count 为使用的木板数量），则不需要再进行递归调用。我们可以使用以(total, plank count)为键的散列集合来实现该方法。

> 很多求职者都会在此处出错。他们并没有在(total, plank count)再次出现时停止递归调用，而是在 total 再次出现时停止了递归调用。这并不正确。两块长度为 1 的木板与一块长度为 2 的木板并不相等，这是因为剩余可用的木板数量并不一样。在记忆化算法中，选择以什么值为键时需十分谨慎。

该解法的代码与前述解法的代码大同小异。

```
1   HashSet<Integer> allLengths(int k, int shorter, int longer) {
2     HashSet<Integer> lengths = new HashSet<Integer>();
3     HashSet<String> visited = new HashSet<String>();
4     getAllLengths(k, 0, shorter, longer, lengths, visited);
5     return lengths;
6   }
7
8   void getAllLengths(int k, int total, int shorter, int longer,
9                      HashSet<Integer> lengths, HashSet<String> visited) {
10    if (k == 0) {
11      lengths.add(total);
12      return;
13    }
14    String key = k + " " + total;
15    if (visited.contains(key)) {
16      return;
17    }
18    getAllLengths(k - 1, total + shorter, shorter, longer, lengths, visited);
19    getAllLengths(k - 1, total + longer, shorter, longer, lengths, visited);
20    visited.add(key);
21  }
```

为简便起见，我们将键设置为由 total 和使用的木板数量组成的字符串。一些人或许会认为最好使用一个数据结构存储键值。这样做当然会有一定的益处，但是也有不便之处。你需要和面试官讨论一下此处应如何权衡。

该算法的运行时间需要费些功夫才能求出。

一种方法是，我们可以将该算法想象为填充一个两轴分别为和（SUM）与已使用木板数量

（PLANK COUNT）的表格，其中和的最大可能值为 K*LONGER，而已使用木板数量的最大可能值为 K。因此，运行时间不会慢于 $O(K^2 \times \text{LONGER})$。

当然，这其中的很多和事实上永远不会出现。我们可以获得多少个不重复的和呢？请注意，具有相同数量的每种木板的路径总会得到相同的和。对于一种木板，由于最多只能使用 K 块，因此我们只能获得 $K+1$ 种不同的和。因此，表格的大小实际上为 $(K+1)^2$，运行时间为 $O(K^2)$。

3. 最优解法

如果再读一遍上一段内容，你或许会注意到一件有趣的事：我们最多只能获得 K 种不同的和。这不正是该题目的关键所在吗（即找出所有可能的和）？

事实上，我们不需要遍历所有木板的使用方案，只需遍历所有不重复的、由 K 块木板组成的集合即可（我们使用了集合，不需要元素有序）。如果木板的种类只有两种的话，选择 K 块木板只有 K 种方法：{0 块木板 A，K 块木板 B}，{1 块木板 A，K–1 块木板 B}，{2 块木板 A，K–2 块木板 B}，以此类推。

一个简单的 for 循环语句即可解决此题。在每个“木板使用序列”中，我们只需计算和即可。

```
1  HashSet<Integer> allLengths(int k, int shorter, int longer) {
2    HashSet<Integer> lengths = new HashSet<Integer>();
3    for (int nShorter = 0; nShorter <= k; nShorter++) {
4      int nLonger = k - nShorter;
5      int length = nShorter * shorter + nLonger * longer;
6      lengths.add(length);
7    }
8    return lengths;
9  }
```

此处使用了 HashSet 以便该解法与前述解法保持一致。事实上并非必须如此，这是因为此解法中我们不会遇到重复的元素，只需使用 ArrayList 即可。然而，如果使用 ArrayList，则需要注意处理两种木板长度相同这个边界情况。在此情况下，我们只需返回一个长度为 1 的 ArrayList 即可。

16.6 XML 编码

XML 极为冗长，你找到一种编码方式，可将每个标签对应为预先定义好的整数值，该编码方式的语法如下：

```
Element   --> Tag Attributes END Children END
Attribute --> Tag Value
END       --> 0
Tag       --> 映射至某个预定义的整数值
Value     --> 字符串值
```

例如，下列 XML 会被转换压缩成下面的字符串（假定对应关系为 family -> 1、person -> 2、firstName -> 3、lastName -> 4、state -> 5）。

```
<family lastName="McDowell" state="CA">
  <person firstName="Gayle">Some Message</person>
</family>
```

变为：

```
1 4 McDowell 5 CA 0 2 3 Gayle 0 Some Message 0 0
```

编写代码，打印 XML 元素编码后的版本（传入 Element 和 Attribute 对象）。

题目解法

由题目可知，元素会以 Element 和 Attribute 作为参数传入，因此具体实现代码相当简单，可以运用类似于树状结构的做法实现。

我们会不断对 XML 结构的各个部分调用 encode()，XML 元素的类型不同，处理方式会稍有不同。

```
1  void encode(Element root, StringBuilder sb) {
2    encode(root.getNameCode(), sb);
3    for (Attribute a : root.attributes) {
4      encode(a, sb);
5    }
6    encode("0", sb);
7    if (root.value != null && root.value != "") {
8      encode(root.value, sb);
9    } else {
10     for (Element e : root.children) {
11       encode(e, sb);
12     }
13   }
14   encode("0", sb);
15 }
16
17 void encode(String v, StringBuilder sb) {
18   sb.append(v);
19   sb.append(" ");
20 }
21
22 void encode(Attribute attr, StringBuilder sb) {
23   encode(attr.getTagCode(), sb);
24   encode(attr.value, sb);
25 }
26
27 String encodeToString(Element root) {
28   StringBuilder sb = new StringBuilder();
29   encode(root, sb);
30   return sb.toString();
31 }
```

请留意第 17 行代码，有个负责处理字符串的简单 encode 方法。这个方法似乎有些画蛇添足，无非就是插入字符串并附加一个空格。不过，使用这个方法有个好处是，可以确保每个元素之间都有空格。否则，很可能就会忘记附加空白符从而打乱编码。

16.7　平分正方形

给定两个正方形及一个二维平面。请找出将这两个正方形分割成两半的一条直线。假设正方形顶边和底边与 x 轴平行。

题目解法

开始之前，我们需要思考一个问题：究竟题目中所说的“直线”是指什么？直线是由斜率和 y 轴截距定义的，还是由直线上的两点？或者所谓“直线”是指一条线段，且其起点和终点都在正方形的边上？

我们将假设该题是指上述第三种情况（因为这种情况下题目会更有趣一些）：直线的两端都位于正方形的边上。在面试中，你应该与面试官讨论一下题目为何种情况。

这条将两个正方形平分的直线必定通过两个正方形的中心。我们很容易就可以得知该直线的斜率应为 slope = $(y_1 - y_2) / (x_1 - x_2)$。通过两个正方形的中心计算得出直线的斜率之后，我们可以使用相同的公式来计算线段的起点和终点。

下面的代码中，我们假设原点(0, 0)位于左上角。

```
public class Square {
  ...
  public Point middle() {
    return new Point((this.left + this.right) / 2.0,
                     (this.top + this.bottom) / 2.0);
  }

  /* 返回连接 mid1 和 mid2 线段与 1 号正方形的交点，
   * 也就是说，从 mid1 到 mid2 画一条直线，延长至与正方形边界相交 */
  public Point extend(Point mid1, Point mid2, double size) {
    /* 计算连接 mid1 和 mid2 直线的方向 */
    double xdir = mid1.x < mid2.x ? -1 : 1;
    double ydir = mid1.y < mid2.y ? -1 : 1;

    /* 如果 mid1 和 mid2 的 x 值相同，那么会出现除数为 0 的异常，
     * 因此对此种情况特殊计算 */
    if (mid1.x == mid2.x) {
      return new Point(mid1.x, mid1.y + ydir * size / 2.0);
    }

    double slope = (mid1.y - mid2.y) / (mid1.x - mid2.x);
    double x1 = 0;
    double y1 = 0;

    /* 使用公式(y1-y2)/(x1-x2)计算斜率。注意，如果斜率较大（> 1），
     * 那么直线将与 y 轴的中点相距 size/2 个单位。如果斜率较小（< 1），
     * 那么直线将与 x 轴的中点相距 size/2 个单位 */
    if (Math.abs(slope) == 1) {
      x1 = mid1.x + xdir * size / 2.0;
      y1 = mid1.y + ydir * size / 2.0;
    } else if (Math.abs(slope) < 1) { // 较小斜率
      x1 = mid1.x + xdir * size / 2.0;
      y1 = slope * (x1 - mid1.x) + mid1.y;
    } else { // 较大斜率
      y1 = mid1.y + ydir * size / 2.0;
      x1 = (y1 - mid1.y) / slope + mid1.x;
    }
    return new Point(x1, y1);
  }

  public Line cut(Square other) {
    /* 计算那条线将与正方形的边相交 */
    Point p1 = extend(this.middle(), other.middle(), this.size);
    Point p2 = extend(this.middle(), other.middle(), -1 * this.size);
    Point p3 = extend(other.middle(), this.middle(), other.size);
    Point p4 = extend(other.middle(), this.middle(), -1 * other.size);

    /* 对于上述的点，找到线段的起止点。start 为最左点（如果相同则为较靠上方的点），
     * end 为最右点（如果相同则为较靠下方的点）*/
    Point start = p1;
    Point end = p1;
    Point[] points = {p2, p3, p4};
    for (int i = 0; i < points.length; i++) {
      if (points[i].x < start.x ||
```

```
55          (points[i].x == start.x && points[i].y < start.y)) {
56        start = points[i];
57      } else if (points[i].x > end.x ||
58                 (points[i].x == end.x && points[i].y > end.y)) {
59        end = points[i];
60      }
61    }
62
63    return new Line(start, end);
64  }
```

该题的主要目的在于考查你编写代码时是否足够细心。该题目很容易就遗漏特殊情况（比如，两个正方形的中点相同）。你应该在解题之前就列出所有特殊情况的清单以便后面可以正确处理所有情况。解出该题需要进行仔细和充分的测试。

16.8　最佳直线

给定一个二维平面及平面上的若干点。请找出一条直线，其通过的点的数目最多。

题目解法

刚看到此题时会觉得解法非常简单。某种程度上确实如此。

对于每两个点，我们只需要在平面上画一条通过这两点的无限长度的直线（换句话说，并非线段）。我们可以使用一个散列表来记录哪条直线出现的次数最多。该解法花费的时间为 $O(N^2)$，这是因为一共需要画 N^2 条直线。

我们可以使用斜率和 y 轴截距来表示一条直线（而不是通过直线上的两点进行表示），以便检查从（$x1, y1$）至（$x2, y2$）的直线是否和从（$x3, y3$）至（$x4, y4$）的直线为同一条直线。

为了找到出现次数最多的直线，我们需要对所有的直线进行迭代，使用散列表记录每条直线出现的次数。该算法非常简单。

但是，有一处稍有些复杂。在我们的定义中，如果两条直线有相同的斜率和 y 轴截距，则两条直线相等。我们在进一步的操作中，会根据这两个值（特别是斜率）对所有的直线计算散列值。问题就在于浮点数并不总能被精确地表示为二进制数。我们可以通过检查两个浮点数的差值是否小于 `epsilon` 变量来解决这个问题。

对于散列表会有什么问题？这表示具有两个“相等”斜率的直线，通过散列函数得到的值或许并不一样。要解决散列表中的问题，我们需要将斜率向下近似为下一个能够表示的精度，并使用 `flooredSlop` 作为散列表的键。之后，我们需要将所有**可能**相等的直线取出进行对比，即我们需要从散列表中将 `flooredSlop`、`flooredSlop - epsilon` 和 `flooredSlop + epsilon` 这三个键所对应的值全部取出，这样做才可以保证我们取出了所有可能相等的直线。

```
1   /* 找到穿过最多点的直线 */
2   Line findBestLine(GraphPoint[] points) {
3     HashMapList<Double, Line> linesBySlope = getListOfLines(points);
4     return getBestLine(linesBySlope);
5   }
6
7   /* 将一对点作为一条直线加入到链表中 */
8   HashMapList<Double, Line> getListOfLines(GraphPoint[] points) {
9     HashMapList<Double, Line> linesBySlope = new HashMapList<Double, Line>();
10    for (int i = 0; i < points.length; i++) {
11      for (int j = i + 1; j < points.length; j++) {
12        Line line = new Line(points[i], points[j]);
```

```
13        double key = Line.floorToNearestEpsilon(line.slope);
14        linesBySlope.put(key, line);
15      }
16    }
17    return linesBySlope;
18  }
19
20  /* 返回具有最多相近直线的直线 */
21  Line getBestLine(HashMapList<Double, Line> linesBySlope) {
22    Line bestLine = null;
23    int bestCount = 0;
24
25    Set<Double> slopes = linesBySlope.keySet();
26
27    for (double slope : slopes) {
28      ArrayList<Line> lines = linesBySlope.get(slope);
29      for (Line line : lines) {
30        /* 对与当前直线相近的直线进行计数 */
31        int count = countEquivalentLines(linesBySlope, line);
32
33        /* 如果比当前直线更好，则进行替换 */
34        if (count > bestCount) {
35          bestLine = line;
36          bestCount = count;
37          bestLine.Print();
38          System.out.println(bestCount);
39        }
40      }
41    }
42    return bestLine;
43  }
44
45  /* 检查相近直线构成的散列表。请注意我们需要检查斜率为当前斜率正负 epsilon 的直线，
46   * 这是我们对于相近直线的定义 */
47  int countEquivalentLines(HashMapList<Double, Line> linesBySlope, Line line) {
48    double key = Line.floorToNearestEpsilon(line.slope);
49    int count = countEquivalentLines(linesBySlope.get(key), line);
50    count += countEquivalentLines(linesBySlope.get(key - Line.epsilon), line);
51    count += countEquivalentLines(linesBySlope.get(key + Line.epsilon), line);
52    return count;
53  }
54
55  /* 计算数组中相近直线的数目（斜率和 y 轴交点相差一个 epsilon 之内）*/
56  int countEquivalentLines(ArrayList<Line> lines, Line line) {
57    if (lines == null) return 0;
58
59    int count = 0;
60    for (Line parallelLine : lines) {
61      if (parallelLine.isEquivalent(line)) {
62        count++;
63      }
64    }
65    return count;
66  }
67
68  public class Line {
69    public static double epsilon = .0001;
70    public double slope, intercept;
71    private boolean infinite_slope = false;
72
73    public Line(GraphPoint p, GraphPoint q) {
```

```
    if (Math.abs(p.x - q.x) > epsilon) { // 如果 x 不同
      slope = (p.y - q.y) / (p.x - q.x); // 计算斜率
      intercept = p.y - slope * p.x; // 通过 y=mx+b 计算 y 轴截距
    } else {
      infinite_slope = true;
      intercept = p.x; // 因为斜率为无穷大，所以计算 x 轴截距
    }
  }

  public static double floorToNearestEpsilon(double d) {
    int r = (int) (d / epsilon);
    return ((double) r) * epsilon;
  }

  public boolean isEquivalent(double a, double b) {
    return (Math.abs(a - b) < epsilon);
  }

  public boolean isEquivalent(Object o) {
    Line l = (Line) o;
    if (isEquivalent(l.slope, slope) && isEquivalent(l.intercept, intercept) &&
       (infinite_slope == l.infinite_slope)) {
      return true;
    }
    return false;
  }
}

/* HashMapList 是从 String 到 ArrayList 的散列表。实现细节详见附录 A */
```

在计算直线斜率时要仔细一些。直线有可能是垂直的，即直线不存在 y 轴截距而斜率为无穷大。我们可以使用一个单独的变量（`infinite_slope`）保存该信息。在判断两条直线相等的方法中，我们需要加入该条件。

16.9　珠玑妙算

珠玑妙算游戏（the game of master mind）的玩法如下。

计算机有 4 个槽，每个槽放一个球，颜色可能是红色（R）、黄色（Y）、绿色（G）或蓝色（B）。例如，计算机可能有 RGGB 4 种（槽 1 为红色，槽 2、3 为绿色，槽 4 为蓝色）。

作为用户，你试图猜出颜色组合。打个比方，你可能会猜 YRGB。

要是猜对某个槽的颜色，则算一次“猜中”；要是只猜对颜色但槽位猜错了，则算一次“伪猜中”。注意，“猜中”不能算入“伪猜中”。

举个例子，实际颜色组合为 RGBY，而你猜的是 GGRR，则算一次猜中，一次伪猜中。

给定一个猜测和一种颜色组合，编写一个方法，返回猜中和伪猜中的次数。

题目解法

此题简单明了，但写代码时很容易犯小错误。代码写好后，你应该对照各种测试用例，进行全面彻底的检查。

编写代码时，我们首先会构造一个频率数组，存放每个字符在 `solution` 中出现的次数，但不包括某个槽被“猜中”的次数。然后，迭代 `guess` 算出伪猜中的次数。

下面是这个算法的实现代码。

```
class Result {
  public int hits = 0;
  public int pseudoHits = 0;

  public String toString() {
    return "(" + hits + ", " + pseudoHits + ")";
  }
}

int code(char c) {
  switch (c) {
  case 'B':
    return 0;
  case 'G':
    return 1;
  case 'R':
    return 2;
  case 'Y':
    return 3;
  default:
    return -1;
  }
}

int MAX_COLORS = 4;

Result estimate(String guess, String solution) {
  if (guess.length() != solution.length()) return null;

  Result res = new Result();
  int[] frequencies = new int[MAX_COLORS];

  /* 计算猜中次数并构造频率表 */
  for (int i = 0; i < guess.length(); i++) {
    if (guess.charAt(i) == solution.charAt(i)) {
      res.hits++;
    } else {
      /* 如果没有猜中，则只对频率加一（频率表用于表示伪猜中的次数）。
       * 如果猜中，那么该位置应该已被使用 */
      int code = code(solution.charAt(i));
      frequencies[code]++;
    }
  }

  /* 计算伪猜中 */
  for (int i = 0; i < guess.length(); i++) {
    int code = code(guess.charAt(i));
    if (code >= 0 && frequencies[code] > 0 &&
        guess.charAt(i) != solution.charAt(i)) {
      res.pseudoHits++;
      frequencies[code]--;
    }
  }
  return res;
}
```

注意，问题所需的算法越简单，写出清晰、正确的代码就越显重要。在上面的例子中，我们提取代码专门写了个 code(char c)方法，并创建了一个 Result 类来保存结果，而非只是打印显示。

16.10　水域大小

你有一个用于表示一片土地的整数矩阵，该矩阵中每个点的值代表对应地点的海拔高度。若值为 0 则表示水域。由垂直、水平或对角连接的水域为池塘。池塘的大小是指相连接的水域的个数。编写一个方法来计算矩阵中所有池塘的大小。

示例：

输入：

```
0 2 1 0
0 1 0 1
1 1 0 1
0 1 0 1
```

输出：2，4，1（任意顺序）

题目解法

首先，我们可以尝试遍历该数组。很容易找到水域：单元格为 0 即为水域。

给定一个水域，我们如何计算其周围水域的总量？如果该水域周围没有相连的且数值为 0 的单元格，那么该池塘的尺寸为 1。如果该水域周围有相连的且数值为 0 的单元格，则需要将相连水域、相连水域的项链水域加入到池塘尺寸中。当然，需要谨慎地进行该过程，不要对水域进行重复的计数。可以通过广度优先搜索或者深度优先搜索的变形完成该过程。每当访问过一个单元格时，我们将其永久地标记为“已访问”。

对于每个单元格，需要检查其 8 个相连接的单元格。可以通过编写代码检查上、下、左、右和 4 个对角方向的单元格，也可以使用循环来更简单地实现该功能。

```
ArrayList<Integer> computePondSizes(int[][] land) {
  ArrayList<Integer> pondSizes = new ArrayList<Integer>();
  for (int r = 0; r < land.length; r++) {
    for (int c = 0; c < land[r].length; c++) {
      if (land[r][c] == 0) { // 可选。此处总会返回
        int size = computeSize(land, r, c);
        pondSizes.add(size);
      }
    }
  }
  return pondSizes;
}

int computeSize(int[][] land, int row, int col) {
  /* 如果超出边界或者已经访问过 */
  if (row < 0 || col < 0 || row >= land.length || col >= land[row].length ||
      land[row][col] != 0) { // 访问过或者非水域
    return 0;
  }
  int size = 1;
  land[row][col] = -1; // 标记访问过
  for (int dr = -1; dr <= 1; dr++) {
    for (int dc = -1; dc <= 1; dc++) {
      size += computeSize(land, row + dr, col + dc);
    }
  }
  return size;
}
```

在本题中，我们通过将一个单元格设置为–1 来表示该单元格已被访问过。这样通过一行

（land\[row\]\[col\] != 0）代码就能检查出某单元格是否被访问过以及是否为干燥陆地这两种情况。无论是这两种情况中的哪一种，该单元格的值都不是 0。

你或许还会注意到，并非对 8 个单元格进行了迭代，而是对 9 个单元格进行了迭代。循环中同时包括了当前单元格。我们可以加入一行代码，使得 dr == 0 以及 dc == 0 时不进行递归调用。但是这样做并不能节省很多时间。仅仅是为了避免 1 个递归调用，我们需要对 8 个单元格画蛇添足地执行该 if 语句。而由于访问的单元格已经被标记为“已访问”，该递归调用实际上会立即返回。

如果你不想改变输入的矩阵，可以创建另外一个 visited 矩阵记录已访问的单元格。

```
ArrayList<Integer> computePondSizes(int[][] land) {
  boolean[][] visited = new boolean[land.length][land[0].length];
  ArrayList<Integer> pondSizes = new ArrayList<Integer>();
  for (int r = 0; r < land.length; r++) {
    for (int c = 0; c < land[r].length; c++) {
      int size = computeSize(land, visited, r, c);
      if (size > 0) {
        pondSizes.add(size);
      }
    }
  }
  return pondSizes;
}

int computeSize(int[][] land, boolean[][] visited, int row, int col) {
  /* 如果超出边界或者已经访问过 */
  if (row < 0 || col < 0 || row >= land.length || col >= land[row].length ||
    visited[row][col] || land[row][col] != 0) {
  return 0;
  }
  int size = 1;
  visited[row][col] = true;
  for (int dr = -1; dr <= 1; dr++) {
    for (int dc = -1; dc <= 1; dc++) {
      size += computeSize(land, visited, row + dr, col + dc);
    }
  }
  return size;
}
```

两种实现方法花费的时间都为 $O(WH)$，其中 W 是矩阵的宽度，H 是矩阵的高度。

> 请注意，很多人经常使用 $O(N)$或者 $O(N^2)$的表述方式，仿佛 N 存在着一种固有的定义。其实并非这样。假设有一个方形矩阵。你可以将运行时间表述为 $O(N)$或者 $O(N^2)$。两种方法都是对的，只是 N 的含义有所不同。当 N 表示方形矩阵的边长时，运行时间为 $O(N^2)$。当 N 表示方形矩阵的单元格数量时，运行时间为 $O(N)$。对于 N 的定义务请谨慎。实际上，如果题目中 N 的定义有可能出现歧义，完全不使用 N 或许是上乘之选。

有些人或许会将运行时间错误地计算为 $O(N^4)$。他们认为 computeSize 运行时间为 $O(N^2)$，而该方法会被调用 $O(N^2)$次（显然，他们也假设矩阵的大小为 $N \times N$）。尽管这两条理由都是正确的，但是你不能将它们简单地相乘。这是因为当单次调用 computeSize 的时间复杂度上升时，该方法的调用次数会出现下降。

例如，假设我们第一次调用 computeSize 方法。该调用或许会花费 $O(N^2)$的时间，但是我们在此之后再也不会调用该方法。

另一种分析时间复杂度的方法是通过计算每个单元格在一次调用中被触及的次数。每个单元格会被 `computePondSizes` 函数触及一次。另外，每个单元格可能会被它的每个相邻单元格触及一次。每个单元格被触及的次数仍然为常数。因此，对于 $N \times N$ 的矩阵来说，总体的运行时间仍为 $O(N^2)$，或者一般而言表示为 $O(WH)$。

16.11　T9 键盘

在老式手机上，用户通过数字键盘输入，手机将提供与这些数字相匹配的单词列表。每个数字映射到 0 至 4 个字母。给定一个数字序列，实现一个算法来返回匹配单词的列表。你会得到一张含有有效单词的列表（存储你想要的任何数据结构）。映射如下图所示。

1	**2** abc	**3** def
4 ghi	**5** jkl	**6** mno
7 pqrs	**8** tuv	**9** wxyz
	0	

示例：

输入：8733

输出：tree，used

题目解法

可以通过几种不同的方法解答此题。让我们从蛮力法开始。

1. 蛮力法

想象一下，如果通过笔算，你会如何解答此题？你或许会尝试每一位数字对应字符的所有可能组合。

这也正是我们使用算法解答该题的思路。对于第一位数字，首先遍历所有它对应的字符。对于每一个字符，我们将其添加到 `prefix` 变量的尾部并进行递归，不断地将 `prefix` 传递到下一层递归调用中。当没有剩余的字符时，如果 `prefix`（`prefix` 此时即为整个单词）是一个合法的单词，就将其打印出来。

我们假设传入的单词列表以散列集合（`HashSet`）存储。散列集合与散列表类似，但是它并不提供由键到值的映射，而是提供了判断集合中是否包含某个单词的功能（该操作的运行时间为 $O(1)$）。

```
ArrayList<String> getValidT9Words(String number, HashSet<String> wordList) {
  ArrayList<String> results = new ArrayList<String>();
  getValidWords(number, 0, "", wordList, results);
  return results;
}

void getValidWords(String number, int index, String prefix,
                   HashSet<String> wordSet, ArrayList<String> results) {
  /* 如果是完整单词则打印 */
  if (index == number.length() && wordSet.contains(prefix)) {
    results.add(prefix);
    return;
  }
```

```
14
15    /* 获取匹配该位数字的字符 */
16    char digit = number.charAt(index);
17    char[] letters = getT9Chars(digit);
18
19    /* 遍历其余选项 */
20    if (letters != null) {
21      for (char letter : letters) {
22        getValidWords(number, index + 1, prefix + letter, wordSet, results);
23      }
24    }
25  }
26
27  /* 返回映射到此数字的所有字符 */
28  char[] getT9Chars(char digit) {
29    if (!Character.isDigit(digit)) {
30      return null;
31    }
32    int dig = Character.getNumericValue(digit) - Character.getNumericValue('0');
33    return t9Letters[dig];
34  }
35
36  /* 数字到字符的映射 */
37  char[][] t9Letters = {null, null, {'a', 'b', 'c'}, {'d', 'e', 'f'},
38    {'g', 'h', 'i'}, {'j', 'k', 'l'}, {'m', 'n', 'o'}, {'p', 'q', 'r', 's'},
39    {'t', 'u', 'v'}, {'w', 'x', 'y', 'z'}
40  };
```

该算法花费的时间为 $O(4^N)$，其中 N 是字符串的长度。这是因为，对于每一次 `getValidWords` 调用，我们都递归地将其分为 4 个分支，而该递归直到重复 N 次后才停止。

对于较长的字符串，该算法极其缓慢。

2. 优化解法

再来回顾一下笔算时如何解题。假如有一个例子是 33835676368（对应的单词为 development）。如果你进行笔算，我打赌你一定不会以 fttf（3383）作为起始，这是因为没有任何合法单词会以这 4 个字符开始。

理想情况下，我们希望该算法也可以进行类似的优化，不要尝试那些明显会失败的递归路径。特别是如果字典中的单词都不以 `prefix` 为前缀，则无须继续递归下去。

单词查找树（详见 9.4.4 节）是可以达到该目的的数据结构。只要我们构造的字符串不是合法的前缀，就退出递归。

```
1   ArrayList<String> getValidT9Words(String number, Trie trie) {
1     ArrayList<String> results = new ArrayList<String>();
2     getValidWords(number, 0, "", trie.getRoot(), results);
3     return results;
4   }
5
6   void getValidWords(String number, int index, String prefix, TrieNode trieNode,
7                      ArrayList<String> results) {
8     /* 如果是完整单词则打印 */
9     if (index == number.length()) {
10      if (trieNode.terminates()) { // 完整单词
11        results.add(prefix);
12      }
13      return;
14    }
15
16    /* 获取匹配该位数字的字符 */
```

```
    char digit = number.charAt(index);
    char[] letters = getT9Chars(digit);

    /* 遍历其余选项 */
    if (letters != null) {
      for (char letter : letters) {
        TrieNode child = trieNode.getChild(letter);
        /* 如果有单词以 prefix + letter 为开始则继续递归 */
        if (child != null) {
          getValidWords(number, index + 1, prefix + letter, child, results);
        }
      }
    }
  }
```

很难描述该算法的时间复杂度，这取决于你如何组织语言。但是，在实践中，这种“提前返回”的策略会使得程序运行得快很多。

3. 最优算法

无论你是否相信，我们确实可以使得该算法更快一些。需要做一些预处理操作，这并不是很难，反正最终都需要构造单词查找树。

该题要求列出 T9 键盘中一串数字可能代表的所有单词。与其动态地生成结果（遍历大量的潜在字符串，而其中许多字符串并不是该题的解），不如预先进行计算。

该算法由如下两个步骤构成。

- 预处理

(1) 构造一个散列表，将一串数字映射到一组字符串上。

(2) 遍历字典中的每个单词，并将这些单词转换为其 T9 形式的表达式（比如，APPLE -> 27753）。将每个结果都存于步骤(1)中的散列表中。例如，8733 会映射为{used, tree}。

- 单词查找

在散列表中查找给定数字并返回一组字符串。

只需如上步骤即可轻松解答此题！

```
/* 查询单词 */
ArrayList<String> getValidT9Words(String numbers,
                                  HashMapList<String, String> dictionary) {
  return dictionary.get(numbers);
}

/* 预计算 */

/* 创建一个散列表，从一个数字映射到其代表的所有单词 */
HashMapList<String, String> initializeDictionary(String[] words) {
  /* 创建一个散列表，从一个字符映射到数值 */
  HashMap<Character, Character> letterToNumberMap = createLetterToNumberMap();

  /* 创建单词到数字的映射 */
  HashMapList<String, String> wordsToNumbers = new HashMapList<String, String>();
  for (String word : words) {
    String numbers = convertToT9(word, letterToNumberMap);
    wordsToNumbers.put(numbers, word);
  }
  return wordsToNumbers;
}

/* 将数字到字母的映射转化为字母到数字的映射 */
```

```
HashMap<Character, Character> createLetterToNumberMap() {
  HashMap<Character, Character> letterToNumberMap =
    new HashMap<Character, Character>();
  for (int i = 0; i < t9Letters.length; i++) {
    char[] letters = t9Letters[i];
    if (letters != null) {
      for (char letter : letters) {
        char c = Character.forDigit(i, 10);
        letterToNumberMap.put(letter, c);
      }
    }
  }
  return letterToNumberMap;
}

/* 将字符串转化为 T9 表示法 */
String convertToT9(String word, HashMap<Character, Character> letterToNumberMap) {
  StringBuilder sb = new StringBuilder();
  for (char c : word.toCharArray()) {
    if (letterToNumberMap.containsKey(c)) {
      char digit = letterToNumberMap.get(c);
      sb.append(digit);
    }
  }
  return sb.toString();
}

char[][] t9Letters = /* 同前 */

/* HashMapList 是从 String 到 ArrayList 的散列表。实现细节详见附录 A */
```

获得映射到一个数字的单词列表需要花费 $O(N)$的时间，其中 N 为数字的位数。在散列表查找值时花费的时间为 $O(N)$（我们需要将数字转换为散列表）。如果你可以确定单词的长度一定小于某个特定的值，那么也可以将运行时间描述为 $O(1)$。

请注意，你可能很容易就认为：“哦，线性时间复杂度也不是很快。”但是，快与慢取决于线性复杂度是相对于什么因素。相对于单词长度的线性复杂度是极其快速的，相对于字典大小的线性复杂度则没有那么快。

16.12 数对和

设计一个算法，找出数组中两数之和为指定值的所有整数对。

题目解法

此题有两种解法，至于哪一种“比较好”，取决于你在时间效率、空间效率和代码复杂度之间如何取舍。

首先从定义入手。如果要找一对总和为 z 的数，那么 x 的补数为 $z-x$（即与 x 相加得 z 的数）。举个例子，如果要找一对总和为 12 的数，那么，–5 的补数为 17。

1. 蛮力法

一种蛮力解法是对所有数对进行迭代，如果发现数对的和与目标和相等，则打印该数对。

```
ArrayList<Pair> printPairSums(int[] array, int sum) {
  ArrayList<Pair> result = new ArrayList<Pair>();
  for (int i = 0 ; i < array.length; i++) {
    for (int j = i + 1; j < array.length; j++) {
```

```
        if (array[i] + array[j] == sum) {
          result.add(new Pair(array[i], array[j]));
        }
      }
    }
    return result;
  }
```

如果数组中存在重复元素（比如{5, 6, 5}），该算法可能会将同一个数对和打印两次。你应该与面试官讨论一下这种情况。

2. 优化解法

可以通过散列表对前面的算法进行优化，散列表用于保存一个键及其所对应的“没有配对”数值的个数。我们从头至尾对数组进行扫描。对于每一个元素 x，需要检查数组中位于该元素位置之前的所有元素中，有多少没有配对的 x 的补数。如果数量至少为 1，则存在一个没有配对的 x 的补数，我们将这一对数字加入到结果中，并将散列表中 x 的补数对应的值减一，以便表示该元素已经进行了配对；如果数量为 0，则将散列表中 x 的值加一，以便表示该数字尚未配对。

```
ArrayList<Pair> printPairSums(int[] array, int sum) {
  ArrayList<Pair> result = new ArrayList<Pair>();
  HashMap<Integer, Integer> unpairedCount = new HashMap<Integer, Integer>();
  for (int x : array) {
    int complement = sum - x;
    if (unpairedCount.getOrDefault(complement, 0) > 0) {
      result.add(new Pair(x, complement));
      adjustCounterBy(unpairedCount, complement, -1); // 减少 complement 变量的值
    } else {
      adjustCounterBy(unpairedCount, x, 1); // 增加计数
    }
  }
  return result;
}

void adjustCounterBy(HashMap<Integer, Integer> counter, int key, int delta) {
  counter.put(key, counter.getOrDefault(key, 0) + delta);
}
```

该算法会打印重复的结果，但是不会重复使用一个元素。该算法花费的时间为 $O(N)$，占用的空间也为 $O(N)$。

3. 另一种解法

另一种方法是：对数组进行排序，并在一次扫描中找到所有数对。假设有如下数组：

```
{-2, -1, 0, 3, 5, 6, 7, 9, 13, 14}
```

令 first 指向数组开头，last 指向数组结尾。要找出 first 的补数，就将 last 往回移动，直至找到补数。如果 first + last < sum，则数组中不存在 first 的补数，因此可以向前移动 first，等到 first 比 last 大时停止操作。

为什么这么做就能找出 first 的所有补数？因为这个数组是排好序的，而且我们是从最小的数字开始逐一尝试的。当 first 与 last 的总和小于 sum 时，可以确定就算继续尝试更小的数（像 last 那样往回移动）也找不到补数。

为什么这么做可以找出 last 的所有补数？因为所有数值对必定由 first 和 last 组成。找出 first 的所有补数，就等于找出了 last 的所有补数。

```
void printPairSums(int[] array, int sum) {
```

```
    Arrays.sort(array);
    int first = 0;
    int last = array.length - 1;
    while (first < last) {
      int s = array[first] + array[last];
      if (s == sum) {
        System.out.println(array[first] + " " + array[last]);
        first++;
        last--;
      } else {
        if (s < sum) first++;
       else last--;
      }
    }
  }
```

该算法花费 $O(N \log N)$的时间进行排序，花费 $O(N)$的时间查找数对。

请注意，由于假定数组是无序的，如果以 O 表示时间复杂度，还有一种算法和该算法速度一样快。我们可以使用二分查找法查找每个元素的补数。若使用此方法，算法一共分为两步，每一步花费的时间都为 $O(N \log N)$。

16.13 LRU 缓存

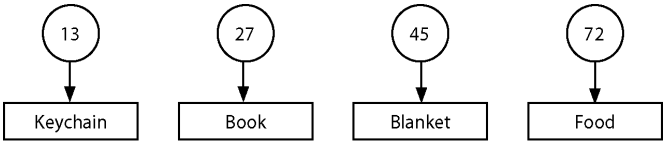
设计和构建一个“最近最少使用”缓存，该缓存会删除最近最少使用的项目。缓存应该从键映射到值（允许插入和检索特定键对应的值），并在初始化时指定最大容量。当缓存被填满时，它应该删除最近最少使用的项目。

题目解法

首先应该定义该题目的范围。我们的目标究竟是什么？

- **插入一个键值对。**我们需要插入一个类似于(键, 值)的对。
- **通过键获取值。**我们需要能够使用键从缓存中获取值。
- **查找最近最少使用的元素。**我们需要知道最近最少使用的元素（并且有可能需要所有元素的使用顺序）。
- **更新最近使用的元素。**如果通过键从缓存中获取了某个值，需要更新使用顺序，使得该元素为最近使用的元素。
- **移除元素。**缓存需要设置最大容量，如果元素数目超过了最大容量，缓存应该移除最近最少使用的元素。

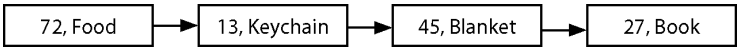
上面提到的(键, 值)对意味着我们可以使用散列表，使得通过键获取值的操作更为简单。



可惜，使用散列表，并不能快速移除最近使用的元素。我们可以在每一个元素上标记一个时间戳，并通过对散列表中所有元素的迭代，移除具有最小时间戳的元素。但是，该方法运行十分缓慢（插入操作花费的时间为 $O(N)$）。

取而代之的一种方法是，我们可以使用链表这种数据结构，其中链表的元素按照最近使用顺序进行排序。这样做便于标记最近使用的元素（即将元素插入到链表头部）或移除最近最少

使用的元素（即从链表尾部删除元素）。

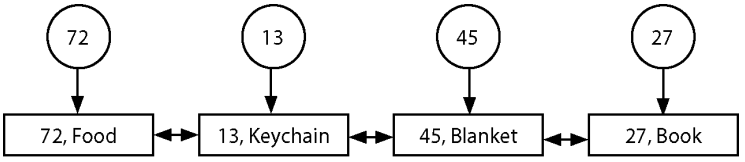


可惜，该方法无法快速使用键获取对应的值。我们可以对链表进行迭代，并通过键查找对应的元素，可是这样做会非常缓慢（获取元素花费的时间为 $O(N)$）。

上面的每一种方法都很好地解决了一半问题（两个方法解决了不同的两个部分），但是都没有完全解答该题目。

我们是否可以使用两种方法的精髓之处呢？当然可以。我们可以同时使用两种方法。

对于上面例子中描述的链表，我们现在使用双向链表进行存储。这样一来就便于从链表中间移除元素了。而对于前面提到的散列表，我们现在使用链表中的节点（而不是直接使用键值对中的值）作为散列表的值。



至此，该算法如下所示。

- **插入一个键值对**。创建一个由键值对组成的链表。对于插入的键值对，将其加入到链表的头部，同时将键->链表节点的映射加入到散列表中。
- **通过键获取值**。在散列表中查找给定的键，并返回其对应的值。同时，更新最近使用的元素（具体方法见下面代码）。
- **查找最近最少使用的元素**。最近最少使用的元素位于链表的尾部。
- **更新最近使用的元素**。将对应的节点移动到链表的头部。此时无须更新散列表。
- **移除元素**。移除链表尾部的元素。从被移除的节点中获取键，并将该键对应的映射从散列表中移除。

下面的代码实现了本题中使用的类和算法。

```
public class Cache {
  private int maxCacheSize;
  private HashMap<Integer, LinkedListNode> map =
    new HashMap<Integer, LinkedListNode>();
  private LinkedListNode listHead = null;
  public LinkedListNode listTail = null;

  public Cache(int maxSize) {
    maxCacheSize = maxSize;
  }

  /* 获得键对应的值并标记最近使用过 */
  public String getValue(int key) {
    LinkedListNode item = map.get(key);
    if (item == null) return null;

    /* 移动到链表前端并标记最近使用过 */
    if (item != listHead) {
      removeFromLinkedList(item);
      insertAtFrontOfLinkedList(item);
    }
    return item.value;
```

```
23    }
24
25    /* 从链表中移除节点 */
26    private void removeFromLinkedList(LinkedListNode node) {
27      if (node == null) return;
28
29      if (node.prev != null) node.prev.next = node.next;
30      if (node.next != null) node.next.prev = node.prev;
31      if (node == listTail) listTail = node.prev;
32      if (node == listHead) listHead = node.next;
33    }
34
35    /* 插入到链表前端 */
36    private void insertAtFrontOfLinkedList(LinkedListNode node) {
37      if (listHead == null) {
38        listHead = node;
39        listTail = node;
40      } else {
41        listHead.prev = node;
42        node.next = listHead;
43        listHead = node;
44      }
45    }
46
47    /* 将键值对从缓存中移除，即从链表和散列表中移除 */
48    public boolean removeKey(int key) {
49      LinkedListNode node = map.get(key);
50      removeFromLinkedList(node);
51      map.remove(key);
52      return true;
53    }
54
55    /* 将键值对插入到缓存中。如果需要则删除旧的值。
56     * 将键值对插入到链表和散列表中 */
57    public void setKeyValue(int key, String value) {
58      /* 如果已经存在则删除 */
59      removeKey(key);
60
61      /* 如果超过限制，则删除最久没有使用的 */
62      if (map.size() >= maxCacheSize && listTail != null) {
63        removeKey(listTail.key);
64      }
65
66      /* 插入新节点 */
67      LinkedListNode node = new LinkedListNode(key, value);
68      insertAtFrontOfLinkedList(node);
69        map.put(key, node);
70    }
71
72    private static class LinkedListNode {
73      private LinkedListNode next, prev;
74      public int key;
75      public String value;
76      public LinkedListNode(int k, String v) {
77        key = k;
78        value = v;
79      }
80    }
81  }
```

请注意，我们选择将 `LinkedListNode` 类作为 `Cache` 类的内部类。这是因为其他类都不需要访问该类，且该类应该只存在于 `Cache` 类的定义范围之内。

16.14　计算器

给定一个包含正整数、加（+）、减（–）、乘（×）、除（/）的算数表达式（括号除外），计算其结果。

示例：

输入：2 * 3 + 5/6 * 3 + 15
输出：23.5

题目解法

我们应该认识到的第一个显而易见的事实是，从左向右依次计算每个运算符是不可行的。乘法和除法是“更高优先级”的运算，因此它们必须在加法之前发生。

例如，如果你得到一个简单的表达式：3 + 6 × 2，则必须首先执行乘法，然后再执行加法。如果你只是从左到右地处理这个方程，那么最终会得到不正确的结果 18，而不是正确的结果 15。我相信你一定知道这些道理，但是确实有必要事先强调一下。

解法 1

我们仍然可以从左到右处理该方程，只需要在处理时更加智能化一些。乘法和除法需要组合在一起，以便每当我们发现这些运算时，立即对其周围的变量进行计算。

例如，假设我们有如下的表达式：

```
2 - 6 - 7 * 8/2 + 5
```

你可以立即计算 2 – 6 并将其存储到结果变量中。但是，当发现 7 ×（某表达式）时，知道需要先完全处理该表达式，再将计算结果加入到结果变量中。

可以通过从左到右依次读取表达式并维护两个变量的方法做到这一点。

- 第一个变量是 processing，它维护当前各项的结果（运算符与值）。在加法和减法的情况下，只需保存当前的各项即可。在乘法和除法的情况下，则需要保存完整的表达式序列（直到发现下一个加法或减法）。
- 第二个变量是 result。如果下一项是加法或减法（或者没有下一项），则 processing 会被加入到 result 中。

在上面的例子中，我们将执行以下操作。

(1) 读取+2。将其加入到 processing 变量中。将 processing 变量加入到 result 变量中。清空 processing 变量。

```
processing = {+, 2} --> null
result = 0          --> 2
```

(2) 读取–6。将其加入到 processing 变量中。将 processing 变量加入到 result 变量中。清空 processing 变量。

```
processing = {-, 6} --> null
result = 2          --> -4
```

(3) 读取–7。将其加入到 processing 变量中。发现下一个运算符为 × 。继续处理。

```
processing = {-, 7}
result = -4
```

(4) 读取 × 8。将其加入到 processing 变量中。发现下一个运算符为/。继续处理。

```
processing = {-, 56}
result = -4
```

(5) 读取/2。将其加入到 processing 变量中。发现下一个运算符为+，该运算符会终止当前的乘法与除法表达式。将 processing 变量加入到 result 变量中。清空 processing 变量。

```
processing = {-, 28} --> null
result = -4          --> -32
```

(6) 读取+5。将其加入到 processing 变量中。将 processing 变量加入到 result 变量中。清空 processing 变量。

```
processing = {+, 5} --> null
result = -32        --> -27
```

下面的代码实现了该算法：

```
/* 计算四则运算的结果，即从左至右读取内容并计算结果。
 * 当发现乘除法时，我们应使用一个临时变量 */
double compute(String sequence) {
  ArrayList<Term> terms = Term.parseTermSequence(sequence);
  if (terms == null) return Integer.MIN_VALUE;

  double result = 0;
  Term processing = null;
  for (int i = 0; i < terms.size(); i++) {
    Term current = terms.get(i);
    Term next = i + 1 < terms.size() ? terms.get(i + 1) : null;

    /* 将当前项应用于 processing */
    processing = collapseTerm(processing, current);

    /* 如果下一项是加减法，则此组计算已完成。
     * 我们应将 processing 结果添加到 result 中 */
    if (next == null || next.getOperator() == Operator.ADD
      || next.getOperator() == Operator.SUBTRACT) {
      result = applyOp(result, processing.getOperator(), processing.getNumber());
      processing = null;
    }
  }

  return result;
}

/* 使用第二项的运算符和每一项的数字合并项 */
Term collapseTerm(Term primary, Term secondary) {
  if (primary == null) return secondary;
  if (secondary == null) return primary;

  double value = applyOp(primary.getNumber(), secondary.getOperator(),
                 secondary.getNumber());
  primary.setNumber(value);
  return primary;
}

double applyOp(double left, Operator op, double right) {
  if (op == Operator.ADD) return left + right;
  else if (op == Operator.SUBTRACT) return left - right;
  else if (op == Operator.MULTIPLY) return left * right;
  else if (op == Operator.DIVIDE) return left / right;
  else return right;
}

public class Term {
```

```
48    public enum Operator {
49      ADD, SUBTRACT, MULTIPLY, DIVIDE, BLANK
50    }
51
52    private double value;
53    private Operator operator = Operator.BLANK;
54
55    public Term(double v, Operator op) {
56      value = v;
57      operator = op;
58    }
59
60    public double getNumber() { return value; }
61    public Operator getOperator() { return operator; }
62    public void setNumber(double v) { value = v; }
63
64    /* 将四则运算解析为一组项(Term)。例如，3-5*6 被解析为
65     * [{BLANK,3}, {SUBTRACT, 5}, {MULTIPLY, 6}]。
66     * 如果发现非法格式则返回 null */
67    public static ArrayList<Term> parseTermSequence(String sequence) {
68      /* 代码详见下载附件 */
69    }
70  }
```

该算法花费的时间为 $O(N)$，其中 N 为原始字符串的长度。

解法 2

我们也可以通过两个栈的方法解决该问题：一个数字栈与一个运算符栈。

```
2 - 6 - 7 * 8 / 2 + 5
```

处理方式如下。

- 每当发现一个数字，就将其加入到 numberStack 栈中。
- 只要运算符的优先级高于当前运算符栈顶部元素的优先级，就将其加入到 operatorStack 栈中。如果表达式 priority(currentOperator) <= priority(operatorStack.top()) 成立，我们则按以下方法“折叠”栈顶元素。
  - 折叠操作：将 numberStack 栈顶的两个元素取出，同时将 operatorStack 栈顶元素取出，并将计算结果加入到 numberStack 栈中。
  - 优先级：加法与减法具有相同的优先级，同时它们的优先级要小于乘法与除法（乘法与除法优先级相同）。

  不断重复该折叠操作直至上述的表达式不再成立。届时，将 currentOperator 加入到 operatorStack 栈中。
- 最后，对栈执行折叠操作。

来看一个例子：2 - 6 - 7 * 8 / 2 + 5。

	action	numberStack	operatorStack
2	numberStack.push(2)	2	[empty]
-	operatorStack.push(-)	2	-
6	numberStack.push(6)	6, 2	-
-	collapseStacks [2 - 6] operatorStack.push(-)	-4 -4	[empty] -
7	numberStack.push(7)	7, -4	-
*****	operatorStack.push(*)	7, -4	*, -
8	numberStack.push(8)	8, 7, -4	*, -
/	collapseStack [7 * 8] numberStack.push(/)	56, -4 56, -4	- /, -
2	numberStack.push(2)	2, 56, -4	/, -
+	collapseStack [56 / 2] collapseStack [-4 - 28] operatorStack.push(+)	28, -4 -32 -32	- [empty] +
5	numberStack.push(5)	5, -32	+
	collapseStack [-32 + 5]	-27	[empty]
	return -27		

下面的代码实现了该算法。

```
1   public enum Operator {
2     ADD, SUBTRACT, MULTIPLY, DIVIDE, BLANK
3   }
4
5   double compute(String sequence) {
6     Stack<Double> numberStack = new Stack<Double>();
7     Stack<Operator> operatorStack = new Stack<Operator>();
8
9     for (int i = 0; i < sequence.length(); i++) {
10      try {
11        /* 获取数字并压栈 */
12        int value = parseNextNumber(sequence, i);
13        numberStack.push((double) value);
14
15        /* 下一运算符 */
16        i += Integer.toString(value).length();
17        if (i >= sequence.length()) {
18          break;
19        }
20
21        /* 获取运算符，按需进行合并，压栈 */
22        Operator op = parseNextOperator(sequence, i);
23        collapseTop(op, numberStack, operatorStack);
24        operatorStack.push(op);
25      } catch (NumberFormatException ex) {
26        return Integer.MIN_VALUE;
27      }
28    }
29
30    /* 最后一次合并项 */
31    collapseTop(Operator.BLANK, numberStack, operatorStack);
32    if (numberStack.size() == 1 && operatorStack.size() == 0) {
33      return numberStack.pop();
34    }
35    return 0;
```

```
}

/* 不断合并顶部项直至 priority(futureTop) > priority(top)。
 * 合并项即将两个数字和一个运算符出栈，并将结果压入到数栈*/
void collapseTop(Operator futureTop, Stack<Double> numberStack,
                 Stack<Operator> operatorStack) {
  while (operatorStack.size() >= 1 && numberStack.size() >= 2) {
    if (priorityOfOperator(futureTop) <=
        priorityOfOperator(operatorStack.peek())) {
      double second = numberStack.pop();
      double first = numberStack.pop();
      Operator op = operatorStack.pop();
      double collapsed = applyOp(first, op, second);
      numberStack.push(collapsed);
    } else {
      break;
    }
  }
}

/*返回运算符的优先级，即加法 == 减法 < 乘法 == 除法 */
int priorityOfOperator(Operator op) {
  switch (op) {
      case ADD: return 1;
      case SUBTRACT: return 1;
      case MULTIPLY: return 2;
      case DIVIDE: return 2;
      case BLANK: return 0;
  }
  return 0;
}

/* 对运算符进行计算：left [op] right */
double applyOp(double left, Operator op, double right) {
  if (op == Operator.ADD) return left + right;
  else if (op == Operator.SUBTRACT) return left - right;
  else if (op == Operator.MULTIPLY) return left * right;
  else if (op == Operator.DIVIDE) return left / right;
  else return right;
}

/* 返回指定位移处的数字 */
int parseNextNumber(String seq, int offset) {
  StringBuilder sb = new StringBuilder();
  while (offset < seq.length() && Character.isDigit(seq.charAt(offset))) {
    sb.append(seq.charAt(offset));
    offset++;
  }
  return Integer.parseInt(sb.toString());
}

/* 返回指定位移处的运算符 */
Operator parseNextOperator(String sequence, int offset) {
  if (offset < sequence.length()) {
    char op = sequence.charAt(offset);
    switch(op) {
      case '+': return Operator.ADD;
      case '-': return Operator.SUBTRACT;
      case '*': return Operator.MULTIPLY;
      case '/': return Operator.DIVIDE;
    }
```

```
97    }
98    return Operator.BLANK;
99  }
```

该算法花费的时间为 $O(N)$，其中 N 为原始字符串的长度。

这个解决方案涉及很多恼人的字符串解析代码。请记住，在面试中编写出所有这些代码细节并没有那么重要。事实上，面试官甚至可能会让你假设表达式在传入时已经被提前解析为某种数据结构。

从一开始就请注意使代码模块化，并将代码中单调乏味或不太有趣的部分“外包”到其他函数中。你应该专心完成算法的核心计算功能，而其余的细节可以等有时间再来实现。

第 17 章

高难度题

17.1 洗牌

设计一个用来洗牌的函数。要求做到完美洗牌，也就是说，这副牌 52! 种排列组合出现的概率相同。假设给定一个完美的随机数发生器。

题目解法

这是一个非常有名的面试题，也是众所周知的算法。如果你恰巧对该算法一无所知，那么请继续读下去吧。

让我们想象一下 n 元数组，假设它如下所示。

```
[1] [2] [3] [4] [5]
```

使用简单构造法，我们可以问自己这样一个问题：假设有一个处理 $n-1$ 张牌的方法 `shuffle(...)`，我们可以用这个来洗 n 张牌吗？

当然可以。事实上，这很简单。我们先洗前 $n-1$ 张牌，然后取第 n 张牌，再将它与数组中的一张牌随机交换。就是这样操作。

这个算法递归实现方法如下。

```
1  /* lower 和 higher（含）之间的随机数 */
2  int rand(int lower, int higher) {
3    return lower + (int)(Math.random() * (higher - lower + 1));
4  }
5
6  int[] shuffleArrayRecursively(int[] cards, int i) {
7    if (i == 0) return cards;
8
9    shuffleArrayRecursively(cards, i - 1); // 打乱先前部分的次数
10   int k = rand(0, i); // 随机挑选索引进行交换
11
12   /* 交换元素 k 和 i */
13   int temp = cards[k];
14   cards[k] = cards[i];
15   cards[i] = temp;
16
17   /* 返回元素次序被打乱的数组 */
18   return cards;
19 }
```

这个算法若用迭代法实现，该怎么做呢？让我们思考一下。所要做的是，在数组中进行迭代，对于每个元素 i，将 `array[i]` 与 0 和 i 之间的一个随机元素进行交换。

迭代实现方法实际上是一种非常干净的算法，如下所示。

```
void shuffleArrayIteratively(int[] cards) {
  for (int i = 0; i < cards.length; i++) {
    int k = rand(0, i);
    int temp = cards[k];
    cards[k] = cards[i];
    cards[i] = temp;
  }
}
```

通常，我们看到这个算法是通过迭代法实现的。

17.2 随机集合

编写一个方法，从大小为 n 的数组中随机选出 m 个整数。要求每个元素被选中的概率相同。

题目解法

就像上一个问题一样，我们可以使用简单构造法来递归地解决该问题。

假设我们有一个算法可以从大小为 $n-1$ 的数组中随机抽取 m 个元素，那么如何使用这个算法从一个大小为 n 的数组中随机抽取 m 个元素呢?

我们可以先从前 $n-1$ 个元素中随机抽取一个大小为 m 的集合。然后，只需要确定是否应该将 array[n]插入到子集中（该过程需要从子集中抽取一个随机元素）。一种简单的方法就是从 0 到 n 中随机选取数字 k，如果 $k<m$，那么将 array[n]插入到 subset[k]中。对于将 array[n]以一定的概率插入子集中，或者适当从子集中删除一个随机元素，这样做都是可取的。

该递归算法的伪代码如下所示。

```
int[] pickMRecursively(int[] original, int m, int i) {
  if (i + 1 == m) { // 终止条件
    /* 返回 original 数组的前 m 个元素 */
  } else if (i + 1 > m) {
    int[] subset = pickMRecursively(original, m, i - 1);
    int k = random value between 0 and i, inclusive
    if (k < m) {
      subset[k] = original[i];
    }
    return subset;
  }
  return null;
}
```

用迭代法写代码会更简洁。对于此方法，我们初始化一个 subset 数组作为 original 的前 m 个元素。然后从元素 m 开始遍历数组，每当 $k<m$ 时，就将 array[i]随机插入到 subset 的位置 k。

```
int[] pickMIteratively(int[] original, int m) {
  int[] subset = new int[m];

  /* 用 original 数组的前 m 个元素填入 subset */
  for (int i = 0; i < m ; i++) {
    subset[i] = original[i];
  }

  /* 访问 original 数组的剩余元素 */
  for (int i = m; i < original.length; i++) {
    int k = rand(0, i); // 取得 0 到 i（含）之间的随机数
    if (k < m) {
      subset[k] = original[i];
```

```
    }
  }

  return subset;
}
```

这两种解法无疑都与对数组进行洗牌操作的算法大同小异。

17.3　婴儿名字

每年，政府都会公布一万个最常见的婴儿名字和它们出现的频率，也就是同名婴儿的数量。有些名字有多种拼法，例如，John 和 Jon 本质上是相同的名字，但被当成了两个名字公布出来。给定两个列表，一个是名字及对应的频率，另一个是本质相同的名字对。设计一个算法打印出每个真实名字的实际频率。注意，如果 John 和 Jon 是相同的，并且 Jon 和 Johnny 相同，则 John 与 Johnny 也相同，即它们有传递和对称性。在结果列表中，任选一个名字做为真实名字就可以。

示例：

输入：

Names: John(15)、Jon(12)、Chris(13)、Kris(4)、Christopher(19)

Synonyms: (Jon, John)、(John, Johnny)、(Chris, Kris)、(Chris, Christopher)

输出：John(27)、Kris(36)

题目解法

让我们先找到一个好例子。该例子需要包含一些同义名字和一些无同义名字。此外，其同义名字列表要具有多样性，有些名字可以列在左边，有些名字则列在右边。例如，创建一个包含 John、Jonathan、Jon 和 Johnny 的分组时，不要总是把 Johnny 列在左边。

下面这个列表应该能满足题目要求。

名　字	计　数
John	10
Jon	3
Davis	2
Kari	3
Johnny	11
Carlton	8
Carleton	2
Jonathan	9
Carrie	5

名　字	等 同 于
Jonathan	John
Jon	Johnny
Johnny	John
Kari	Carrie
Carleton	Carlton

最后的名字列表应该是：John (33)，Kari (8)，Davis(2)，Carleton (10)。

解法 1

假设以散列表的形式列出一个婴儿名字列表（如果不是，也很容易创建一个散列表）。

可以从同义名字列表开始读取名字对。读取名字对(Jonathan, John)的时候，可以把 Jonathan 和 John 这对名字合并在一起。但是，需要记住看到过的那对名字，因为将来可能会发现 Jonathan 也等同于其他名字。

我们可以使用一个散列表（`L1`），使其从一个名字映射到它所对应的“真实”名字。还需要知道，对于给定的“真实”名字，所有的名字都等同于该名字。该信息将存储在散列表 `L2` 中。

注意，L2 的作用是反向查找 L1。

```
READ (Jonathan, John)
    L1.ADD Jonathan -> John
    L2.ADD John -> Jonathan
READ (Jon, Johnny)
    L1.ADD Jon -> Johnny
    L2.ADD Johnny -> Jon
READ (Johnny, John)
    L1.ADD Johnny -> John
    L1.UPDATE Jon -> John
    L2.UPDATE John -> Jonathan, Johnny, Jon
```

比如说，如果我们后来发现 John 等同于 Jonny，则需要在 L1 和 L2 中查找并合并所有与之相同的名字。

该方法可行，但要跟踪这两个列表则过于复杂。

另外一种办法是，可以把这些名字看作“等同物类”。当我们找到名字对(Jonathan, John)时，可以将它们放入相同的集合（或等价类）中。每个名字都映射到它的等同物类，而集合中的所有项目都映射到相同的集合实例上。

如果需要合并两个集合，那么将一个集合复制到另一个集合中，并更新散列表使其指向新的集合。

```
READ (Jonathan, John)
    CREATE Set1 = Jonathan, John
    L1.ADD Jonathan -> Set1
    L1.ADD John -> Set1
READ (Jon, Johnny)
    CREATE Set2 = Jon, Johnny
    L1.ADD Jon -> Set2
    L1.ADD Johnny -> Set2
READ (Johnny, John)
    COPY Set2 into Set1.
        Set1 = Jonathan, John, Jon, Johnny
    L1.UPDATE Jon -> Set1
    L1.UPDATE Johnny -> Set1
```

在上面的最后一步中，我们遍历了 Set2 中的所有项，并更新每一项的引用，使其指向 Set1。当这样做的时候，我们一直跟踪名字的总频率。

```
1  HashMap<String, Integer> trulyMostPopular(HashMap<String, Integer> names,
2                                            String[][] synonyms) {
3    /* 解析链表并初始化相同的类别 */
4    HashMap<String, NameSet> groups = constructGroups(names);
5
6    /* 合并相同类别 */
7    mergeClasses(groups, synonyms);
8
9    /* 转换为散列表 */
10   return convertToMap(groups);
11 }
12
13 /* 此部分是算法的核心。检查每组值，合并相同的类别并将第二个类别映射到第一个集合上 */
14 void mergeClasses(HashMap<String, NameSet> groups, String[][] synonyms) {
15   for (String[] entry : synonyms) {
16     String name1 = entry[0];
17     String name2 = entry[1];
18     NameSet set1 = groups.get(name1);
19     NameSet set2 = groups.get(name2);
```

```
    if (set1 != set2) {
      /* 将较小的集合合并至较大的集合 */
      NameSet smaller = set2.size() < set1.size() ? set2 : set1;
      NameSet bigger = set2.size() < set1.size() ? set1 : set2;

      /* 合并链表 */
      Set<String> otherNames = smaller.getNames();
      int frequency = smaller.getFrequency();
      bigger.copyNamesWithFrequency(otherNames, frequency);

      /* 更新映射 */
      for (String name : otherNames) {
        groups.put(name, bigger);
      }
    }
  }
}

/* 遍历(姓名, 频率)组合，并初始化一个从姓名到 NameSets 的映射 */
HashMap<String, NameSet> constructGroups(HashMap<String, Integer> names) {
  HashMap<String, NameSet> groups = new HashMap<String, NameSet>();
  for (Entry<String, Integer> entry : names.entrySet()) {
    String name = entry.getKey();
    int frequency = entry.getValue();
    NameSet group = new NameSet(name, frequency);
    groups.put(name, group);
  }
  return groups;
}

HashMap<String, Integer> convertToMap(HashMap<String, NameSet> groups) {
  HashMap<String, Integer> list = new HashMap<String, Integer>();
  for (NameSet group : groups.values()) {
    list.put(group.getRootName(), group.getFrequency());
  }
  return list;
}

public class NameSet {
  private Set<String> names = new HashSet<String>();
  private int frequency = 0;
  private String rootName;

  public NameSet(String name, int freq) {
    names.add(name);
    frequency = freq;
    rootName = name;
  }

  public void copyNamesWithFrequency(Set<String> more, int freq) {
    names.addAll(more);
    frequency += freq;
  }

  public Set<String> getNames() { return names; }
  public String getRootName() { return rootName; }
  public int getFrequency() { return frequency; }
  public int size() { return names.size(); }
}
```

这个算法的时间复杂度分析起来有点棘手。一种思考方式是考虑其最坏的情况究竟是什么。

对于这个算法，最坏的情况是所有的名字都相同，我们必须不断地合并所有集合。同样，对于最坏的情况，应尽量以最糟糕的方式进行合并，即重复合并成对的集合。每次合并都需要将一个集合中的元素复制到现有集合中，并更新这些项指向的对象。当集合变大时，该操作会越来越慢。

如果你注意一下归并排序的并行过程（你必须将单元素数组合并为 2 个元素的数组，然后将 2 个元素的数组合并为 4 个元素的数组，直到最后有一个完整的数组），可能会发现其时间复杂度是 $O(N \log N)$，的确如此。

如果你没有注意到该并行过程，那么还有另一种思考方法。

假设我们有名字(a, b, c, d, ..., z)。在最坏情况下，首先将相同的项目合并，即(a, b), (c, d), (e, f), ..., (y, z)。然后将它们合并成(a, b, c, d), (e, f, g, h), ..., (w, x, y, z)。继续合并，直到只剩下一个类为止。

在合并集合的过程中，每一步“扫描”操作，一半的项目被移动到一个新的集合中。因此每一步“扫描”操作花费的时间为 $O(N)$（需要合并的集合会越来越少，但每一个集合大小都会变大）。

我们需要完成多少次“扫描”操作？在每一次扫描中，我们获得集合的数量是之前的一半。因此，需要完成 $O(\log N)$次“扫描”。

由于需要完成 $O(\log N)$次扫描，每次扫描操作需要 $O(N)$的工作量，因此总运行时间是 $O(N \log N)$。

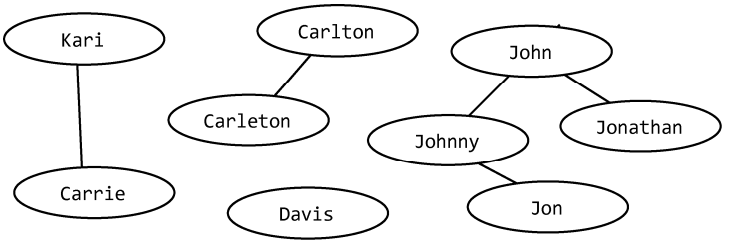
该解法很好，但是让我们看看能不能更快一些。

解法 2：优化解法

为了优化以上解法，我们要想一想该解法究竟为何运行缓慢。根本问题在于指针的合并和更新。

如果不对指针执行合并和更新操作，会怎么样呢？如果仅仅标记了两个名称之间存在等同关系，但实际上并没有对这些信息做任何操作，会怎么样？

在这种情况下，我们其实构建了一个图。



现在该怎么办？从视觉上看，这似乎很容易。每个连通部分都是一组等同物的名称。我们只需要根据连通部分将名字分组，将同一组名字的频率相加，然后从每组中返回任意一个选择的名字即可。

在实践中，我们应该如何操作？可以选择一个名称，并对一个连通部分进行深度优先（或广度优先）搜索，以得出所有名字频率的和。必须确保每个连通部分只访问一次。这很容易实现，只需将一个节点在被发现后标记为 visited，并且只搜索 visited 为 false 的节点。

```
1  HashMap<String, Integer> trulyMostPopular(HashMap<String, Integer> names,
2                                            String[][] synonyms) {
3    /* 创建数据 */
4    Graph graph = constructGraph(names);
5    connectEdges(graph, synonyms);
```

```
6
7     /* 寻找连通部分 */
8     HashMap<String, Integer> rootNames = getTrueFrequencies(graph);
9     return rootNames;
10  }
11
12  /* 将所有姓名以节点的形式加入到图中 */
13  Graph constructGraph(HashMap<String, Integer> names) {
14    Graph graph = new Graph();
15    for (Entry<String, Integer> entry : names.entrySet()) {
16        String name = entry.getKey();
17        int frequency = entry.getValue();
18        graph.createNode(name, frequency);
19    }
20    return graph;
21  }
22
23  /* 连接相似拼写法 */
24  void connectEdges(Graph graph, String[][] synonyms) {
25    for (String[] entry : synonyms) {
26        String name1 = entry[0];
27        String name2 = entry[1];
28        graph.addEdge(name1, name2);
29    }
30  }
31
32  /* 对每一个连通部分进行深度优先搜索。如果一个节点被访问过，则其已经被计算过 */
33  HashMap<String, Integer> getTrueFrequencies(Graph graph) {
34    HashMap<String, Integer> rootNames = new HashMap<String, Integer>();
35    for (GraphNode node : graph.getNodes()) {
36      if (!node.isVisited()) { // 已访问这个连通部分
37        int frequency = getComponentFrequency(node);
38        String name = node.getName();
39        rootNames.put(name, frequency);
40      }
41    }
42    return rootNames;
43  }
44
45  /* 通过深度优先搜索计算总频率并标记已访问 */
46  int getComponentFrequency(GraphNode node) {
47    if (node.isVisited()) return 0; // 已访问
48
49    node.setIsVisited(true);
50    int sum = node.getFrequency();
51    for (GraphNode child : node.getNeighbors()) {
52      sum += getComponentFrequency(child);
53    }
54    return sum;
55  }
56
57  /* GraphNode 和 Graph 的代码无须多做解释，也可以在下载的解答中找到具体代码 */
```

为了分析效率，我们可以分别探讨该算法每个部分的效率。

- 读取数据与数据的大小是线性相关的，所以需要 $O(B+P)$的时间，其中 B 是婴儿名字的数量，P 是同义名字的对数。这是因为我们只对每个输入数据完成常数项数量的计算。
- 为了计算频率，每条边在所有的图形搜索中都被“经过”一次，并且每一个节点都被“经过”一次，以确定其是否被访问过。这部分的时间复杂度是 $O(B+P)$。

因此，算法运行的总时间是 $O(B + P)$。我们至少要在 $B + P$ 的数据中读取，因此，不能得到比这更优的算法了。

17.4 马戏团人塔

有个马戏团正在设计叠罗汉的表演节目，一个人要站在另一人的肩膀上。出于实际和美观的考虑，在上面的人要比下面的人矮一点且轻一点。已知马戏团每个人的身高和体重，请编写代码计算叠罗汉最多能叠几个人。

示例：

输入：(ht, wt)：(65，100) (70，150) (56，90) (75，190) (60，95) (68，110)

输出：从上往下数，叠罗汉最多能叠 6 层：(56，90) (60,95) (65,100) (68,110) (70,150) (75,190)

题目解法

当我们把所有关于这个问题上的“细枝末节”都排除后，对这个问题理解如下。

给定一组项目对，找出最长的序列，使其第一和第二项都保持非递减顺序。

我们可能首先尝试以某一项对所有元素进行排序。这实际上有所助益，但并不能直接找到答案。

对元素以高度排序，会得到一个所有元素应该出现的相对顺序。不过，仍然需要找到以重量排序的最长递增子序列。

解法 1：递归法

一种方法是尝试所有的可能性。以高度进行排序后，遍历数组。对于每一个元素，我们将分为两个选择：将这个元素添加到子序列（如果情况有效），或不添加。

```
1  ArrayList<HtWt> longestIncreasingSeq(ArrayList<HtWt> items) {
2    Collections.sort(items);
3    return bestSeqAtIndex(items, new ArrayList<HtWt>(), 0);
4  }
5
6  ArrayList<HtWt> bestSeqAtIndex(ArrayList<HtWt> array, ArrayList<HtWt> sequence,
7                                 int index) {
8    if (index >= array.size()) return sequence;
9
10   HtWt value = array.get(index);
11
12   ArrayList<HtWt> bestWith = null;
13   if (canAppend(sequence, value)) {
14     ArrayList<HtWt> sequenceWith = (ArrayList<HtWt>) sequence.clone();
15     sequenceWith.add(value);
16     bestWith = bestSeqAtIndex(array, sequenceWith, index + 1);
17   }
18
19   ArrayList<HtWt> bestWithout = bestSeqAtIndex(array, sequence, index + 1);
20   return max(bestWith, bestWithout);
21 }
22
23 boolean canAppend(ArrayList<HtWt> solution, HtWt value) {
24   if (solution == null) return false;
25   if (solution.size() == 0) return true;
26
27   HtWt last = solution.get(solution.size() - 1);
```

```
  return last.isBefore(value);
}

ArrayList<HtWt> max(ArrayList<HtWt> seq1, ArrayList<HtWt> seq2) {
  if (seq1 == null) {
    return seq2;
  } else if (seq2 == null) {
    return seq1;
  }
  return seq1.size() > seq2.size() ? seq1 : seq2;
}

public class HtWt implements Comparable<HtWt> {
  private int height;
  private int weight;
  public HtWt(int h, int w) { height = h; weight = w; }

  public int compareTo(HtWt second) {
    if (this.height != second.height) {
      return ((Integer)this.height).compareTo(second.height);
    } else {
      return ((Integer)this.weight).compareTo(second.weight);
    }
  }

  /* 如果该实例需要被置于 other 之前，那么返回 true。请注意，this.isBefore(other)和
   * other.isBefore(this) 同时返回 false 是有可能的。这与 compareTo 方法不同，
   * 在 compareTo 方法中，如果 a < b，则一定 b > a */
  public boolean isBefore(HtWt other) {
    if (height < other.height && weight < other.weight) {
      return true;
    } else {
      return false;
    }
  }
}
```

这个算法将花费 $O(2^n)$的时间。我们可以使用保存记录的方法（即缓存最好的序列）来优化该算法。

还有一种更整洁的方法。

解法 2：迭代法

假设我们已经分别找到从 A[0]到 A[3]所有元素结尾的最长子序列，可以用这些信息来找到终止于 A[4]的最长子序列吗？

```
数组: 13, 14, 10, 11, 12
Longest(以 A[0]结尾): 13
Longest (以 A[1]结尾): 13, 14
Longest (以 A[2]结尾): 10
Longest (以 A[3]结尾): 10, 11
Longest (以 A[4]结尾): 10, 11, 12
```

可以的。只需要将 A[4]附加到可能的最长子序列上。

该算法实现起来相当简单。

```
ArrayList<HtWt> longestIncreasingSeq(ArrayList<HtWt> array) {
  Collections.sort(array);

  ArrayList<ArrayList<HtWt>> solutions = new ArrayList<ArrayList<HtWt>>();
  ArrayList<HtWt> bestSequence = null;
```

```
6
7     /* 计算在每个元素截止的最长序列。跟踪记录总体上的最长序列 */
8     for (int i = 0; i < array.size(); i++) {
9       ArrayList<HtWt> longestAtIndex = bestSeqAtIndex(array, solutions, i);
10      solutions.add(i, longestAtIndex);
11      bestSequence = max(bestSequence, longestAtIndex);
12    }
13
14    return bestSequence;
15  }
16
17  /* 计算在每个元素截止的最长序列 */
18  ArrayList<HtWt> bestSeqAtIndex(ArrayList<HtWt> array,
19      ArrayList<ArrayList<HtWt>> solutions, int index) {
20    HtWt value = array.get(index);
21
22    ArrayList<HtWt> bestSequence = new ArrayList<HtWt>();
23
24    /* 寻找我们可以连接该元素的最长序列 */
25    for (int i = 0; i < index; i++) {
26      ArrayList<HtWt> solution = solutions.get(i);
27      if (canAppend(solution, value)) {
28        bestSequence = max(solution, bestSequence);
29      }
30    }
31
32    /* 在尾部增添该元素 */
33    ArrayList<HtWt> best = (ArrayList<HtWt>) bestSequence.clone();
34    best.add(value);
35
36    return best;
37  }
```

该算法在 $O(n^2)$的时间复杂度内运行。确实存在一个 $O(n \log(n))$的算法，但它要复杂得多，而且即使有所助益，你也不太可能在一场面试中推导出来。但是，如果你乐于探索该解法，不妨在网上搜索一下，即可找到关于此解法的多种解释。

17.5 第 *k* 个数

有些数的素因子只有 3，5，7，请设计一个算法找出第 *k* 个数。注意，不是必须有这些素因子，而是必须不包含其他的素因子。例如，前几个数按顺序应该是 1，3，5，7，9，15，21。

题目解法

先明确此题题干，即满足 $3^a \times 5^b \times 7^c$这一形式的第 k 小的值。先试试蛮力法。

1. 蛮力法

我们知道这个 k 的最大值可以是 $3^k \times 5^k \times 7^k$。因此，一个笨方法是，将 a、b 和 c 赋值为 0 和 k 之间的所有可能元素，并计算出 $3^a \times 5^b \times 7^c$的值。我们可以把所有的结果全部放入一个列表，对列表进行排序，然后选择第 k 小的值。

```
1   int getKthMagicNumber(int k) {
2     ArrayList<Integer> possibilities = allPossibleKFactors(k);
3     Collections.sort(possibilities);
4     return possibilities.get(k);
5   }
6
```

```
ArrayList<Integer> allPossibleKFactors(int k) {
  ArrayList<Integer> values = new ArrayList<Integer>();
  for (int a = 0; a <= k; a++) { // 3 的循环
    int powA = (int) Math.pow(3, a);
    for (int b = 0; b <= k; b++) { // 5 的循环
      int powB = (int) Math.pow(5, b);
      for (int c = 0; c <= k; c++) { // 7 的循环
        int powC = (int) Math.pow(7, c);
        int value = powA * powB * powC;

        /* 检查溢出 */
        if (value < 0 || powA == Integer.MAX_VALUE ||
           powB == Integer.MAX_VALUE ||
           powC == Integer.MAX_VALUE) {
          value = Integer.MAX_VALUE;
        }
        values.add(value);
      }
    }
  }
  return values;
}
```

这个方法的时间复杂度是多少？我们嵌套了 `for` 循环，每个循环都进行 k 次迭代。`allPossibleKFactors` 的运行时间是 $O(k^3)$。然后，将 k^3 个结果排序需要 $O(k^3 \log (k^3))$的时间（相当于 $O(k^3 \log k)$）。最终得出的时间复杂度为 $O(k^3 \log k)$。

你可以对此做一些优化，并能较好地解决整数溢出这一问题，但老实说，这个算法相当慢。与其这样，不如将精力放在重新设计一个算法上。

2. 改进解法

让我们想象一下所得结果是什么，如下所示。

1	-	$3^0 * 5^0 * 7^0$
3	3	$3^1 * 5^0 * 7^0$
5	5	$3^0 * 5^1 * 7^0$
7	7	$3^0 * 5^0 * 7^1$
9	3*3	$3^2 * 5^0 * 7^0$
15	3*5	$3^1 * 5^1 * 7^0$
21	3*7	$3^1 * 5^0 * 7^1$
25	5*5	$3^0 * 52 * 7^0$
27	3*9	$3^3 * 5^0 * 7^0$
35	5*7	$3^0 * 5^1 * 7^1$
45	5*9	$3^2 * 5^1 * 7^0$
49	7*7	$3^0 * 5^0 * 7^2$
63	3*21	$3^2 * 5^0 * 7^1$

问题在于列表中的下一个值是什么？下一个值将是下列三者之一：

- 3 ×(数字列表中的某值)；
- 5 ×(数字列表中的某值)；
- 7 ×(数字列表中的某值)。

如果你没能立即看出其中缘由，可以这样想：无论下一个值是多少（我们称之为 nv），我们将其除以 3。这个数字出现过了吗？只要 nv 的因数有 3，那么答案就是肯定的。除以 5 和除以 7 是一样的。

因此，我们知道 A_k 可以表示为(3、5 或 7) × ($\{A_1, \cdots, A_{k-1}\}$中的某值)。我们还知道，根据定义，A_k 是列表中的下一个数字。因此，A_k 将是最小的“新”数字（$\{A_1, \cdots, A_{k-1}\}$中已经存在的数字），它可以通过将列表中的每个值乘以 3、5 或 7 来生成。

怎么才能找到 A_k？实际上，可以把列表中的每个数乘以 3、5 和 7，然后找到最小的还没有被添加到列表中的元素。这个解法的时间复杂度是 $O(k^2)$。该解法不错，不过我认为还可以做得更好。

与试图从列表中“取出”一个存在元素（通过将它们全部乘以 3、5 和 7）来计算 A_k 不同的是，我们可以考虑通过列表中存在的值“压入”三个后续值，也就是说，每个数字 A_i 最终将会以下列形式出现在列表中。

- $3 \times A_i$
- $5 \times A_i$
- $7 \times A_i$

我们可以利用这一思路提前进行规划。每当在列表中添加一个数字 A_i 时，就会在某个临时列表中存入 $3A_i$、$5A_i$ 和 $7A_i$。为了生成 A_{i+1}，我们通过这个临时列表查找最小值。

该代码大致如下所示。

```
int removeMin(Queue<Integer> q) {
  int min = q.peek();
  for (Integer v : q) {
    if (min > v) {
      min = v;
    }
  }
  while (q.contains(min)) {
    q.remove(min);
  }
  return min;
}

void addProducts(Queue<Integer> q, int v) {
  q.add(v * 3);
  q.add(v * 5);
  q.add(v * 7);
}

int getKthMagicNumber(int k) {
  if (k < 0) return 0;

  int val = 1;
  Queue<Integer> q = new LinkedList<Integer>();
  addProducts(q, 1);
  for (int i = 0; i < k; i++) {
    val = removeMin(q);
    addProducts(q, val);
  }
  return val;
}
```

这个算法肯定比第一个算法要好得多，但仍然不够完美。

3. 最优解法

为了生成一个新的元素 A_i，我们需要搜索一个链表，其中每个元素是下列三者之一：

- $3 \times$(列表中前面的某值)；

- 5 ×(列表中前面的某值);
- 7 ×(列表中前面的某值)。

可以优化哪些不必要的操作?

想象一下如下列表。

q_6 = {$7A_1$, $5A_2$, $7A_2$, $7A_3$, $3A_4$, $5A_4$, $7A_4$, $5A_5$, $7A_5$}

当搜索这个列表时，检查 $7A_1$ 是否小于 min，然后再检查 $7A_5$ 是否小于 min。这似乎有点不知变通，不是吗？已知 $A_1 < A_5$，所以只需检查 $7A_1$。

如果从一开始就把列表和常数因子分开，那么只需要检查 3、5、7 的乘积中的第一个即可。所有后续元素都将大于第一个乘积，也就是说，上面的列表应如下所示。

Q_{36} = {$3A_4$}
Q_{56} = {$5A_2$, $5A_4$, $5A_5$}
Q_{76} = {$7A_1$, $7A_2$, $7A_3$, $7A_4$, $7A_5$}

为了得到最小值，我们只需要看每个队列最前面的元素，如下所示。

```
y = min(Q3.head(),Q5.head(),Q7.head())
```

一旦计算 y，需要将 $3y$ 插入到 $Q3$, $5y$ 插入到 $Q5$, $7y$ 插入到 $Q7$。但是，只有当这些元素不在另一个列表中时，才可进行插入操作。

为什么像 $3y$ 这样的数字会有可能已经存在于等待队列中？这是因为，如果 y 来自于 $Q7$，那么意味着 $y = 7x$，其中 x 是较小的数字。如果 $7x$ 是最小值，那么我们一定已经在队列中遇到过 $3x$。看到 $3x$ 时，做了些什么呢？我们将 $7 \times 3x$ 插入到了 $Q7$ 当中。注意，$7 \times 3x = 3 \times 7x = 3y$。

换句话说，如果从 $Q7$ 中提取一个元素，那么该元素应形如 $7 \times \text{suffix}$，此时，我们已经处理了 $3 \times \text{suffix}$ 和 $5 \times \text{suffix}$ 这两个元素。在处理 $3 \times \text{suffix}$ 时，已经将 $7 \times 3 \times \text{suffix}$ 插入到 $Q7$ 中。在处理 $5 \times \text{suffix}$ 时，已经在 $Q7$ 中插入了 $7 \times 5 \times \text{suffix}$。我们唯一还没有看到的值是 $7 \times 7 \times \text{suffix}$，因此，只需在 $Q7$ 中插入 $7 \times 7 \times \text{suffix}$ 即可。

下面举个例子进一步阐明这一点。

```
initialize:
        Q3 = 3
        Q5 = 5
        Q7 = 7
remove min = 3. insert 3*3 in Q3, 5*3 into Q5, 7*3 into Q7.
        Q3 = 3*3
        Q5 = 5, 5*3
        Q7 = 7, 7*3
remove min = 5. 3*5 is a dup, since we already did 5*3. insert 5*5 into Q5, 7*5 into Q7.
        Q3 = 3*3
        Q5 = 5*3, 5*5
        Q7 = 7, 7*3, 7*5.
remove min = 7. 3*7 and 5*7 are dups, since we already did 7*3 and 7*5. insert 7*7 into Q7.
        Q3 = 3*3
        Q5 = 5*3, 5*5
        Q7 = 7*3, 7*5, 7*7
remove min = 3*3 = 9. insert 3*3*3 in Q3, 3*3*5 into Q5, 3*3*7 into Q7.
        Q3 = 3*3*3
        Q5 = 5*3, 5*5, 5*3*3
        Q7 = 7*3, 7*5, 7*7, 7*3*3
remove min = 5*3 = 15. 3*(5*3) is a dup, since we already did 5*(3*3). insert
5*5*3 in Q5, 7*5*3 into Q7.
        Q3 = 3*3*3
        Q5 = 5*5, 5*3*3, 5*5*3
```

```
        Q7 = 7*3, 7*5, 7*7, 7*3*3, 7*5*3
remove min = 7*3 = 21. 3*(7*3) and 5*(7*3) are dups, since we already did 7*(3*3)
and 7*(5*3). insert 7*7*3 into Q7.
        Q3 = 3*3*3
        Q5 = 5*5, 5*3*3, 5*5*3
        Q7 = 7*5, 7*7, 7*3*3, 7*5*3, 7*7*3
```

该问题的伪代码如下所示。

(1) 初始化 array 和 $Q3$、$Q5$、$Q7$ 队列。

(2) 将 1 插入 array。

(3) 分别将 1×3、1×5 和 1×7 插入到 $Q3$、$Q5$ 和 $Q7$ 中。

(4) 将 x 赋值为 $Q3$、$Q5$、$Q7$ 中的最小元素。将 x 附加到 magic 后。

(5) 如果 x 出现于：

$Q3$：将 $x\times3$、$x\times5$ 和 $x\times7$ 加入到 $Q3$、$Q5$、$Q7$ 尾部。从 $Q3$ 中删除 x。

$Q5$：将 $x\times5$ 和 $x\times7$ 加入到 $Q5$、$Q7$ 尾部。从 $Q5$ 中删除 x。

$Q7$：只将 $x\times7$ 加入到 $Q7$ 尾部。从 $Q7$ 中删除 x。

(6) 重复步骤(4)和步骤(5)，直到找到 k 个元素。

下面的代码实现了这个算法。

```
1  int getKthMagicNumber(int k) {
2    if (k < 0) {
3      return 0;
4    }
5    int val = 0;
6    Queue<Integer> queue3 = new LinkedList<Integer>();
7    Queue<Integer> queue5 = new LinkedList<Integer>();
8    Queue<Integer> queue7 = new LinkedList<Integer>();
9    queue3.add(1);
10
11   /* 从第 0 到 k 次循环 */
12   for (int i = 0; i <= k; i++) {
13     int v3 = queue3.size() > 0 ? queue3.peek() : Integer.MAX_VALUE;
14     int v5 = queue5.size() > 0 ? queue5.peek() : Integer.MAX_VALUE;
15     int v7 = queue7.size() > 0 ? queue7.peek() : Integer.MAX_VALUE;
16     val = Math.min(v3, Math.min(v5, v7));
17     if (val == v3) { // 加入到 3、5、7 的队列中
18       queue3.remove();
19       queue3.add(3 * val);
20       queue5.add(5 * val);
21     } else if (val == v5) { // 加入到 5、7 的队列中
22       queue5.remove();
23       queue5.add(5 * val);
24     } else if (val == v7) { // 加入到 7 的队列中
25       queue7.remove();
26     }
27     queue7.add(7 * val); // 永远都需要加入到 7 的队列中
28   }
29   return val;
30 }
```

看到这个题目，要竭尽全力来解出此题，尽管该题真得很难。你可以先试试蛮力法（该解法富有挑战性，但不是很复杂），然后尝试优化该解法或者试着找到数字的模式。

当你束手无策时，面试官很可能会助你一臂之力。无论你做什么，都不要轻言放弃。大声说出你的想法，讲出疑惑之处，并阐述思考过程。面试官很可能会站出来给你一些提示。

记住，并没有人会期望你在该问题上表现得完美无缺，只会将你的表现与其他候选人作对比从而进行评估。对于一个棘手的问题，每个人都会表现得磕磕绊绊。

17.6　单词距离

有个内含单词的超大文本文件，给定任意两个单词，找出在这个文件中这两个单词的最短距离（相隔单词数）。如果寻找过程在这个文件中会重复多次，而每次寻找的单词不同，你能对此优化吗?

题目解法

在此题中，我们假设单词 word1 和 word2 谁在前谁在后无关紧要，当然最好与面试官确认能否作此假设。

要解决此题，我们只需遍历一次这个文件。在遍历期间，我们会记下最后看见 word1 和 word2 的地方，并把它们的位置存入 location1 和 location2 中。如果当前的位置比已知最优位置更好，则更新已知最优位置。

下面是该算法的实现代码。

```
LocationPair findClosest(String[] words, String word1, String word2) {
  LocationPair best = new LocationPair(-1, -1);
  LocationPair current = new LocationPair(-1, -1);
  for (int i = 0; i < words.length; i++) {
    String word = words[i];
    if (word.equals(word1)) {
      current.location1 = i;
      best.updateWithMin(current);
    } else if (word.equals(word2)) {
      current.location2 = i;
      best.updateWithMin(current); // 如果更短，则更新值
    }
  }
  return best;
}

public class LocationPair {
  public int location1, location2;
  public LocationPair(int first, int second) {
    setLocations(first, second);
  }

  public void setLocations(int first, int second) {
    this.location1 = first;
    this.location2 = second;
  }

  public void setLocations(LocationPair loc) {
    setLocations(loc.location1, loc.location2);
  }

  public int distance() {
    return Math.abs(location1 - location2);
  }

  public boolean isValid() {
    return location1 >= 0 && location2 >= 0;
  }
```

```
39
40    public void updateWithMin(LocationPair loc) {
41      if (!isValid() || loc.distance() < distance()) {
42        setLocations(loc);
43      }
44    }
45  }
```

如果上述代码要被重复调用（查询其他单词对的最短距离），则可以构造一个散列表，记录每个单词及其出现的位置。我们只需读取一次单词列表。在那之后可以使用相似的算法，但是只需要对位置进行迭代即可。

以下面的列表为例。

```
listA: {1, 2, 9, 15, 25}
listB: {4, 10, 19}
```

假设指针 pA 和 pB 指向每个列表的头部。我们的目标是让 pA 和 pB 指向尽可能接近的值。

第一对可能的值是(1, 4)。

我们能找到的下一对值是什么？如果移动 pB，那么距离一定会变大。如果移动 pA，可能会得到更好的一对值。让我们移动 pA。

第二对可能的值是(2, 4)。这比前一对值要好，所以把它记录成最优值。

我们再次移动 pA，得到(9, 4)。这比之前的值要差。

因为 pA 的值大于 pB 的值，我们现在开始移动 pB。得到(9, 10)。

接下来会得到(15, 10)，然后是(15, 19)，再然后是(25, 19)。

可以实现如下所示的算法。

```
1   LocationPair findClosest(String word1, String word2,
2                            HashMapList<String, Integer> locations) {
3     ArrayList<Integer> locations1 = locations.get(word1);
4     ArrayList<Integer> locations2 = locations.get(word2);
5     return findMinDistancePair(locations1, locations2);
6   }
7
8   LocationPair findMinDistancePair(ArrayList<Integer> array1,
9                                    ArrayList<Integer> array2) {
10    if (array1 == null || array2 == null || array1.size() == 0 ||
11        array2.size() == 0) {
12      return null;
13    }
14
15    int index1 = 0;
16    int index2 = 0;
17    LocationPair best = new LocationPair(array1.get(0), array2.get(0));
18    LocationPair current = new LocationPair(array1.get(0), array2.get(0));
19
20    while (index1 < array1.size() && index2 < array2.size()) {
21      current.setLocations(array1.get(index1), array2.get(index2));
22      best.updateWithMin(current); // 如果更短，则更新值
23      if (current.location1 < current.location2) {
24        index1++;
25      } else {
26        index2++;
27      }
28    }
29
30    return best;
31  }
```

```
32
33 /* 预计算 */
34 HashMapList<String, Integer> getWordLocations(String[] words) {
35   HashMapList<String, Integer> locations = new HashMapList<String, Integer>();
36   for (int i = 0; i < words.length; i++) {
37     locations.put(words[i], i);
38   }
39   return locations;
40 }
41
42 /* HashMapList<String, Integer> 是从 String 到
43  * ArrayList<Integer>的散列表。实现细节详见附录 A */
```

该算法的预处理步骤花费 $O(N)$的时间，其中 N 为字符串中单词的数目。

找到最接近的位置将会花费 $O(A + B)$时间，其中 A 是第一个单词出现的次数，B 是第二个单词出现的次数。

17.7　恢复空格

哦，不！你不小心把一个长篇文章中的空格、标点都删掉了，并且大写也弄成了小写。像句子“I reset the computer. It still didn’t boot!”已经变成了“iresetthecomputeritstilldidntboot”。在处理标点符号和大小写之前，你得先把它断成词语。当然了，你有一本厚厚的词典，用一个 string 的集合表示。不过，有些词没在词典里。假设文章用 string 表示，设计一个算法，把文章断开，要求未识别的字符最少。

示例：

输入：jesslookedjustliketimherbrother

输出：jess looked just like tim her brother（7 个未识别的字符）

题目解法

一些面试官喜欢直奔主题，抛出具体的问题。但也有一些人喜欢给你很多不必要的上下文，比如这个问题。在这种情况下，抓住问题的题干至关重要。

在该题目中，核心问题实际上是要找到一种方法来将字符串分解成单个单词，这种方法在解析过程中要尽量少用“省略”字符。

注意，我们并不试图“理解”字符串。我们可以把“thisisawesome”这个字符串解析为“this is a we some”，也可以将其解析为“this is awesome”。

1. 蛮力法

解决这个问题的关键在于找到一种方法来定义解法（即解析字符串）的子问题，其中一种方法是对字符串进行递归操作。

我们首先要做出的选择就是在哪里插入空格。是在第一个字符之后吗？还是第二个字符或者第三个字符之后？

让我们以 thisismikesfavoritefood 这个字符串为例。在哪里插入第一个空格呢？

- 在 t 之后插入 1 个空格，会得到 1 个无效的字符。
- 在 th 之后插入 1 个空格，会得到 2 个无效字符。
- 在 thi 之后插入 1 个空格，会得到 3 个无效字符。
- 在 this 之后插入 1 个空格，会得到 1 个完整的词。此时没有无效字符。
- 在 thisi 之后插入 1 个空格，会得到 5 个无效字符。

- ……以此类推。

在选择第一个插入空格的位置后，可以递归地选择第二个插入空格的位置，然后是第三个插入空格的位置，以此类推，直到处理完这个字符串为止。

在所有这些选项的返回值中，我们选择其中最好的一个（无效字符最少）。

函数应该返回什么？我们既需要在递归路径中使用无效字符的数量，也需要实际的解析结果。因此，只需要使用一个自定义的 `ParseResult` 类返回这两个结果。

```
1  String bestSplit(HashSet<String> dictionary, String sentence) {
2    ParseResult r = split(dictionary, sentence, 0);
3    return r == null ? null : r.parsed;
4  }
5
6  ParseResult split(HashSet<String> dictionary, String sentence, int start) {
7    if (start >= sentence.length()) {
8      return new ParseResult(0, "");
9    }
10
11   int bestInvalid = Integer.MAX_VALUE;
12   String bestParsing = null;
13   String partial = "";
14   int index = start;
15   while (index < sentence.length()) {
16     char c = sentence.charAt(index);
17     partial += c;
18     int invalid = dictionary.contains(partial) ? 0 : partial.length();
19     if (invalid < bestInvalid) { // 短路
20       /* 递归，在此字符后加入一个空格。如果此处比当前最好情况更好，则用此处代替当前最好情况 */
21       ParseResult result = split(dictionary, sentence, index + 1);
22       if (invalid + result.invalid < bestInvalid) {
23         bestInvalid = invalid + result.invalid;
24         bestParsing = partial + " " + result.parsed;
25         if (bestInvalid == 0) break; // 短路
26       }
27     }
28
29     index++;
30   }
31   return new ParseResult(bestInvalid, bestParsing);
32 }
33
34 public class ParseResult {
35   public int invalid = Integer.MAX_VALUE;
36   public String parsed = " ";
37   public ParseResult(int inv, String p) {
38     invalid = inv;
39     parsed = p;
40   }
41 }
```

我们在该解法中实现了两处“短路”操作。

- 第 21 行：如果当前无效字符的数量超过了已知最佳的无效字符数，即知道这条递归路径不是理想路径，那么继续该路径没有意义。
- 第 29 行：如果发现一个路径没有无效字符，即知道没有更好的方案了。不妨使用这条路径。

这个解法的时间复杂度是什么？在实践中很难真正地描述该时间复杂度，因为它取决于所使用的语言（如英语）。

一种算出该时间复杂度的方法是，想象一种奇怪的语言，使用该语言基本上所有的递归路径都会被计算。在这种情况下，我们在每个字符上都做出两种选择。如果有 n 个字符，时间复杂度则为 $O(2^n)$。

2. 优化解法

当有一个指数级时间复杂度的递归算法时，我们通常通过记忆技术（即缓存结果）来优化该算法。为此，我们需要找到共同的子问题。

递归路径在哪里会重叠？也就是说，共同的子问题出现在哪里？

让我们再来想象一下这个字符串 thisismikesfavoritefood。再次假设所有词都是有效词汇。

在本例中，我们尝试在 t 之后插入第一个空格，并尝试在 th（以及许多其他选项）之后插入第一个空格。想想下一个选项是什么。

```
split(thisismikesfavoritefood) ->
       t + split(hisismikesfavoritefood)
    OR th + split(isismikesfavoritefood)
    OR ...

split(hisismikesfavoritefood) ->
       h + split(isismikesfavoritefood)
    OR ...

...
```

在 t 和 h 之后加上一个空格与在 th 之后插入一个空格会有相同的递归路径。当有相同的结果时，计算 split(isismikesfavoritefood)两次是毫无意义的。

我们应该缓存结果。可以通过使用散列表进行缓存，该散列表应从当前子字符串映射到 ParseResult 对象。

实际上，不需要让当前的子字符串成为散列表的键。字符串中的 start 索引足够表示一个子字符串。毕竟，如果我们要使用子字符串，可以调用 sentence.substring(start, sentence.length)。因此，散列表将从一个起始索引映射到从该索引到字符串结束所产生的最佳解析结果。

同时，因为起始索引是散列表的键，所以根本不需要一个真正的散列表。我们使用一个由 ParseResult 对象组成的数组即可。该数组同样可以起到从索引映射到对象的作用。

代码与之前的函数基本相同，但该解法代码中使用了 memo 表（缓存）。第一次调用该函数时，首先查询该表；返回时，设置该表的值。

```
1  String bestSplit(HashSet<String> dictionary, String sentence) {
2    ParseResult[] memo = new ParseResult[sentence.length()];
3    ParseResult r = split(dictionary, sentence, 0, memo);
4    return r == null ? null : r.parsed;
5  }
6
7  ParseResult split(HashSet<String> dictionary, String sentence, int start,
8                    ParseResult[] memo) {
9    if (start >= sentence.length()) {
10     return new ParseResult(0, "");
11   } if (memo[start] != null) {
12     return memo[start];
13   }
14
15   int bestInvalid = Integer.MAX_VALUE;
16   String bestParsing = null;
17   String partial = "";
18   int index = start;
```

```
  while (index < sentence.length()) {
    char c = sentence.charAt(index);
    partial += c;
    int invalid = dictionary.contains(partial) ? 0 : partial.length();
    if (invalid < bestInvalid) { // 短路
      /* 递归，在此字符后加入一个空格。如果此处比当前最好情况更好，则用此处代替当前最好情况 */
      ParseResult result = split(dictionary, sentence, index + 1, memo);
      if (invalid + result.invalid < bestInvalid) {
        bestInvalid = invalid + result.invalid;
        bestParsing = partial + " " + result.parsed;
        if (bestInvalid == 0) break; // 短路
      }
    }

    index++;
  }
  memo[start] = new ParseResult(bestInvalid, bestParsing);
  return memo[start];
}
```

理解该解法的时间复杂度比之前的解法更加棘手。再想象一个非常奇特的例子。在这个例子中，所有的单词看起来都是一个有效的单词。

可行的一种方法是，认识到`split(i)`只会对每个 i 的值进行一次计算。假设我们已经通过`split(n - 1)`调用了`split(i+1)`，当调用`split(i)`时会发生什么？

```
split(i) -> calls:
    split(i + 1)
    split(i + 2)
    split(i + 3)
    split(i + 4)
    ...
    split(n - 1)
```

每个递归调用都已经计算过了，所以它们会立即返回。做 $n-i$ 次时间复杂度为 $O(1)$的调用将总共花费 $O(n-i)$的时间。这意味着，`split(i)`将最多花费 $O(i)$的时间。

我们现在可以将同一逻辑方法应用于`split(i - 1)`、`split(i - 2)`等调用中去。如果我们调用一次`split(n - 1)`，调用两次`split(n - 2)`，调用三次`split(n - 3)`，……调用 n 次`split(0)`，那么总共会有多少次调用？这其实是从 1 到 n 所有数的和，即 $O(n^2)$。

因此，这个函数的时间复杂度是 $O(n^2)$。

17.8 最长单词

给定一组单词，编写一个程序，找出其中的最长单词，且该单词由这组单词中的其他单词组合而成。

示例：

输入：`cat, banana, dog, nana, walk, walker, dogwalker`

输出：`dogwalker`

题目解法

此题看似复杂，让我们先来简化一番。如果只是想知道由列表中的其他两个单词组成的最长单词，该怎么处理？

我们可以通过遍历整个列表，从最长单词到最短单词，将每个单词分割成所有可能的两半，

然后检查左右两半是否在列表中。

上述做法的伪码大致如下。

```
String getLongestWord(String[] list) {
  String[] array = list.SortByLength();
  /* 创建 map 以便查找 */
  HashMap<String, Boolean> map = new HashMap<String, Boolean>;

  for (String str : array) {
    map.put(str, true);
  }

  for (String s : array) {
    // 切分成所有可能的两半
    for (int i = 1; i < s.length(); i++) {
      String left = s.substring(0, i);
      String right = s.substring(i);
      // 检查左右两半是否在数组中
      if (map[left] == true && map[right] == true) {
        return s;
      }
    }
  }
  return str;
}
```

若知道最长单词由另外两个单词组合而成时，以上方法非常行之有效。但是，若单词可以由任意数量的其他单词组成，又该怎么办呢?

在这种情况下，我们可以采用非常相似的做法，只修改一处：之前会检查右半部分是否在数组中，现在改为递归检查右半部分可否由数组其他元素构建出来。

下面是该算法的实现代码。

```
String printLongestWord(String arr[]) {
  HashMap<String, Boolean> map = new HashMap<String, Boolean>();
  for (String str : arr) {
    map.put(str, true);
  }
  Arrays.sort(arr, new LengthComparator()); // 按长度排序
  for (String s : arr) {
    if (canBuildWord(s, true, map)) {
      System.out.println(s);
      return s;
    }
  }
  return "";
}

boolean canBuildWord(String str, boolean isOriginalWord,
                     HashMap<String, Boolean> map) {
  if (map.containsKey(str) && !isOriginalWord) {
    return map.get(str);
  }
  for (int i = 1; i < str.length(); i++) {
    String left = str.substring(0, i);
    String right = str.substring(i);
    if (map.containsKey(left) && map.get(left) == true &&
       canBuildWord(right, false, map)) {
      return true;
    }
```

```
28    }
29    map.put(str, false);
30    return false;
31  }
```

注意，可对该解法稍作优化。我们使用动态规划方法缓存了多次调用之间的结果。这样一来，如需反复检查有无办法构造 testingtester，只需计算一次即可。

布尔标志 isOriginalWord 用于完成上述优化。调用 canBuildWord 方法时，会传入原始单词和每个子串，对于该算法，第一步会先检查缓存里有无之前计算好的结果。但是，这里也有个问题：对于原始单词，map 会将这些单词初始化为 true，但我们又不想返回 true（因为单词不能只由它本身组成）。因此，对于原始单词，我们会利用 isOriginalWord 标志直接跳过这项检查。

17.9 按摩师

一个有名的按摩师会收到源源不断的预约请求，每个预约都可以选择接或不接。在每次预约服务之间要有 15 分钟的休息时间，因此她不能接受时间相邻的预约。给定一个预约请求序列（都是 15 分钟的倍数，没有重叠，也无法移动），替按摩师找到最优的预约集合（总预约时间最长），返回总的分钟数。

示例：

输入：{30, 15, 60, 75, 45, 15, 15, 45}
输出：180 minutes ({30, 60, 45, 45})

题目解法

让我们先看一个例子。通过直观地作图来更好地理解这个问题。图中每个数字表示预约的分钟数。

r_0 = 75	r_1 = 105	r_2 = 120	r_3 = 75	r_4 = 90	r_5 = 135

或者我们也可以将所有的值（包括休息时间）除以 15 分钟，这样就可以得到数组{5, 7, 8, 5, 6, 9}。两种表示方法相同，但现在休息时间是 1 分钟。

这个问题的最佳预约方法总计有 330 分钟，由{r_0 = 75, r_2 = 120, r_5 = 135}组成。注意，我们有意选择了一个例子，在这个例子中，最佳的预约顺序不是通过严格的交替序列形成的。

我们还应该认识到，首先选择最长预约（“贪婪”策略）未必是最佳选择。例如，像{45, 60, 45, 15}这样的序列在最优集合中不会包含 60。

解法 1：递归法

我们第一个想到是递归法。当列出预约清单时，实际上有多种选择。是否选择这个预约？如果选择预约 i，则必须跳过预约 $i + 1$，因为不能选择连续的预约。预约 $i + 2$ 是一种可能的但不一定是最好的选择。

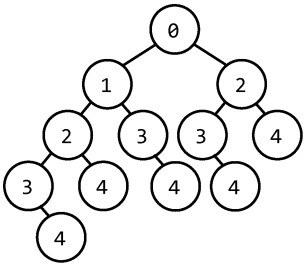
```
1  int maxMinutes(int[] massages) {
2    return maxMinutes(massages, 0);
3  }
4
5  int maxMinutes(int[] massages, int index) {
6    if (index >= massages.length) { // 超出边界
7      return 0;
8    }
```

```
9
10    /* 有此预约最佳情况 */
11    int bestWith = massages[index] + maxMinutes(massages, index + 2);
12
13    /* 无此预约最佳情况 */
14    int bestWithout = maxMinutes(massages, index + 1);
15
16    /* 从该 index 开始返回子数组的最佳值 */
17    return Math.max(bestWith, bestWithout);
18  }
```

因为在每个元素中做了两种选择，而我们重复了 n 次（其中 n 为按摩次数），所以这个解法的运行时间是 $O(2^n)$。

由于递归调用使用了栈，因此空间复杂度是 $O(n)$。

我们也可以通过一个长度为 5 的数组来描述递归调用树。每个节点中的数字表示调用 `maxMinutes` 时的索引值。例如，你会发现，`maxMinutes(massages, 0)`调用 `maxMinutes(massages, 1)` 和 `maxMinutes(massages, 2)`。



和许多递归问题一样，我们应该评估一下是否有可能记忆重复的子问题。事实上确实可以。

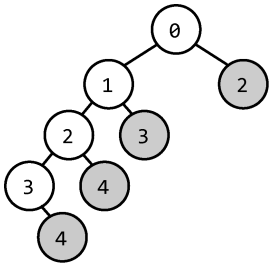
解法 2：递归法 + 记忆法

我们将在相同输入的情况下重复调用 `maxMinutes`。例如，当决定是否要选择预约 0 时，将以索引 2 为输入调用 `maxMinutes`。当决定是否要选择预约 1 时，也将以索引 2 为输入调用 `maxMinutes`。我们需要记忆该结果。

该 memo 表只是一个从 `index` 到最大分钟的映射。因此，一个简单的数组就足够了。

```
1   int maxMinutes(int[] massages) {
2     int[] memo = new int[massages.length];
3     return maxMinutes(massages, 0, memo);
4   }
5
6   int maxMinutes(int[] massages, int index, int[] memo) {
7     if (index >= massages.length) {
8       return 0;
9     }
10
11    if (memo[index] == 0) {
12      int bestWith = massages[index] + maxMinutes(massages, index + 2, memo);
13      int bestWithout = maxMinutes(massages, index + 1, memo);
14      memo[index] = Math.max(bestWith, bestWithout);
15    }
16
17    return memo[index];
18  }
```

为了确定运行时间，我们将像以前一样绘制相同的递归调用树，但是将会把立即返回的调用涂成灰色，并将不会发生的调用完全删除。



如果画一棵更大的树，会看到类似的图案。这棵树看起来基本是线性的，只有左边有一个分支。这表明该解法的时间复杂度为 $O(n)$，空间复杂度也为 $O(n)$。空间使用来自于递归调用栈以及 memo 表。

解法 3：迭代法

可以做得更好吗？我们当然无法在时间复杂度上有所提高，因为必须读取每一个预约信息。然而，我们也许能够提高空间复杂度。这意味着需要以非递归方式解决问题。

让我们再看一下第一个例子。

$r_0 = 30$	$r_1 = 15$	$r_2 = 60$	$r_3 = 75$	$r_4 = 45$	$r_5 = 15$	$r_6 = 15$	$r_7 = 45$

正如我们在问题描述中所指出的，不接受相邻的预约。

不过，还可以观察到另一件事：我们不应该**跳过**连续三次预约，也就是说，如果想选择 r_0 和 r_3，那么可以跳过 r_1 和 r_2。但是不会跳过 r_1、r_2 和 r_3，因为这并不是最佳方案。我们可以通过选择中间元素来改进预约的集合。

这意味着，如果取 r_0，那么肯定会跳过 r_1，选择 r_2 和 r_3 中的一个。这极大地限制了我们来计算可能的情况，为使用迭代法解题创造了可能。

回想一下之前是怎么用递归法加上记忆法解题的，尝试反推其中的逻辑方法，换句话说，试着用迭代法解答此题。

一种有效的方法是对数组从后向前进行计算。在每个元素处，计算子数组的解题方法。

- `best(7): {`r_7 `= 45}`的最佳方案是什么？如果选择 r_7，总计得到 45 分钟。所以 `best(7) = 45`。
- `best(6): {`r_6 `= 15, ...}`的最佳方案是什么？依然是 45 分钟。所以 `best(6) = 45`。
- `best(5): {`r_5 `= 15, ...}`的最佳方案是什么？有如下两种选择。
  - 选择 r_5 `= 15` 并将其与 `best(7) = 45` 合并。
  - 选择 `best(6) = 45`。

  前者总计 60 分钟，所以 `best(5) = 60`。
- `best(4): {`r_4 `= 45, ...}`的最佳方案是什么？有如下两种选择。
  - 选择 r_4 `= 45` 并将其与 `best(6) = 45` 合并。
  - 选择 `best(5) = 60`。

  前者总计 90 分钟，所以 `best(4) = 90`。
- `best(3): {`r_3 `= 75, ...}`的最佳方案是什么？有如下两种选择。
  - 选择 r_3 `= 75` 并将其与 `best(5) = 60` 合并。
  - 选择 `best(4) = 90`。

  前者总计 135 分钟，所以 `best(3) = 135`。
- `best(2): {`r_2 `= 60, ...}`的最佳方案是什么？有如下两种选择。

- 选择 r_2 = 60 并将其与 best(4) = 90 合并。
- 选择 best(3) = 135。

前者总计 150 分钟，所以 best(2) = 150。

- best(1): {r_1 = 15, ...}的最佳方案是什么？有如下两种选择。
  - 选择 r_1 = 15 并将其与 best(3) = 135 合并。
  - 选择 best(2) = 150。

  前后两者结果一样，所以 best(1) = 150。
- best(0): {r_0 = 30, ...}的最佳方案是什么？有如下两种选择。
  - 选择 r_0 = 30 并将其与 best(2) = 150 合并。
  - 选择 best(1) = 150。

  前者总计 180 分钟，所以 best(0) = 180。

因此，我们返回 180 分钟。

下面的代码实现了这个算法。

```
1  int maxMinutes(int[] massages) {
2    /* 分配额外的两个元素的空间，这样我们就无须在 7~8 行的代码中检查边界 */
3    int[] memo = new int[massages.length + 2];
4    memo[massages.length] = 0;
5    memo[massages.length + 1] = 0;
6    for (int i = massages.length - 1; i >= 0; i--) {
7      int bestWith = massages[i] + memo[i + 2];
8      int bestWithout = memo[i + 1];
9      memo[i] = Math.max(bestWith, bestWithout);
10   }
11   return memo[0];
12 }
```

这个解法的运行时间是 $O(n)$，空间复杂度也是 $O(n)$。

该解法在某些方面很好，因为采用了迭代法，但该解法实际上并没有“胜出”。递归解法和此解法具有相同的时间和空间复杂度。

解法 4：优化时间和空间的迭代

在回顾最后一个解法的时候，可知我们只短暂使用了 memo 表中的值。一旦超过某索引若干个元素之后，就不再使用该索引了。

事实上，对于任何给定的索引 i，我们只需要知道 i + 1 和 i + 2 的最佳值。因此，可以删除 memo 表，只使用两个整数。

```
1  int maxMinutes(int[] massages) {
2    int oneAway = 0;
3    int twoAway = 0;
4    for (int i = massages.length - 1; i >= 0; i--) {
5      int bestWith = massages[i] + twoAway;
6      int bestWithout = oneAway;
7      int current = Math.max(bestWith, bestWithout);
8      twoAway = oneAway;
9      oneAway = current;
10   }
11   return oneAway;
12 }
```

该解法给出了可能情况下最优的时间和空间复杂度，分别为 $O(n)$和 $O(1)$。

为什么要从后向前计算？在许多问题中，通过数组从后向前计算是一种常见技巧。

然而，如果我们愿意，也可以从前向后计算。对一些人来说，这样做思考起来更容易，对其他人来说则更困难一些。在从前向后的解法中，我们应该问自己“以 a[i]结尾的最佳集合是什么”，而不是问“从 a[i]开始的最佳集合是什么”。

17.10 最短超串

假设你有两个数组，一个长一个短，短的元素均不相同。找到长数组中包含短数组所有的元素的最短子数组，其出现顺序无关紧要。

示例：

输入：{1, 5, 9} | {7, 5, 9, 0, 2, 1, 3, 5, 7, 9, 1, 1, 5, 8, 8, 9, 7}
输出：[7, 10] (the underlined portion above)

题目解法

照例，可以先试试蛮力法。试着把它想象成你在对该题进行手动计算。你会怎么做？

让我们用问题中的例子来推导这个问题。将较小的数组称为 smallArray 并把较大的数组称为 bigArray。

1. 蛮力法

一种虽慢但简单的方法是对 bigArray 进行迭代，并对其进行小规模的重复遍历。

在 bigArray 的每个索引位置上，向前扫描，查找 smallArray 中每个元素的下一个出现位置。下一个出现位置的最大值将告诉我们从该索引开始的最短子数组［我们称之为“终结位置”（closure），也就是说，终结位置是从该索引开始的“终止”一个完整子数组的元素。例如，索引 3 的终结位置（在示例中值为 0）为索引 9］。

通过查找数组中每个索引的终结位置，我们可以找到最短的子数组。

```
1  Range shortestSupersequence(int[] bigArray, int[] smallArray) {
2    int bestStart = -1;
3    int bestEnd = -1;
4    for (int i = 0; i < bigArray.length; i++) {
5      int end = findClosure(bigArray, smallArray, i);
6      if (end == -1) break;
7      if (bestStart == -1 || end - i < bestEnd - bestStart) {
8        bestStart = i;
9        bestEnd = end;
10     }
11   }
12   return new Range(bestStart, bestEnd);
13 }
14
15 /* 给定一个索引位置，寻找终结位置（即若以该位置为子数组的终结位置，
16  * 该子数组将包含所有 smallArray 的元素）。这将成为 smallArray 每个
17  * 元素所对应的下一个位置中的最大值 */
18 int findClosure(int[] bigArray, int[] smallArray, int index) {
19   int max = -1;
20   for (int i = 0; i < smallArray.length; i++) {
21     int next = findNextInstance(bigArray, smallArray[i], index);
22     if (next == -1) {
23       return -1;
24     }
25     max = Math.max(next, max);
26   }
```

```
  return max;
}

/* 获取从 index 位置开始的下一个出现位置 */
int findNextInstance(int[] array, int element, int index) {
  for (int i = index; i < array.length; i++) {
    if (array[i] == element) {
      return i;
    }
  }
  return -1;
}

public class Range {
  private int start;
  private int end;
  public Range(int s, int e) {
    start = s;
    end = e;
  }

  public int length() { return end - start + 1; }
  public int getStart() { return start; }
  public int getEnd() { return end; }

  public boolean shorterThan(Range other) {
    return length() < other.length();
  }
}
```

这个算法可能会花费 $O(SB^2)$的时间，其中 B 是 `bigString` 的长度，S 是 `smallString` 的长度。这是因为对于 B 个字符中的每一个字符，我们都有可能完成 $O(SB)$的计算：对字符串的其余部分进行 S 次扫描，其中有可能存在 B 个字符。

2. 优化解法

考虑一下如何优化上述算法。该算法运行缓慢的核心原因在于重复性的搜索。能否找到一种更快的方法，使得在给定一个索引的情况下，可以找出下一个特定字符出现的位置？

先看一个例子。设想有下面的数组，是否有一种方法可以快速地从每个索引位置找到下一个 5？

```
7, 5, 9, 0, 2, 1, 3, 5, 7, 9, 1, 1, 5, 8, 8, 9, 7
```

是的。因为要重复地完成该计算，所以可以在一次（由后向前的）扫描中预先计算出这些信息。由前向后遍历数组，跟踪上一次（最近）出现 5 的位置。

value	7	5	9	0	2	1	3	5	7	9	1	1	5	8	8	9	7
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
next 5	1	1	7	7	7	7	7	7	12	12	12	12	12	x	x	x	x

对`{1, 5, 9}`中的每一个元素执行此操作，只需由后向前扫描 3 次。

有些人想把以上方法合并成 1 次由后向前的扫描来处理所有 3 个值。这种方法似乎更快，但其实并非如此。在 1 次由后向前的扫描中，每次迭代要进行 3 次比较。N 次对列表进行扫描，每次扫描进行 3 次比较，该方法最终花费的时间不会优于 $3N$ 次对列表进行扫描而每次扫描进行 1 次比较这一方法。同时，只进行 1 次比较的扫描方法可以保持代码的整洁性。

value	7	5	9	0	2	1	3	5	7	9	1	1	5	8	8	9	7
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
next 1	5	5	5	5	5	5	10	10	10	10	10	11	x	x	x	x	x
next 5	1	1	7	7	7	7	7	7	12	12	12	12	12	x	x	x	x
next 9	2	2	2	9	9	9	9	9	9	9	15	15	15	15	15	15	x

`findNextInstance` 函数现在可以使用此表来查找下一个出现位置，而无须进行搜索计算。

但是，实际上可以让该方法更简单一些。使用上面的表，我们可以快速计算每个索引的终结位置。终结位置只是一列中的最大值。如果一列中存在一个值 x，同时不存在终结位置，则说明后续字符中再无该字符出现。

索引和终结位置之间的差即为从该索引开始的最小子数组。

value	7	5	9	0	2	1	3	5	7	9	1	1	5	8	8	9	7
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
next 1	5	5	5	5	5	5	10	10	10	10	10	11	x	x	x	x	x
next 5	1	1	7	7	7	7	7	7	12	12	12	12	12	x	x	x	x
next 9	2	2	2	9	9	9	9	9	9	9	15	15	15	15	15	15	x
closure	5	5	7	9	9	9	10	10	12	12	15	15	x	x	x	x	x
diff.	5	4	5	6	5	4	4	3	4	3	5	4	x	x	x	x	x

现在，我们要做的就是求出这张表中的最小距离。

```
Range shortestSupersequence(int[] big, int[] small) {
  int[][] nextElements = getNextElementsMulti(big, small);
  int[] closures = getClosures(nextElements);
  return getShortestClosure(closures);
}

/* 构造下一次出现位置的表格 */
int[][] getNextElementsMulti(int[] big, int[] small) {
  int[][] nextElements = new int[small.length][big.length];
  for (int i = 0; i < small.length; i++) {
    nextElements[i] = getNextElement(big, small[i]);
  }
  return nextElements;
}

/* 向反方向遍历，获取一个包含从每个索引位置开始的“下一个出现位置”构成的链表 */
int[] getNextElement(int[] bigArray, int value) {
  int next = -1;
  int[] nexts = new int[bigArray.length];
  for (int i = bigArray.length - 1; i >= 0; i--) {
    if (bigArray[i] == value) {
      next = i;
    }
    nexts[i] = next;
  }
  return nexts;
}

/* 获取每个索引位置的终结位置 */
int[] getClosures(int[][] nextElements) {
  int[] maxNextElement = new int[nextElements[0].length];
  for (int i = 0; i < nextElements[0].length; i++) {
    maxNextElement[i] = getClosureForIndex(nextElements, i);
  }
  return maxNextElement;
```

```
}

/* 给定索引和表格，获取该表格的终结位置（该列的最小值） */
int getClosureForIndex(int[][] nextElements, int index) {
  int max = -1;
  for (int i = 0; i < nextElements.length; i++) {
    if (nextElements[i][index] == -1) {
      return -1;
    }
    max = Math.max(max, nextElements[i][index]);
  }
  return max;
}

/* 获取最短终结位置 */
Range getShortestClosure(int[] closures) {
  int bestStart = -1;
  int bestEnd = -1;
  for (int i = 0; i < closures.length; i++) {
    if (closures[i] == -1) {
      break;
    }
    int current = closures[i] - i;
    if (bestStart == -1 || current < bestEnd - bestStart) {
      bestStart = i;
      bestEnd = closures[i];
    }
  }
  return new Range(bestStart, bestEnd);
}
```

这个算法可能会花费 $O(SB)$的时间，其中 B 是 `bigString` 的长度，S 是 `smallString` 的长度。原因在于，我们通过对数组进行 S 次扫描来构建下一字符出现位置的表格，而每次扫描都需要 $O(B)$的时间。

该算法占用 $O(SB)$的空间。

3. 更优解法

尽管以上解法算得上上乘之选，但仍然可以通过减少空间使用量进行优化。记住在上述算法中创建的表格。

value	7	5	9	0	2	1	3	5	7	9	1	1	5	8	8	9	7
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
next 1	5	5	5	5	5	5	10	10	10	10	10	11	x	x	x	x	x
next 5	1	1	7	7	7	7	7	7	12	12	12	12	12	x	x	x	x
next 9	2	2	2	9	9	9	9	9	9	9	15	15	15	15	15	15	x
closure	5	5	7	9	9	9	10	10	12	12	15	15	x	x	x	x	x

实际上，我们需要的是终结位置一行，它是所有其他行的最小值。不需要在运行整个算法时始终都保存其他所有信息。

一种取而代之的方法是，当我们进行每一次扫描时，只需更新终结位置一行的最小值。剩下的算法基本上和前述算法大同小异。

```
Range shortestSupersequence(int[] big, int[] small) {
  int[] closures = getClosures(big, small);
  return getShortestClosure(closures);
}

```

```
/* 获取每个索引位置的终结位置 */
int[] getClosures(int[] big, int[] small) {
  int[] closure = new int[big.length];
  for (int i = 0; i < small.length; i++) {
    sweepForClosure(big, closure, small[i]);
  }
  return closure;
}

/* 向反方向遍历，如果下一个终结位置大于当前终结位置则更新终结位置链表 */
void sweepForClosure(int[] big, int[] closures, int value) {
  int next = -1;
  for (int i = big.length - 1; i >= 0; i--) {
    if (big[i] == value) {
      next = i;
    }
    if ((next == -1 || closures[i] < next) &&
        (closures[i] != -1)) {
      closures[i] = next;
    }
  }
}

/* 获取最短终结位置 */
Range getShortestClosure(int[] closures) {
  Range shortest = new Range(0, closures[0]);
  for (int i = 1; i < closures.length; i++) {
    if (closures[i] == -1) {
      break;
    }
    Range range = new Range(i, closures[i]);
    if (!shortest.shorterThan(range)) {
      shortest = range;
    }
  }
  return shortest;
}
```

该算法仍然以 $O(SB)$的时间运行，但是现在只需要 $O(B)$的额外内存。

4. 另一种更优解法

还有另一种截然不同的解法。假设我们有一个列表，包含每个元素在 `smallArray` 中出现的位置。

value	7	5	9	9	2	1	3	5	7	9	1	1	5	8	8	9	7
index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

```
1 -> {5, 10, 11}
5 -> {1, 7, 12}
9 -> {2, 3, 9, 15}
```

第一个有效子序列（包含 1、5 和 9）是什么？可以查看每个列表的头部来获知此信息。头部的最小值是子序列范围的开始，头部的最大值是子序列范围的结束。在这种情况下，第一个子序列的范围是`[1, 5]`。这是目前我们得到的“最佳”子序列。

怎样才能找到下一个子序列呢？下一个子序列不包括索引 1，所以将其从列表中移除。

```
1 -> {5, 10, 11}
5 -> {7, 12}
9 -> {2, 3, 9, 15}
```

下一个子序列是[2, 7]。这比之前的“最佳”子序列更差，所以可以将其忽略。

那么，下一个子序列是什么？我们可以从前面的（2）中取出最小值，然后找出答案。

```
1 -> {5, 10, 11}
5 -> {7, 12}
9 -> {3, 9, 15}
```

下一个子序列是[3, 7]，与当前的最佳子序列不相上下。

我们可以沿着这条路径重复这个过程，继续计算下去。最后将遍历从给定点开始的所有“最小”子序列。

(1) 当前子序列范围是[头部最小值, 头部最大值]。与最佳子序列进行比较，并在必要时进行更新。

(2) 删除头部最小值。

(3) 重复此过程。

该算法的时间复杂度是 $O(SB)$。这是因为对于 B 个元素中的每一个元素，我们都将其与 S 个其他列表的头部进行比较，以找出最小值。

该算法还算不错，但是让我们看看能否更快地计算出最小值。

在这些最小值计算中重复进行的操作是：提取一些元素，找到并移除最小值，加入一个元素，然后再找到最小值。

可以通过使用小顶堆使该过程运行更快。首先，把每个列表头部放在一个小堆里，删除最小值，查找包含该最小值的列表并添加新的头部。重复此过程。

要获取最小元素的来源列表，我们需要使用一个 HeapNode 类来存储 locationWithinList（索引）和 listId。这样，当移除最小值时，可以跳转回正确的列表并将其新的头部添加到堆中。

```
1  Range shortestSupersequence(int[] array, int[] elements) {
2    ArrayList<Queue<Integer>> locations = getLocationsForElements(array, elements);
3    if (locations == null) return null;
4    return getShortestClosure(locations);
5  }
6
7  /* 获取一组队列（链表），用于对每个 smallArray 中元素出现在 bigArray 索引位置进行排序 */
8  ArrayList<Queue<Integer>> getLocationsForElements(int[] big, int[] small) {
9    /* 以值和索引位置初始化散列表 */
10   HashMap<Integer, Queue<Integer>> itemLocations =
11     new HashMap<Integer, Queue<Integer>>();
12   for (int s : small) {
13     Queue<Integer> queue = new LinkedList<Integer>();
14     itemLocations.put(s, queue);
15   }
16
17   /* 遍历 bigArray，将该项的位置加入到散列表中 */
18   for (int i = 0; i < big.length; i++) {
19     Queue<Integer> queue = itemLocations.get(big[i]);
20     if (queue != null) {
21       queue.add(i);
22     }
23   }
24
25   ArrayList<Queue<Integer>> allLocations = new ArrayList<Queue<Integer>>();
26   allLocations.addAll(itemLocations.values());
27   return allLocations;
28 }
29
```

```
Range getShortestClosure(ArrayList<Queue<Integer>> lists) {
  PriorityQueue<HeapNode> minHeap = new PriorityQueue<HeapNode>();
  int max = Integer.MIN_VALUE;

  /* 插入每个链表中的最小值 */
  for (int i = 0; i < lists.size(); i++) {
    int head = lists.get(i).remove();
    minHeap.add(new HeapNode(head, i));
    max = Math.max(max, head);
  }

  int min = minHeap.peek().locationWithinList;
  int bestRangeMin = min;
  int bestRangeMax = max;

  while (true) {
    /* 删除最小节点 */
    HeapNode n = minHeap.poll();
    Queue<Integer> list = lists.get(n.listId);

    /* 比较当前范围与最佳范围 */
    min = n.locationWithinList;
    if (max - min < bestRangeMax - bestRangeMin) {
      bestRangeMax = max;
      bestRangeMin = min;
    }

    /* 如果没有更多元素，则没有更多的子序列。我们可以就此跳出 */
    if (list.size() == 0) {
      break;
    }

    /* 将链表的新头节点加入到堆中 */
    n.locationWithinList = list.remove();
    minHeap.add(n);
    max = Math.max(max, n.locationWithinList);
  }

  return new Range(bestRangeMin, bestRangeMax);
}
```

我们在 `getShortestClosure` 中遍历了 B 个元素，每次传入 `for` 循环时都将花费 $O(\log S)$ 的时间（从堆中插入/删除的时间）。因此，在最坏情况下，该算法将花费 $O(B \log S)$的时间。

17.11 连续中值

随机产生数字并传递给一个方法。你能否完成这个方法，在每次产生新值时，寻找当前所有值的中间值并保存。

题目解法

一种解法是使用两个优先级堆（priority heap），即一个大顶堆，存放小于中位数的值，以及一个小顶堆，存放大于中位数的值。这会将所有元素大致分为两半，中间的两个元素位于两个堆的堆顶。这样一来，要找出中间值就是小事一桩。

不过，“大致分为两半”又是什么意思呢？“大致”的意思是，如果有奇数个值，其中一个堆就会多一个值。经观察可知，以下两点为真。

- 如果 maxHeap.size() > minHeap.size()，则 maxHeap.top()为中间值。
- 如果 maxHeap.size() == minHeap.size()，则 maxHeap.top()和 minHeap.top()的平均值为中间值。

当要重新平衡这两个堆时，我们会确保 maxHeap 一定会多一个元素。

这个算法的解法如下所示。有新的值生成时，如果这个值小于等于中间值，就放入 maxHeap 中，否则放入 minHeap 中。两个堆的元素个数相等或者 maxHeap 可能多一个元素。这个限制条件很容易得到保证，不满足的话，只要从一个堆搬移一个元素到另一个堆即可。通过查看 maxHeap 或两个堆的堆顶元素，就能以常数时间获取中间值，而更新操作的用时为 $O(\log(n))$。

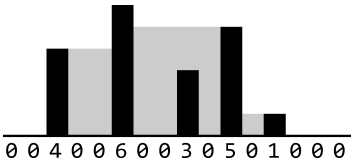
```
Comparator<Integer> maxHeapComparator, minHeapComparator;
PriorityQueue<Integer> maxHeap, minHeap;

void addNewNumber(int randomNumber) {
  /* addNewNumber 满足 maxHeap.size() >= minHeap.size() */
  if (maxHeap.size() == minHeap.size()) {
    if ((minHeap.peek() != null) &&
      randomNumber > minHeap.peek()) {
      maxHeap.offer(minHeap.poll());
      minHeap.offer(randomNumber);
    } else {
      maxHeap.offer(randomNumber);
    }
  } else {
    if(randomNumber < maxHeap.peek()) {
      minHeap.offer(maxHeap.poll());
      maxHeap.offer(randomNumber);
    }
    else {
      minHeap.offer(randomNumber);
    }
  }
}

double getMedian() {
  /* maxHeap 至少和 minHeap 一样大。如果 maxHeap 是空的，则 minHeap 也为空 */
  if (maxHeap.isEmpty()) {
    return 0;
  }
  if (maxHeap.size() == minHeap.size()) {
    return ((double)minHeap.peek()+(double)maxHeap.peek()) / 2;
  } else {
    /* 如果 maxHeap 和 minHeap 大小不一样，则 maxHeap 必定多一个元素。
     * 返回 maxHeap 的顶部元素 */
    return maxHeap.peek();
  }
}
```

17.12　直方图的水量

给定一个直方图（也称柱状图），假设有人从上面源源不断地倒水，最后直方图能存多少水量？直方图的宽度为 1。

示例（黑色部分是直方图，灰色部分是水）：

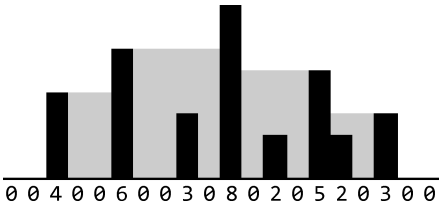
输入：{0, 0, 4, 0, 0, 6, 0, 0, 3, 0, 5, 0, 1, 0, 0, 0}



输出：26

题目解法

这道题目较难，因此，让我们使用一个简单的例子来更好地解题。



我们要仔细研究这个例子以便从中获益。灰色区域的面积究竟是由什么决定的？

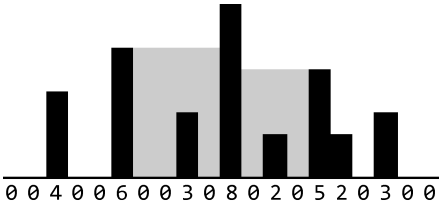
解法 1

让我们观察最高的长方形。它的高度为 8。该长方形有什么作用？它的确是最高的，但实际上并不重要，如果是个酒吧，即使高度是 100 其实也无关紧要。这并不会影响柱状图的容积。

最高的长方形在其左右两侧形成了一道屏障。但是积水量实际上由左右两侧的另一侧最高的长方形控制。

- **最高长方形直接相邻的左侧积水。**左侧下一个最高的长方形高度为 6。我们可以将其全部区域注水，但是计算面积时需要减去最高的长方形与第二高的长方形之间的所有长方形占用的面积。因此可以得到直接相邻的左侧积水面积为$(6-0)+(6-0)+(6-3)+(6-0)=21$。
- **最高长方形直接相邻的右侧积水。**右侧下一个最高的长方形高度为 5。因此可以计算出积水面积为$(5-0)+(5-2)+(5-0)=13$。

至此，我们只得到了部分积水的面积。



其余的该如何计算呢？

事实上现在有两个子图，分别位于左右两侧，只需重复以上操作即可计算出积水面积。解题过程与上面大同小异，如下所示。

(1) 找到最高的长方形。事实上，至此已得知最高的长方形。左侧子图中最高的长方形是其右侧边界（6），右侧子图中最高的长方形是其左侧边界（5）。

(2) 在每个子图中找到第二高的长方形。

(3) 计算最高的长方形和第二高的长方形中的积水面积。

(4) 以该图的边界进行递归。

下面的代码实现了该算法。

```
int computeHistogramVolume(int[] histogram) {
  int start = 0;
  int end = histogram.length - 1;

  int max = findIndexOfMax(histogram, start, end);
  int leftVolume = subgraphVolume(histogram, start, max, true);
  int rightVolume = subgraphVolume(histogram, max, end, false);

  return leftVolume + rightVolume;
}

/* 计算直方图的子图面积。max 应为 start 或 end。找到第二高的位置，
 * 计算最高与第二高的长方形之间的面积。之后计算子图的面积 */
int subgraphVolume(int[] histogram, int start, int end, boolean isLeft) {
  if (start >= end) return 0;
  int sum = 0;
  if (isLeft) {
    int max = findIndexOfMax(histogram, start, end - 1);
    sum += borderedVolume(histogram, max, end);
    sum += subgraphVolume(histogram, start, max, isLeft);
  } else {
    int max = findIndexOfMax(histogram, start + 1, end);
    sum += borderedVolume(histogram, start, max);
    sum += subgraphVolume(histogram, max, end, isLeft);
  }

  return sum;
}

/* 计算 start 和 end 之间最高的长方形 */
int findIndexOfMax(int[] histogram, int start, int end) {
  int indexOfMax = start;
  for (int i = start + 1; i <= end; i++) {
    if (histogram[i] > histogram[indexOfMax]) {
      indexOfMax = i;
    }
  }
  return indexOfMax;
}

/* 计算 start 和 end 之间的面积。假设最高的长方形位于 start 处，第二高的长方形位于 end 处 */
int borderedVolume(int[] histogram, int start, int end) {
  if (start >= end) return 0;

  int min = Math.min(histogram[start], histogram[end]);
  int sum = 0;
  for (int i = start + 1; i < end; i++) {
    sum += min - histogram[i];
  }
  return sum;
}
```

因为需要反复扫描直方图以寻找最高的长方形，该算法在最坏情况下花费 $O(N^2)$的时间，其中 N 是直方图中长方形的数目。

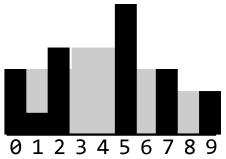
解法 2（优化解法）

为了优化先前的算法，我们来思考一下上述算法效率低下的确切原因：对 `findIndexOfMax` 的不断调用。这表明应该重点对其进行优化。

应该注意到的一点是，我们不会将任意范围传递给 `findIndexOfMax` 函数。该函数实际上总是寻找从一个点到一个边界（右边界或左边界）的最大值。可以更快地确定从给定点到每个边界的最大高度是多少吗？

答案是可以。我们可以在 $O(N)$的时间内预先计算这些信息。

通过对直方图的两次扫描（一次从右向左移动，另一次从左向右移动），可以构造一个表格，该表格可以反映从任意索引 i 开始，其右侧最高长方形的索引位置和左侧最高长方形的索引位置。



索引: 0 1 2 3 4 5 6 7 8 9
高度: 3 1 4 0 0 6 0 3 0 2
左侧最高长方形的索引位置: 0 0 2 2 2 5 5 5 5 5
右侧最高长方形的索引位置: 5 5 5 5 5 5 7 7 9 9

算法的其余部分与上述算法本质上相同。

我们选择使用`HistogramData`对象来存储该额外信息，但是同样也可以使用一个二维数组。

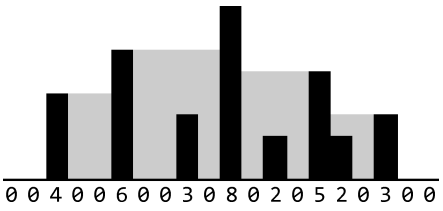
```
1  int computeHistogramVolume(int[] histogram) {
2    int start = 0;
3    int end = histogram.length - 1;
4
5    HistogramData[] data = createHistogramData(histogram);
6
7    int max = data[0].getRightMaxIndex(); // 获取总体最大值
8    int leftVolume = subgraphVolume(data, start, max, true);
9    int rightVolume = subgraphVolume(data, max, end, false);
10
11   return leftVolume + rightVolume;
12 }
13
14 HistogramData[] createHistogramData(int[] histo) {
15   HistogramData[] histogram = new HistogramData[histo.length];
16   for (int i = 0; i < histo.length; i++) {
17     histogram[i] = new HistogramData(histo[i]);
18   }
19
20   /* 设置左侧 max 的值 */
21   int maxIndex = 0;
22   for (int i = 0; i < histo.length; i++) {
23     if (histo[maxIndex] < histo[i]) {
24       maxIndex = i;
25     }
26     histogram[i].setLeftMaxIndex(maxIndex);
27   }
28
29   /* 设置右侧 max 的值 */
30   maxIndex = histogram.length - 1;
31   for (int i = histogram.length - 1; i >= 0; i--) {
32     if (histo[maxIndex] < histo[i]) {
33       maxIndex = i;
34     }
35     histogram[i].setRightMaxIndex(maxIndex);
36   }
37
```

```
  return histogram;
}

/* 计算直方图的子图面积。max 应为 start 或 end。找到第二高的位置，
 * 计算最高与第二高的长方形之间的面积。之后计算子图的面积 */
int subgraphVolume(HistogramData[] histogram, int start, int end,
                   boolean isLeft) {
  if (start >= end) return 0;
  int sum = 0;
  if (isLeft) {
    int max = histogram[end - 1].getLeftMaxIndex();
    sum += borderedVolume(histogram, max, end);
    sum += subgraphVolume(histogram, start, max, isLeft);
  } else {
    int max = histogram[start + 1].getRightMaxIndex();
    sum += borderedVolume(histogram, start, max);
    sum += subgraphVolume(histogram, max, end, isLeft);
  }

  return sum;
}

/* 计算 start 和 end 之间的面积。假设最高的长方形位于 start 处，第二高的长方形位于 end 处 */
int borderedVolume(HistogramData[] data, int start, int end) {
  if (start >= end) return 0;

  int min = Math.min(data[start].getHeight(), data[end].getHeight());
  int sum = 0;
  for (int i = start + 1; i < end; i++) {
    sum += min - data[i].getHeight();
  }
  return sum;
}

public class HistogramData {
  private int height;
  private int leftMaxIndex = -1;
  private int rightMaxIndex = -1;

  public HistogramData(int v) { height = v; }
  public int getHeight() { return height; }
  public int getLeftMaxIndex() { return leftMaxIndex; }
  public void setLeftMaxIndex(int idx) { leftMaxIndex = idx; };
  public int getRightMaxIndex() { return rightMaxIndex; }
  public void setRightMaxIndex(int idx) { rightMaxIndex = idx; };
}
```

该算法花费 $O(N)$的时间。需要查看所有长方形，因此无法找到更优化的算法。

解法 3（优化且简化的解法）

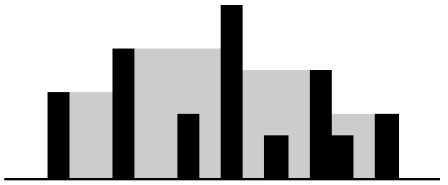
虽然无法使解法在大 O 表示下运行更快，但是可以大大简化该算法。根据刚刚了解的潜在算法，再来看一个例子。



正如我们所看到的，积水量取决于最高的长方形至左侧和右侧特定区域的面积（具体来说，左边两个最高的长方形中较矮的一个，以及右边最高的长方形）。例如，积水位于高度为 6 的长方形和高度为 8 的长方形之间的区域，积水高度为 6。高度为 6 的长方形是第二高的，因此决定了积水的高度。

积水的总体积是每个长方形上方水的体积。我们能否有效地计算出每个长方形上方有多少水？

可以的。在解法 2 中，我们能够预先计算出每个索引左侧和右侧最高长方形的高度。该两项数值中的最小值将表示长方形的“水位”。水位和该长方形高度的差值则是水的体积。



高度: 0 0 4 0 0 6 0 0 3 0 8 0 2 0 5 2 0 3 0 0
左侧最大高度: 0 0 4 4 4 6 6 6 6 6 8 8 8 8 8 8 8 8 8 8
右侧最大高度: 8 8 8 8 8 8 8 8 8 8 5 5 5 5 3 3 3 0 0
较小值: 0 0 4 4 4 6 6 6 6 6 5 5 5 5 3 3 3 0 0
差值: 0 0 0 4 4 0 6 6 3 6 0 5 3 5 0 1 3 0 0 0

至此，该算法通过以下简单的几步即可完成。

(1) 从左向右扫描，跟踪已知的最大高度并设定左侧最大高度（LEFT MAX）的值。

(2) 从右向左扫描，跟踪已知的最大高度并设定右侧最大高度（RIGHT MAX）的值。

(3) 扫描直方图，计算每个索引位置左侧最大高度和右侧最大高度中的较小值（MIN）。

(4) 扫描直方图，计算长方形和上述步骤中最小值的差值。对差值求和。

在实际的实现过程中，我们不需要保存太多的数据。步骤(2)、步骤(3)和步骤(4)可以合并为同一次扫描。首先，在一次扫描中计算左侧最大高度。然后反向扫描，随着扫描跟踪右侧最大高度。在每个元素处，计算左右最大值的较小值，然后计算“较小值”和索引位置长方形高度之间的差值。将该差值计入总和中。

```
1  /* 遍历所有长方形计算其上部面积，其中水的面积 = 高度- min（左侧最高长方形，右侧最高长方形）
2   * [如果此值为正数]。第一次遍历时计算左侧最高长方形，第二次遍历时计算右侧最高长方形、
3   * 长方形的最小值和差值 */
4  int computeHistogramVolume(int[] histo) {
5    /* 计算左侧最大长方形 */
6    int[] leftMaxes = new int[histo.length];
7    int leftMax = histo[0];
8    for (int i = 0; i < histo.length; i++) {
9      leftMax = Math.max(leftMax, histo[i]);
10     leftMaxes[i] = leftMax;
11   }
12
13   int sum = 0;
14
15   /* 计算右侧最大长方形 */
16   int rightMax = histo[histo.length - 1];
17   for (int i = histo.length - 1; i >= 0; i--) {
18     rightMax = Math.max(rightMax, histo[i]);
19     int secondTallest = Math.min(rightMax, leftMaxes[i]);
20
21     /* 如果左侧或者右侧有更高的长方形，则有积水。计算面积并计入总和中 */
22       if (secondTallest > histo[i]) {
```

```
        sum += secondTallest - histo[i];
      }
    }

    return sum;
  }
```

是的，这真的就是全部代码。它仍然花费 $O(N)$的时间，但是读、写该段代码都要简单得多。

17.13　最大黑方阵

给定一个方阵，其中每个单元（像素）非黑即白。设计一个算法，找出 4 条边皆为黑色像素的最大子方阵。

题目解法

和许多问题一样，此题也有难易两种解法，下面将逐一讲解。

1.“简单”解法：$O(N^4)$

我们知道最大子方阵的长度可能为 N，而且 $N \times N$ 的方阵只有一个，很容易就能检查这个方阵，符合要求则返回。

如果找不到 $N \times N$ 的方阵，可以尝试第二大的子方阵：$(N-1) \times (N-1)$。我们会迭代所有该尺寸的方阵，一旦找到符合要求的子方阵，立即返回。如果还未找到，则继续尝试 $N-2$、$N-3$，等等。由于我们是从大到小逐级搜索方阵，因此第一个找到的必定是最大的方阵。

实现代码具体如下。

```
Subsquare findSquare(int[][] matrix) {
  for (int i = matrix.length; i >= 1; i--) {
    Subsquare square = findSquareWithSize(matrix, i);
    if (square != null) return square;
  }
  return null;
}

Subsquare findSquareWithSize(int[][] matrix, int squareSize) {
  /* 在长度为 N 的边中，有 (N - sz + 1) 个长度为 sz 的方阵 */
  int count = matrix.length - squareSize + 1;

  /* 对所有边长为 squareSize 的方阵进行迭代 */
  for (int row = 0; row < count; row++) {
    for (int col = 0; col < count; col++) {
      if (isSquare(matrix, row, col, squareSize)) {
        return new Subsquare(row, col, squareSize);
      }
    }
  }
  return null;
}

boolean isSquare(int[][] matrix, int row, int col, int size) {
  // 检查上下边界
  for (int j = 0; j < size; j++){
    if (matrix[row][col+j] == 1) {
      return false;
    }
    if (matrix[row+size-1][col+j] == 1){
      return false;
```

```
    }
  }

  // 检查左右边界
  for (int i = 1; i < size - 1; i++){
    if (matrix[row+i][col] == 1){
      return false;
    }
    if (matrix[row+i][col+size-1] == 1) {
      return false;
    }
  }
  return true;
}
```

2. 预处理解法：$O(N^3)$

上面的“简单”解法之所以执行速度慢，很大一部分原因在于，每次检查一个可能符合要求的方阵，都要执行 $O(N)$的工作。通过预先做些处理，就可以把 isSquare 的时间复杂度降为 $O(1)$，而整个算法的时间复杂度降至 $O(N^3)$。

仔细分析 isSquare 的具体用处，就会发现它只需知道特定单元下方及右边的 squareSize 项是否为零。我们可以预先以直接、迭代的方式算好这些数据。

我们从右到左、自下而上迭代访问每个单元，并执行如下计算。

```
if A[r][c] is white, zeros right and zeros below are 0
else A[r][c].zerosRight = A[r][c + 1].zerosRight + 1
     A[r][c].zerosBelow = A[r + 1][c].zerosBelow + 1
```

下面这个例子给出了一个矩阵的相关值。

(0s right, 0s below)

0,0	1,3	0,0
2,2	1,2	0,0
2,1	1,1	0,0

原始矩阵

W	B	W
B	B	W
B	B	W

现在，使用 isSquare 方法不必再迭代 $O(N)$个元素，只需检查角落的 zerosRight 和 zerosBelow 即可。

下面是该算法的实现代码。注意，除了 findSquare 调用了 processSquare 以及之后操作了新的数据类型之外，findSquare 和 findSquareWithSize 基本相同。

```
public class SquareCell {
  public int zerosRight = 0;
  public int zerosBelow = 0;
  /* 声明、getter和setter */
}

Subsquare findSquare(int[][] matrix) {
  SquareCell[][] processed = processSquare(matrix);
  for (int i = matrix.length; i >= 1; i--) {
    Subsquare square = findSquareWithSize(processed, i);
    if (square != null) return square;
  }
  return null;
}
```

```
15
16 Subsquare findSquareWithSize(SquareCell[][] processed, int size) {
17   /* 与第一个算法相同 */
18 }
19
20 boolean isSquare(SquareCell[][] matrix, int row, int col, int sz) {
21   SquareCell topLeft = matrix[row][col];
22   SquareCell topRight = matrix[row][col + sz - 1];
23   SquareCell bottomLeft = matrix[row + sz - 1][col];
24
25   /* 分别检查上、下、左、右边 */
26   if (topLeft.zerosRight < sz || topLeft.zerosBelow < sz ||
27       topRight.zerosBelow < sz || bottomLeft.zerosRight < sz) {
28     return false;
29   }
30   return true;
31 }
32
33 SquareCell[][] processSquare(int[][] matrix) {
34   SquareCell[][] processed =
35     new SquareCell[matrix.length][matrix.length];
36
37   for (int r = matrix.length - 1; r >= 0; r--) {
38     for (int c = matrix.length - 1; c >= 0; c--) {
39       int rightZeros = 0;
40       int belowZeros = 0;
41       // 只有是黑色单元格时才需要处理
42       if (matrix[r][c] == 0) {
43         rightZeros++;
44         belowZeros++;
45         // 下一列在同一行上
46         if (c + 1 < matrix.length) {
47           SquareCell previous = processed[r][c + 1];
48           rightZeros += previous.zerosRight;
49         }
50         if (r + 1 < matrix.length) {
51           SquareCell previous = processed[r + 1][c];
52           belowZeros += previous.zerosBelow;
53         }
54       }
55       processed[r][c] = new SquareCell(rightZeros, belowZeros);
56     }
57   }
58   return processed;
59 }
```

17.14　单词矩阵

给定一份几百万个单词的清单，设计一个算法，创建由字母组成的最大矩形，其中每一行组成一个单词（自左向右），每一列也组成一个单词（自上而下）。不要求这些单词在清单里连续出现，但要求所有行等长，所有列等高。

题目解法

很多与字典有关的问题，通过预先做些处理就可以解出来。对于此题，哪一部分可以做预处理呢？

如果要创建一个单词矩形，就必须满足以下要求：每一行等长，每一列等高。因此，我们

可以将这个字典的单词按长短进行分组，姑且把这个分组叫作 D，其中 D[i]包含长度为 i 的单词串。

接下来，观察要找的最大矩形。可能形成的绝对最大的矩形有多大呢？它会是 length(largest word)^2。

```
int maxRectangle = longestWord * longestWord;
for z = maxRectangle to 1 {
  for each pair of numbers (i, j) where i*j = z {
    /* 试着用单词构建矩形，成功则返回 */
  }
}
```

从最大可能的矩形迭代至最小的矩形，可以保证第一个找到的符合要求的矩形就是题目要求的最大矩形。

现在，轮到困难的部分：makeRectangle(int l, int h)。这个方法试图构建长 l 高 h 的单词矩形。

一种做法是迭代所有长 h 的有序单词集合，然后检查每一列字母是否形成有效单词。这么做也行得通，但是效率相当低下。

假设我们正试着构造 6 × 5 的矩形，前几行单词如下所示。

```
there
queen
pizza
.....
```

至此可知，第一列开头几个字母为 tqp。我们知道或者说应该知道，字典里没有以 tqp 开头的单词。既然明摆着最终创建不出有效的矩形，为何还要自寻烦恼，继续构造下去呢？

这就引出一个更优的解法。我们可以构建一棵单词查找树（trie），从而轻易查出某个子串是否为字典里单词的前缀。再一行一行自上而下构造矩形时，检查每一列字母是否均为有效前缀。如果不是，则立即失败并中止，不再继续构造这个矩形。

下面是该算法的实现代码，长且复杂，我们会逐步解说。

一开始会做些预处理，将单词按长度分组。我们会创建一个单词查找树（每一个 trie 包含某长度的单词）数组，但直到真正需要时，才会构建单词查找树。

```
WordGroup[] groupList = WordGroup.createWordGroups(list);
int maxWordLength = groupList.length;
Trie trieList[] = new Trie[maxWordLength];
```

maxRectangle 方法是代码的“主体”，从可能的最大矩形（maxWordLength^2）开始，然后试着构建该大小的矩形。若构建失败，该方法会将最大面积减一，并尝试新的、较小的尺寸。由此，第一个成功构建的矩形必定是最大的。

```
Rectangle maxRectangle() {
  int maxSize = maxWordLength * maxWordLength;
  for (int z = maxSize; z > 0; z--) { // 从最大面积开始
    for (int i = 1; i <= maxWordLength; i ++ ) {
      if (z % i == 0) {
        int j = z / i;
        if (j <= maxWordLength) {
          /* 构造长度 i、高度 j 的矩形。注意 i * j = z */
          Rectangle rectangle = makeRectangle(i, j);
          if (rectangle != null) return rectangle;
        }
      }
```

```
    }
  }
  return null;
}
```

maxRectangle 又调用了 makeRectangle 方法，用于构造指定长度和高度的矩形。

```
Rectangle makeRectangle(int length, int height) {
  if (groupList[length-1] == null || groupList[height-1] == null) {
    return null;
  }

  /* 若不存在，就构建该单词长度的 trie */
  if (trieList[height - 1] == null) {
    LinkedList<String> words = groupList[height - 1].getWords();
    trieList[height - 1] = new Trie(words);
  }

  return makePartialRectangle(length, height, new Rectangle(length));
}
```

makePartialRectangle 方法真正负责构建矩形，参数为预期的最终长度和高度以及部分成形的矩形。如果矩形的高度已达到最后想要的高度，就直接查看每一列能否构成有效且完整的单词，然后返回。

否则，检查每一列字母能否构成有效前缀。如若不能，就立即中止，因为这个部分成形的矩形最后不可能构建出有效的矩形。

不过，如果到目前为止一切顺利，所有列都是有效的单词前缀，那么，就继续搜索相应长度的单词，追加至当前矩形的后面，然后进入递归试着以{追加中新单词的矩形}为基础构建矩形。

```
Rectangle makePartialRectangle(int l, int h, Rectangle rectangle) {
  if (rectangle.height == h) { // 检查矩形是否已完成
    if (rectangle.isComplete(l, h, groupList[h - 1])) {
      return rectangle;
    }
    return null;
  }

  /* 将所有列与 trie 比较，检查是否有效 */
  if (!rectangle.isPartialOK(l, trieList[h - 1])) {
    return null;
  }

  /* 迭代访问该长度的所有单词，并加入当前的部分矩形，然后试着递归构建出矩形 */
  for (int i = 0; i < groupList[l-1].length(); i++) {
    /* 当前矩形加上新单词构建新矩形 */
    Rectangle orgPlus = rectangle.append(groupList[l-1].getWord(i));

    /* 试着以这个新的、部分矩形构建新矩形 */
    Rectangle rect = makePartialRectangle(l, h, orgPlus);
    if (rect != null) {
      return rect;
    }
  }
  return null;
}
```

Rectangle 类代表一个部分或完整的单词矩形，可以调用 isPartialOk 方法来检查矩形到

目前为止是否有效（即每一列都是有效的单词前缀）。isComplete 方法的功能类似，不过只检查每一列是否为完整的单词。

```
public class Rectangle {
  public int height, length;
  public char[][] matrix;

  /* 构造一个“空”的矩形，长度是固定的，但高度会随着单词的加入而变化 */
  public Rectangle(int l) {
    height = 0;
    length = l;
  }

  /* 根据指定长度和高度的字符数组构造矩形，使用指定的字母矩阵
   * 表示（假定参数指定的长度和高度与数组参数的大小相符）*/
  public Rectangle(int length, int height, char[][] letters) {
    this.height = letters.length;
    this.length = letters[0].length;
    matrix = letters;
  }

  public char getLetter (int i, int j) { return matrix[i][j]; }
  public String getColumn(int i) { ... }

  /* 检查所有列是否都为有效。所有列已知为有效的，因为它们是直接从字典里取出的 */
  public boolean isComplete(int l, int h, WordGroup groupList) {
    if (height == h) {
      /* 检查每一列是否为字典里的单词 */
      for (int i = 0; i < l; i++) {
        String col = getColumn(i);
        if (!groupList.containsWord(col)) {
          return false;
        }
      }
      return true;
    }
    return false;
  }

  public boolean isPartialOK(int l, Trie trie) {
    if (height == 0) return true;
    for (int i = 0; i < l; i++ ) {
      String col = getColumn(i);
      if (!trie.contains(col)) {
        return false;
      }
    }
    return true;
  }

  /* 在当前矩形上追加 s 来新建 Rectangle */
  public Rectangle append(String s) { ... }
}
```

WordGroup 类是个简单的容器，包含某长度的所有单词。为方便查找，我们会将单词存储在散列表和 ArrayList 中。

WordGroup 中的列表由静态方法 createWordGroups 创建。

```
public class WordGroup {
  private HashMap<String, Boolean> lookup = new HashMap<String, Boolean>();
  private ArrayList<String> group = new ArrayList<String>();
```

```
    public boolean containsWord(String s) { return lookup.containsKey(s); }
    public int length() { return group.size(); }
    public String getWord(int i) { return group.get(i); }
    public ArrayList<String> getWords() { return group; }

    public void addWord (String s) {
      group.add(s);
      lookup.put(s, true);
    }

    public static WordGroup[] createWordGroups(String[] list) {
      WordGroup[] groupList;
      int maxWordLength = 0;
      /* 找出最长单词的长度 */
      for (int i = 0; i < list.length; i++) {
        if (list[i].length() > maxWordLength) {
          maxWordLength = list[i].length();
        }
      }

      /* 将字典里的单词按长度分组，相同长度的分为一组。
       * groupList[i]会包含一串单词，每个单词的长度为 length (i+1)t */
      groupList = new WordGroup[maxWordLength];
      for (int i = 0; i < list.length; i++) {
        /* 此处使用了 wordLength - 1 而非 wordLength。这是因为
         * 该数值被用作了索引，并不存在长度为 0 的单词 */
        int wordLength = list[i].length() - 1;
        if (groupList[wordLength] == null) {
          groupList[wordLength] = new WordGroup();
        }
        groupList[wordLength].addWord(list[i]);
      }
      return groupList;
    }
  }
```

此题完整代码（包括 Trie 和 TrieNode 的代码），可在本书所附的源码包里找出。注意，面对复杂如是的问题，你很可能只需要写出伪码即可。毕竟，要在这么短的时间内写出全部代码几乎是不可能的。

对于面试者，不论身处国内还是国外，本书都能提供指导性的帮助，号称“程序员面试红宝书”。同时，我也推荐技术面试官们前来阅读，借鉴硅谷成熟的技术招聘体系，从而挖掘积极面向未来的出色工程师。

—— 张云浩 力扣（LeetCode）CTO

在应聘时扬长避短，在工作中脱颖而出，依靠的是持之以恒的练习与积累。然而技术行业的机会稍纵即逝，这本书将会帮你迅速调整状态，明确目标，从而掌握更多的致胜筹码。

—— 王浩 京东高级技术总监

本书详细介绍了程序员从求职到面试过程中需要掌握的技能。一方面，本书结合算法题库，帮助你快速了解企业需求，让你能够信心百倍地迎接面试；另一方面，本书可以作为工具书，帮助你解决实际开发中的难题。

—— 张耀辉 腾讯云智慧农业总经理

本书是谷歌资深面试官的经验之作，全书紧扣面试环节，全面而详尽地介绍了程序员要为面试做哪些准备以及如何应对面试。书中对知名公司的面试题目进行了透彻的分析与讲解，能够帮助读者打牢基础、提升内功，在面试中过关斩将，获得心仪的工作。

图灵教育 | 图灵社区

分类建议 计算机/算法

人民邮电出版社网址：www.ptpress.com.cn