



测试用例：

- 功能测试（从数据流中读出奇数个数字；从数据流中读出偶数个数字）。
- 边界值测试（从数据流中读出0个、1个、2个数字）



本题考点：

- 考查应聘者对时间复杂度的分析能力。
- 考查应聘者对数据结构的理解程度。应聘者只有对各个常用数据容器的特点非常了解，知道它们的优缺点及适用场景，才能找出最优的解法。

面试题 42：连续子数组的最大和

题目：输入一个整型数组，数组里有正数也有负数。数组中的一个或连续多个整数组成一个子数组。求所有子数组的和的最大值。要求时间复杂度为 $O(n)$ 。

例如，输入的数组为 {1, -2, 3, 10, -4, 7, 2, -5}，和最大的子数组为 {3, 10, -4, 7, 2}，因此输出为该子数组的和 18。

看到这道题，很多人都能想到最直观的方法，即枚举数组的所有子数组并求出它们的和。一个长度为 n 的数组，总共有 $n(n+1)/2$ 个子数组。计算出所有子数组的和，最快也需要 $O(n^2)$ 时间。通常最直观的方法不会是最优的解法，面试官将提示我们还有更快的算法。

❖ 解法一：举例分析数组的规律

我们试着从头到尾逐个累加示例数组中的每个数字。初始化和为 0。第一步加上第一个数字 1，此时和为 1。第二步加上数字 -2，和就变成了 -1。第三步加上数字 3。我们注意到由于此前累加的和是 -1，小于 0，那如果用 -1 加上 3，得到的和是 2，比 3 本身还小。也就是说，从第一个数字开始的子数组的和会小于从第三个数字开始的子数组的和。因此，我们不用考虑从第一个数字开始的子数组，之前累加的和也被抛弃。

我们从第三个数字重新开始累加，此时得到的和是 3。第四步加 10，得到的和为 13。第五步加上 -4，和为 9。我们发现，由于 -4 是一个负数，因此累加 -4 之后得到的和比原来的和还要小。因此，我们要把之前得到的和 13 保存下来，因为它有可能是最大的子数组的和。第六步加上数字 7，9 加 7 的结果是 16，此时的和比之前最大的和 13 还要大，把最大的子数组的和由 13 更新为 16。第七步加上 2，累加得到的和为 18，同时更新最大子数组的和。第八步加上最后一个数字 -5，由于得到的和为 13，小于此前最大的和 18，因此，最终最大的子数组的和为 18，对应的子数组是 {3, 10, -4, 7, 2}。整个过程可以用表 5.3 总结如下。

表 5.3 计算数组{1, -2, 3, 10, -4, 7, 2, -5}中子数组的最大和的过程

| 步 骤 | 操 作 | 累加的子数组和 | 最大的子数组和 |
|-----|---------------|---------|---------|
| 1 | 加 1 | 1 | 1 |
| 2 | 加 -2 | -1 | 1 |
| 3 | 抛弃前面的和 -1，加 3 | 3 | 3 |
| 4 | 加 10 | 13 | 13 |
| 5 | 加 -4 | 9 | 13 |
| 6 | 加 7 | 16 | 16 |
| 7 | 加 2 | 18 | 18 |
| 8 | 加 -5 | 13 | 18 |

把过程分析清楚之后，我们就可以动手写代码了。下面是一段参考代码：

```
bool g_InvalidInput = false;

int FindGreatestSumOfSubArray(int *pData, int nLength)
{
    if((pData == nullptr) || (nLength <= 0))
    {
        g_InvalidInput = true;
        return 0;
    }

    g_InvalidInput = false;

    int nCurSum = 0;
    int nGreatestSum = 0x80000000;
    for(int i = 0; i < nLength; ++i)
    {
        if(nCurSum <= 0)
            nCurSum = pData[i];
        nCurSum += pData[i];
        if(nCurSum > nGreatestSum)
            nGreatestSum = nCurSum;
    }
}
```

```

        else
            nCurSum += pData[i];

        if(nCurSum > nGreatestSum)
            nGreatestSum = nCurSum;
    }

    return nGreatestSum;
}

```

面试的时候我们要考虑无效的输入，如输入的数组参数为空指针、数组长度小于等于0等情况。此时我们让函数返回什么数字？如果返回0，那我们又怎么区分子数组的和的最大值是0和无效输入这两种不同情况呢？因此，我们定义了一个全局变量来标记是否输入无效。

❖ 解法二：应用动态规划法

如果算法的功底足够扎实，那么我们还可以用动态规划的思想来分析这个问题。如果用函数 $f(i)$ 表示以第*i*个数字结尾的子数组的最大和，那么我们需要求出 $\max[f(i)]$ ，其中 $0 \leq i < n$ 。我们可用如下递归公式求 $f(i)$ ：

$$f(i) = \begin{cases} \text{pData}[i] & i = 0 \text{ 或者 } f(i-1) \leq 0 \\ f(i-1) + \text{pData}[i] & i \neq 0 \text{ 并且 } f(i-1) > 0 \end{cases}$$

这个公式的意义：当以第*i*-1个数字结尾的子数组中所有数字的和小于0时，如果把这个负数与第*i*个数累加，则得到的结果比第*i*个数字本身还要小，所以这种情况下以第*i*个数字结尾的子数组就是第*i*个数字本身（如表5.3中的第3步）。如果以第*i*-1个数字结尾的子数组中所有数字的和大于0，则与第*i*个数字累加就得到以第*i*个数字结尾的子数组中所有数字的和。

虽然通常我们用递归的方式分析动态规划的问题，但最终都会基于循环去编码。上述公式对应的代码和前面给出的代码一致。递归公式中的 $f(i)$ 对应的变量是nCurSum，而 $\max[f(i)]$ 就是nGreatestSum。因此，可以说这两种思路是异曲同工的。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/42_GreatestSumOfSubarrays



测试用例：

- 功能测试(输入的数组中有正数也有负数; 输入的数组中全是正数; 输入的数组中全是负数)。
- 特殊输入测试 (表示数组的指针为 `nullptr` 指针)。



本题考点：

- 考查应聘者对时间复杂度的理解。这道题如果应聘者给出时间复杂度为 $O(n^2)$ 甚至 $O(n^3)$ 的算法，则是不能通过面试的。
- 考查应聘者对动态规划的理解。如果应聘者熟练掌握了动态规划算法，那么他就能轻松地找到解题方案。如果没有想到用动态规划的思想，那么应聘者就需要仔细地分析累加子数组的和的过程，从而找到解题的规律。
- 考查应聘者思维的全面性。能否合理地处理无效的输入，对面试结果有很重要的影响。

面试题 43：1~n 整数中 1 出现的次数

题目：输入一个整数 n ，求 $1 \sim n$ 这 n 个整数的十进制表示中 1 出现的次数。例如，输入 12， $1 \sim 12$ 这些整数中包含 1 的数字有 1、10、11 和 12，1 一共出现了 5 次。

❖ 不考虑时间效率的解法，靠它想拿 Offer 有点难

如果在面试的时候碰到这个问题，则应聘者大多能想到最直观的方法，也就是累加 $1 \sim n$ 中每个整数 1 出现的次数。我们可以每次通过对 10 求余数判断整数的个位数字是不是 1。如果这个数字大于 10，则除以 10 之后再判断个位数字是不是 1。基于这种思路，我们不难写出如下代码：

```
int NumberOf1Between1AndN(unsigned int n)
{
    int number = 0;

    for(unsigned int i = 1; i <= n; ++ i)
        number += NumberOf1(i);
    return number;
}
```

```

int NumberOf1(unsigned int n)
{
    int number = 0;
    while(n)
    {
        if(n % 10 == 1)
            number++;

        n = n / 10;
    }

    return number;
}

```

在上述思路中，我们对每个数字都要做除法和求余运算，以求出该数字中 1 出现的次数。如果输入数字 n , n 有 $O(\log n)$ 位，我们需要判断每一位是不是 1，那么它的时间复杂度是 $O(n \log n)$ 。当输入的 n 非常大的时候，需要大量的计算，运算效率不高。面试官不会满意这种算法，我们仍然需要努力。

❖ 从数字规律着手明显提高时间效率的解法，能让面试官耳目一新

如果希望不用计算每个数字的 1 的个数，那就只能去寻找 1 在数字中出现的规律了。为了找到规律，我们不妨用一个稍微大一点的数字如 21345 作为例子来分析。我们把 1~21345 的所有数字分为两段：一段是 1~1345；另一段是 1346~21345。

我们先看 1346~21345 中 1 出现的次数。1 的出现分为两种情况。首先分析 1 出现在最高位（本例中是万位）的情况。在 1346~21345 的数字中，1 出现在 10000~19999 这 10000 个数字的万位中，一共出现了 10000 (10^4) 次。

值得注意的是，并不是对所有 5 位数而言在万位出现的次数都是 10000 次。对于万位是 1 的数字如输入 12345，1 只出现在 10000~12345 的万位，出现的次数不是 10^4 次，而是 2346 次，也就是除去最高数字之后剩下的数字再加上 1 ($2345+1=2346$ 次)。

接下来分析 1 出现在除最高位之外的其他 4 位数中的情况。例子中 1346~21345 这 20000 个数字中后 4 位中 1 出现的次数是 8000 次。由于最高位是 2，我们可以再把 1346~21345 分成两段：1346~11345 和 11346~21345。每一段剩下的 4 位数字中，选择其中一位是 1，其余三位可以在 0~

9这10个数字中任意选择，因此根据排列组合原则，总共出现的次数是 $2 \times 4 \times 10^3 = 8000$ 次。

至于在1~1345中1出现的次数，我们就可以用递归求得了。这也是我们为什么要把1~21345分成1~1345和1346~21345两段的原因。因为把21345的最高位去掉就变成1345，便于我们采用递归的思路。

基于前面的分析，我们可以写出如下代码（为了编程方便，我们先把数字转换成字符串）：

```
int NumberOf1Between1AndN(int n)
{
    if(n <= 0)
        return 0;

    char strN[50];
    sprintf(strN, "%d", n);

    return NumberOf1(strN);
}

int NumberOf1(const char* strN)
{
    if(!strN || *strN < '0' || *strN > '9' || *strN == '\0')
        return 0;

    int first = *strN - '0';
    unsigned int length = static_cast<unsigned int>(strlen(strN));

    if(length == 1 && first == 0)
        return 0;

    if(length == 1 && first > 0)
        return 1;

    // 假设 strN 是 "21345"
    // numFirstDigit 是数字 10000~19999 的第一位中的数目
    int numFirstDigit = 0;
    if(first > 1)
        numFirstDigit = PowerBase10(length - 1);
    else if(first == 1)
        numFirstDigit = atoi(strN + 1) + 1;

    // numOtherDigits 是 1346~21345 除第一位之外的数位中的数目
    int numOtherDigits = first * (length - 1) * PowerBase10(length - 2);
    // numRecursive 是 1~1345 中的数目
    int numRecursive = NumberOf1(strN + 1);

    return numFirstDigit + numOtherDigits + numRecursive;
}

int PowerBase10(unsigned int n)
```

```

{
    int result = 1;
    for(unsigned int i = 0; i < n; ++ i)
        result *= 10;

    return result;
}

```

这种思路是每次去掉最高位进行递归，递归的次数和位数相同。一个数字 n 有 $O(\log n)$ 位，因此这种思路的时间复杂度是 $O(\log n)$ ，比前面的原始方法要好很多。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/43_NumberOf1



测试用例：

- 功能测试（输入 5、10、55、99 等）。
- 边界值测试（输入 0、1 等）。
- 性能测试（输入较大的数字，如 10000、21235 等）。



本题考点：

- 考查应聘者做优化的激情和能力。最原始的方法大部分应聘者都能想到。当面试官提示还有更快的方法之后，应聘者千万不要轻易放弃尝试。虽然想出时间复杂度为 $O(\log n)$ 的方法不容易，但应聘者要展示自己追求更快算法的激情，多尝试不同的方法，必要的时候可以要求面试官给出提示，但不能轻易说自己想不出来并且放弃努力。
- 考查应聘者面对复杂问题的思维能力。要想找到时间复杂度为 $O(\log n)$ 的方法，应聘者需要有严密的数学思维能力，并且还要通过分析具体例子一步步找到通用的规律。这些能力在实际工作中面对复杂问题的时候都非常有用。

面试题 44：数字序列中某一位的数字

题目：数字以 0123456789101112131415…的格式序列化到一个字符序列中。在这个序列中，第 5 位（从 0 开始计数）是 5，第 13 位是 1，第 19 位是 4，等等。请写一个函数，求任意第 n 位对应的数字。

在面试的时候，很多应聘者都能想到最直观的方法，也就是从 0 开始逐一枚举每个数字。每枚举一个数字的时候，求出该数字是几位数（如 15 是 2 位数、9323 是 4 位数），并把该数字的位数和前面所有数字的位数累加。如果位数之和仍然小于或者等于输入 n ，则继续枚举下一个数字。当累加的数位大于 n 时，那么第 n 位数字一定在这个数字里，我们再从该数字中找出对应的那一位。

还有没有更快一些的办法，比如我们是不是可以找出某些规律从而跳过若干数字？接下来用一个具体的例子来分析如何解决这个问题。比如，序列的第 1001 位是什么？

序列的前 10 位是 0~9 这 10 个只有一位的数字。显然第 1001 位在这 10 个数字之后，因此这 10 个数字可以直接跳过。我们再从后面紧跟着的序列中找第 991（ $991=1001-10$ ）位的数字。

接下来 180 位数字是 90 个 10~99 的两位数。由于 $991 > 180$ ，所以第 991 位在所有的两位数之后。我们再跳过 90 个两位数，继续从后面找 881（ $881=991-180$ ）位。

接下来的 2700 位是 900 个 100~999 的三位数。由于 $811 < 2700$ ，所以第 811 位是某个三位数中的一位。由于 $811 = 270 \times 3 + 1$ ，这意味着第 811 位是从 100 开始的第 270 个数字即 370 的中间一位，也就是 7。

我们可以用如下代码表达这种思路：

```
int digitAtIndex(int index)
{
    if(index < 0)
        return -1;

    int digits = 1;
    while(true)
    {
        int numbers = countOfIntegers(digits);
        if(index < numbers * digits)
            return digitAtIndex(index, digits);

        index -= digits * numbers;
    }
}
```

```

        digits++;
    }

    return -1;
}

```

下面的函数 `countOfIntegers` 得到 m 位的数字总共有多少个？例如，输入 2，则返回两位数（10~99）的个数 90；输入 3，则返回三位数（100~999）的个数 900。

```

int countOfIntegers(int digits)
{
    if(digits == 1)
        return 10;

    int count = (int) std::pow(10, digits - 1);
    return 9 * count;
}

```

当我们知道要找的那一位数字位于某 m 位数之中后，就可以用如下函数找出那一位数字：

```

int digitAtIndex(int index, int digits)
{
    int number = beginNumber(digits) + index / digits;
    int indexFromRight = digits - index % digits;
    for(int i = 1; i < indexFromRight; ++i)
        number /= 10;
    return number % 10;
}

```

在上述函数中，我们需要知道 m 位数的第一个数字。例如，第一个两位数是 10，第一个三位数是 100。第一个 m 位数可以用如下函数求得：

```

int beginNumber(int digits)
{
    if(digits == 1)
        return 0;

    return (int) std::pow(10, digits - 1);
}

```



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/blob/master/44_DigitsInSequence/DigitsInSequence.cpp



测试用例：

- 功能测试（输入 10、190、1000 等）。
- 边界值测试（输入 0、1 等）。



本题考点：

- 考查应聘者做优化的激情和能力。最直观的方法很多应聘者都能想到。当面试官提示还有更快的方法之后，应聘者千万不要轻易放弃尝试。虽然想出好的方法不容易，但应聘者要展示自己追求更快算法的激情，多尝试不同的方法，必要的时候可以要求面试官给出提示，但不能轻易说自己想不出来并且放弃努力。
- 考查应聘者面对复杂问题的思维能力。应聘者需要有严密的数学思维能力，并且还要通过分析具体例子一步步找到通用的规律，才能想出好的算法。这些能力在实际工作中面对复杂问题的时候都非常有用。

面试题 45：把数组排成最小的数

题目：输入一个正整数数组，把数组里所有数字拼接起来排成一个数，打印能拼接出的所有数字中最小的一个。例如，输入数组{3,32,321}，则打印出这 3 个数字能排成的最小数字 321323。

这道题最直接的解法应该是先求出这个数组中所有数字的全排列，然后把每个排列拼起来，最后求出拼起来的数字的最小值。求数组的排列和面试题 38 “字符串的排列” 非常类似，这里不再详细介绍。根据排列组合的知识， n 个数字总共有 $n!$ 个排列。我们再来看一种更快的算法。

这道题其实是希望我们能找到一个排序规则，数组根据这个规则排序之后能排成一个最小的数字。要确定排序规则，就要比较两个数字，也就是给出两个数字 m 和 n ，我们需要确定一个规则判断 m 和 n 哪个应该排在前面，而不是仅仅比较这两个数字的值哪个更大。

根据题目的要求，两个数字 m 和 n 能拼接成数字 mn 和 nm 。如果 $mn < nm$ ，那么我们应该打印出 mn ，也就是 m 应该排在 n 的前面，我们定义此时 m 小于 n ；反之，如果 $nm < mn$ ，则我们定义 n 小于 m ；如果 $mn = nm$ ，则 m 等

于 n 。在下文中，符号“<”、“>”及“=”表示常规意义的数值的大小关系，而文字“大于”、“小于”、“等于”表示我们新定义的大小关系。

接下来考虑怎么去拼接数字，即给出数字 m 和 n ，怎么得到数字 mn 和 nm 并比较它们的大小。直接用数值去计算不难办到，但需要考虑的一个潜在问题就是 m 和 n 都在 int 型能表达的范围内，但把它们拼接起来的数字 mn 和 nm 用 int 型表示就有可能溢出了，所以这还是一个隐形的大数问题。

一个非常直观的解决大数问题的方法就是把数字转换成字符串。另外，由于把数字 m 和 n 拼接起来得到 mn 和 nm ，它们的位数肯定是相同的，因此比较它们的大小只需要按照字符串大小的比较规则就可以了。

基于这种思路，我们可以写出如下代码：

```
const int g_MaxNumberLength = 10;

char* g_StrCombine1 = new char[g_MaxNumberLength * 2 + 1];
char* g_StrCombine2 = new char[g_MaxNumberLength * 2 + 1];

void PrintMinNumber(int* numbers, int length)
{
    if(numbers == nullptr || length <= 0)
        return;
    char** strNumbers = (char**)(new int[length]);
    for(int i = 0; i < length; ++i)
    {
        strNumbers[i] = new char[g_MaxNumberLength + 1];
        sprintf(strNumbers[i], "%d", numbers[i]);
    }

    qsort(strNumbers, length, sizeof(char*), compare);

    for(int i = 0; i < length; ++i)
        printf("%s", strNumbers[i]);
    printf("\n");

    for(int i = 0; i < length; ++i)
        delete[] strNumbers[i];
    delete[] strNumbers;
}

int compare(const void* strNumber1, const void* strNumber2)
{
    strcpy(g_StrCombine1, *(const char**)(strNumber1));
    strcat(g_StrCombine1, *(const char**)(strNumber2));

    strcpy(g_StrCombine2, *(const char**)(strNumber2));
    strcat(g_StrCombine2, *(const char**)(strNumber1));

    return strcmp(g_StrCombine1, g_StrCombine2);
}
```

在上述代码中，我们先把数组中的整数转换成字符串，然后在函数 `compare` 中定义比较规则，并根据该规则调用库函数 `qsort` 排序，最后把排序好的数组中的数字依次打印出来，就是该数组中数字能拼接出来的最小数字。这种思路的时间复杂度和 `qsort` 的时间复杂度相同，也就是 $O(n \log n)$ ，这比用 $n!$ 的时间求出所有排列的思路要好很多。

在上述思路中，我们定义了一种新的比较两个数字大小的规则，这种规则是不是有效的？另外，我们只是定义了比较两个数字大小的规则，却用它来排序一个含有多个数字的数组，最终拼接数组中的所有数字得到的是不是真的就是最小的数字？一些严格的面试官还会要求我们给出严格的数学证明，以确保我们的解决方案是正确的。

我们首先证明之前定义的比较两个数字大小的规则是有效的。一个有效的比较规则需要 3 个条件：自反性、对称性和传递性。我们分别予以证明。

(1) 自反性：显然有 $aa=aa$ ，所以 a 等于 a 。

(2) 对称性：如果 a 小于 b ，则 $ab < ba$ ，所以 $ba > ab$ ，因此 b 大于 a 。

(3) 传递性：如果 a 小于 b ，则 $ab < ba$ 。假设 a 和 b 用十进制表示时分别有 l 位和 m 位，于是 $ab = a \times 10^m + b$ ， $ba = b \times 10^l + a$ 。

$$\begin{aligned} ab < ba &\rightarrow a \times 10^m + b < b \times 10^l + a \rightarrow a \times 10^m - a < b \times 10^l - b \rightarrow \\ a(10^m - 1) &< b(10^l - 1) \rightarrow a/(10^l - 1) < b/(10^m - 1) \end{aligned}$$

同理，如果 b 小于 c ，则 $bc < cb$ 。假设 c 用十进制表示时有 n 位，和前面的证明过程一样，可以得到 $b/(10^m - 1) < c/(10^n - 1)$ 。

$$\begin{aligned} a/(10^l - 1) &< b/(10^m - 1) \text{ 并且 } b/(10^m - 1) < c/(10^n - 1) \rightarrow a/(10^l - 1) < c/(10^n - 1) \rightarrow \\ a(10^n - 1) &< c(10^l - 1) \rightarrow a \times 10^n + c < c \times 10^l + a \rightarrow ac < ca \rightarrow a \text{ 小于 } c \end{aligned}$$

于是我们证明了这种比较规则满足自反性、对称性和传递性，是一种有效的比较规则。接下来我们证明根据这种比较规则把数组排序之后，把数组中的所有数字拼接起来得到的数字的确是最小的。直接证明不是很容易，我们不妨用反证法来证明。

我们把 n 个数按照前面的排序规则排序之后，表示为 $A_1 A_2 A_3 \cdots A_n$ 。假设这样拼接出来的数字并不是最小的，即至少存在两个 x 和 y ($0 < x < y < n$)，交换第 x 个数和第 y 个数后， $A_1 A_2 \cdots A_y \cdots A_x \cdots A_n < A_1 A_2 \cdots A_x \cdots A_y \cdots A_n$ 。

由于 $A_1A_2\cdots A_x\cdots A_y\cdots A_n$ 是按照前面的规则排好的序列，所以有 A_x 小于 A_{x+1} 小于 A_{x+2} 小于…小于 A_{y-2} 小于 A_{y-1} 小于 A_y 。

由于 A_{y-1} 小于 A_y ，所以 $A_{y-1}A_y < A_yA_{y-1}$ 。我们在序列 $A_1A_2\cdots A_x\cdots A_{y-1}A_y\cdots A_n$ 中交换 A_{y-1} 和 A_y ，有 $A_1A_2\cdots A_x\cdots A_{y-1}A_y\cdots A_n < A_1A_2\cdots A_x\cdots A_yA_{y-1}\cdots A_n$ （这个实际上也需要证明，感兴趣的读者可以自己试着证明）。我们就这样一直把 A_y 和前面的数字交换，直到和 A_x 交换为止。于是就有 $A_1A_2\cdots A_x\cdots A_{y-1}A_y\cdots A_n < A_1A_2\cdots A_x\cdots A_yA_{y-1}\cdots A_n < A_1A_2\cdots A_x\cdots A_yA_{y-2}A_{y-1}\cdots A_n < \cdots < A_1A_2\cdots A_yA_x\cdots A_{y-2}A_{y-1}\cdots A_n$ 。

同理，由于 A_x 小于 A_{x+1} ，所以 $A_xA_{x+1} < A_{x+1}A_x$ 。我们在序列 $A_1A_2\cdots A_yA_xA_{x+1}\cdots A_{y-2}A_{y-1}\cdots A_n$ 中只交换 A_x 和 A_{x+1} ，有 $A_1A_2\cdots A_yA_xA_{x+1}\cdots A_{y-2}A_{y-1}\cdots A_n < A_1A_2\cdots A_yA_{x+1}A_x\cdots A_{y-2}A_{y-1}\cdots A_n$ 。接下来一直拿 A_x 和它后面的数字交换，直到和 A_{y-1} 交换为止。于是就有 $A_1A_2\cdots A_yA_xA_{x+1}\cdots A_{y-2}A_{y-1}\cdots A_n < A_1A_2\cdots A_yA_{x+1}A_x\cdots A_{y-2}A_{y-1}\cdots A_n < \cdots < A_1A_2\cdots A_yA_{x+1}A_x\cdots A_{y-2}A_{y-1}\cdots A_n$ 。

所以 $A_1A_2\cdots A_x\cdots A_y\cdots A_n < A_1A_2\cdots A_y\cdots A_x\cdots A_n$ ，这和我们的假设 $A_1A_2\cdots A_y\cdots A_x\cdots A_n < A_1A_2\cdots A_x\cdots A_y\cdots A_n$ 相矛盾。

所以假设不成立，我们的算法是正确的。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/45_SortArrayForMinNumber



测试用例：

- 功能测试（输入的数组中有多个数字；输入的数组中的数字有重复的数位；输入的数组中只有一个数字）。
- 特殊输入测试（表示数组的指针为 `nullptr` 指针）。



本题考点：

- 本题有两个难点：第一个难点是想出一种新的比较规则来排序一个

数组；第二个难点是证明这个比较规则是有效的，并且证明根据这个规则排序之后，把数组中所有数字拼接起来得到的数字是最小的。要想解决这两个难点，要求应聘者有很强的数学功底和逻辑思维能力。

- 考查应聘者解决大数问题的能力。应聘者在面试的时候要意识到，把两个 int 型的整数拼接起来得到的数字可能会超出 int 型数字能够表达的范围，从而导致数字溢出。我们可以用字符串表示数字，这样就能简捷地解决大数问题。

面试题 46：把数字翻译成字符串

题目：给定一个数字，我们按照如下规则把它翻译为字符串：0 翻译成“a”，1 翻译成“b”，……，11 翻译成“l”，……，25 翻译成“z”。一个数字可能有多个翻译。例如，12258 有 5 种不同的翻译，分别是“bccfi”、“bwfi”、“bczi”、“mcfi” 和 “mzi”。请编程实现一个函数，用来计算一个数字有多少种不同的翻译方法。

我们以 12258 为例分析如何从数字的第一位开始一步步计算不同翻译方法的数目。我们有两种不同的选择来翻译第一位数字 1。第一种选择是数字 1 单独翻译成“b”，后面剩下数字 2258；第二种选择是 1 和紧挨着的 2 一起翻译成“m”，后面剩下数字 258。

当最开始的一个或者两个数字被翻译成一个字符之后，我们接着翻译后面剩下的数字。显然，我们可以写一个递归函数来计算翻译的数目。

我们定义函数 $f(i)$ 表示从第 i 位数字开始的不同翻译的数目，那么 $f(i)=f(i+1)+g(i,i+1)\times f(i+2)$ 。当第 i 位和第 $i+1$ 位两位数字拼接起来的数字在 10~25 的范围内时，函数 $g(i,i+1)$ 的值为 1；否则为 0。

尽管我们用递归的思路来分析这个问题，但由于存在重复的子问题，递归并不是解决这个问题的最佳方法。还是以 12258 为例。如前所述，翻译 12258 可以分解成两个子问题：翻译 1 和 2258，以及翻译 12 和 258。接下来我们翻译第一个子问题中剩下的 2258，同样也可以分解成两个子问题：翻译 2 和 258，以及翻译 22 和 58。注意到子问题翻译 258 重复出现了。

递归从最大的问题开始自上而下解决问题。我们也可以从最小的子问题开始自下而上解决问题，这样就可以消除重复的子问题。也就是说，我

们从数字的末尾开始，然后从右到左翻译并计算不同翻译的数目。下面是参考代码：

```

int GetTranslationCount(int number)
{
    if(number < 0)
        return 0;

    string numberInString = to_string(number);
    return GetTranslationCount(numberInString);
}

int GetTranslationCount(const string& number)
{
    int length = number.length();
    int* counts = new int[length];
    int count = 0;

    for(int i = length - 1; i >= 0; --i)
    {
        count = 0;
        if(i < length - 1)
            count = counts[i + 1];
        else
            count = 1;

        if(i < length - 1)
        {
            int digit1 = number[i] - '0';
            int digit2 = number[i + 1] - '0';
            int converted = digit1 * 10 + digit2;
            if(converted >= 10 && converted <= 25)
            {
                if(i < length - 2)
                    count += counts[i + 2];
                else
                    count += 1;
            }
        }

        counts[i] = count;
    }

    count = counts[0];
    delete[] counts;
}

return count;
}

```



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/46_TranslateNumbersToStrings



测试用例：

- 功能测试（只有一位数字；包含多位数字）。
- 特殊输入测试（负数；0；包含25、26的数字）。



本题考点：

- 考查应聘者分析问题的能力。应聘者能够问题中分析出递归的表达式是解决这个问题的前提。
- 考查应聘者对递归及时间效率的理解。如果只是能够把递归分析转换为递归代码，则应聘者不一定能够通过这道题的面试。面试官期待应聘者能够用基于循环的代码来避免不必要的重复计算。

面试题 47：礼物的最大价值

题目：在一个 $m \times n$ 的棋盘的每一格都放有一个礼物，每个礼物都有一定的价值（价值大于0）。你可以从棋盘的左上角开始拿格子里的礼物，并每次向左或者向下移动一格，直到到达棋盘的右下角。给定一个棋盘及其上面的礼物，请计算你最多能拿到多少价值的礼物？

例如，在下面的棋盘中，如果沿着带下画线的数字的线路（1、12、5、7、7、16、5），那么我们能拿到最大价值为53的礼物。

| | | | |
|-----------|----|-----------|----|
| <u>1</u> | 10 | 3 | 8 |
| <u>12</u> | 2 | 9 | 6 |
| <u>5</u> | 7 | 4 | 11 |
| 3 | 7 | <u>16</u> | 5 |

这是一个典型的能用动态规划解决的问题。我们先用递归的思路来分析。我们先定义第一个函数 $f(i,j)$ 表示到达坐标为 (i,j) 的格子时能拿到的礼物总和的最大值。根据题目要求，我们有两种可能的途径到达坐标为 (i,j) 的格

子：通过格子 $(i-1, j)$ 或者 $(i, j-1)$ 。所以 $f(i, j) = \max(f(i-1, j), f(i, j-1)) + \text{gift}[i, j]$ 。 $\text{gift}[i, j]$ 表示坐标为 (i, j) 的格子里礼物的价值。

尽管我们用递归来分析问题，但由于有大量重复的计算，导致递归的代码并不是最优的。相对而言，基于循环的代码效率要高很多。为了缓存中间计算结果，我们需要一个辅助的二维数组。数组中坐标为 (i, j) 的元素表示到达坐标为 (i, j) 的格子时能拿到的礼物价值总和的最大值。

下面是参考代码：

```
int getMaxValue_solution1(const int* values, int rows, int cols)
{
    if(values == nullptr || rows <= 0 || cols <= 0)
        return 0;

    int** maxValues = new int*[rows];
    for(int i = 0; i < rows; ++i)
        maxValues[i] = new int[cols];

    for(int i = 0; i < rows; ++i)
    {
        for(int j = 0; j < cols; ++j)
        {
            int left = 0;
            int up = 0;

            if(i > 0)
                up = maxValues[i - 1][j];

            if(j > 0)
                left = maxValues[i][j - 1];

            maxValues[i][j] = std::max(left, up) + values[i * cols + j];
        }
    }

    int maxValue = maxValues[rows - 1][cols - 1];

    for(int i = 0; i < rows; ++i)
        delete[] maxValues[i];
    delete[] maxValues;

    return maxValue;
}
```

接下来我们考虑进一步的优化。前面我们提到，到达坐标为 (i, j) 的格子时能够拿到的礼物的最大价值只依赖坐标为 $(i-1, j)$ 和 $(i, j-1)$ 的两个格子，因此第 $i-2$ 行及更上面的所有格子礼物的最大价值实际上没有必要保存下来。我们可以用一个一维数组来替代前面代码中的二维矩阵 `maxValues`。该一维数组的长度为棋盘的列数 n 。当我们计算到达坐标为 (i, j) 的格子时能够拿到

的礼物的最大价值 $f(i,j)$, 数组中前 j 个数字分别是 $f(i,0), f(i,1), \dots, f(i,j-1)$, 数组从下标为 j 的数字开始到最后一个数字, 分别为 $f(i-1,j), f(i-1,j+1), \dots, f(i-1,n-1)$ 。也就是说, 该数组前面 j 个数字分别是当前第 i 行前面 j 个格子礼物的最大价值, 而之后的数字分别保存前面第 $i-1$ 行 $n-j$ 个格子礼物的最大价值。

优化之后的代码如下:

```
int getMaxValue_solution2(const int* values, int rows, int cols)
{
    if(values == nullptr || rows <= 0 || cols <= 0)
        return 0;

    int* maxValues = new int[cols];
    for(int i = 0; i < rows; ++i)
    {
        for(int j = 0; j < cols; ++j)
        {
            int left = 0;
            int up = 0;

            if(i > 0)
                up = maxValues[j];

            if(j > 0)
                left = maxValues[j - 1];

            maxValues[j] = std::max(left, up) + values[i * cols + j];
        }
    }

    int maxValue = maxValues[cols - 1];
    delete[] maxValues;
    return maxValue;
}
```



源代码:

本题完整的源代码:

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/47_MaxValueOfGifts



测试用例：

- 功能测试（多行多列的矩阵；一行或者一列的矩阵；只有一个数字的矩阵）。
- 特殊输入测试（指向矩阵数组的指针为 `nullptr`）。



本题考点：

- 考查应聘者用动态规划分析问题的能力。应聘者能够熟练应用动态规划分析问题是解答这道面试题的前提。
- 考查应聘者对递归及时间效率的理解。如果只是能够把递归分析转换为递归代码，则应聘者不一定能够通过这道题的面试。面试官期待应聘者能够用基于循环的代码来避免不必要的重复计算。

面试题 48：最长不含重复字符的子字符串

题目：请从字符串中找出一个最长的不包含重复字符的子字符串，计算该最长子字符串的长度。假设字符串中只包含'a'~'z'的字符。例如，在字符串"arabcacfr"中，最长的不含重复字符的子字符串是"acfr"，长度为4。

我们不难找出字符串的所有子字符串，然后就可以判断每个子字符串中是否包含重复的字符。这种蛮力法唯一的缺点就是效率。一个长度为 n 的字符串有 $O(n^2)$ 个子字符串，我们需要 $O(n)$ 的时间判断一个子字符串中是否包含重复的字符，因此该解法的总的时间效率是 $O(n^3)$ 。

接下来我们用动态规划算法来提高效率。首先定义函数 $f(i)$ 表示以第 i 个字符为结尾的不包含重复字符的子字符串的最长长度。我们从左到右逐一扫描字符串中的每个字符。当我们计算以第 i 个字符为结尾的不包含重复字符的子字符串的最长长度 $f(i)$ 时，我们已经知道 $f(i-1)$ 了。

如果第 i 个字符之前没有出现过，那么 $f(i)=f(i-1)+1$ 。例如，在字符串"arabcacfr"中，显然 $f(0)$ 等于 1。在计算 $f(1)$ 时，下标为 1 的字符'r'之前没有出现过，因此 $f(1)$ 等于 2，即 $f(1)=f(0)+1$ 。到目前为止，最长的不含重复字符的子字符串是"ar"。

如果第 i 个字符之前已经出现过，那情况就要复杂一点了。我们先计算

第 i 个字符和它上次出现在字符串中的位置的距离，并记为 d ，接着分两种情形分析。第一种情形是 d 小于或者等于 $f(i-1)$ ，此时第 i 个字符上次出现在 $f(i-1)$ 对应的最长子字符串之中，因此 $f(i)=d$ 。同时这也意味着在第 i 个字符出现两次所夹的子字符串中再也没有其他重复的字符了。在前面的例子中，我们继续计算 $f(2)$ ，即以下标为 2 的字符'a'为结尾的不含重复字符的子字符串的最长长度。我们注意到字符'a'在之前出现过，该字符上一次出现在下标为 0 的位置，它们之间的距离 d 为 2，也就是字符'a'出现在 $f(1)$ 对应的最长不含重复字符的子字符串"ar"中，此时 $f(2)=d$ ，即 $f(2)=2$ ，对应的最长不含重复字符的子字符串是"ra"。

第二种情形是 d 大于 $f(i-1)$ ，此时第 i 个字符上次出现在 $f(i-1)$ 对应的最长子字符串之前，因此仍然有 $f(i)=f(i-1)+1$ 。我们接下来分析以字符串"arabcacfr"最后一个字符'r'为结尾的最长不含重复字符的子字符串的长度，即求 $f(8)$ 。以它前一个字符'f'为结尾的最长不含重复字符的子字符串是"acf"，因此 $f(7)=3$ 。我们注意到最后一个字符'r'之前在字符串"arabcacfr"中出现过，上一次出现在下标为 1 的位置，因此两次出现的距离 d 等于 7，大于 $f(7)$ 。这说明上一个字符'r'不在 $f(7)$ 对应的最长不含重复字符的子字符串"acf"中，此时把字符'r'拼接到"acf"的后面也不会出现重复字符。因此 $f(8)=f(7)+1$ ，即 $f(8)=4$ ，对应的最长不含重复字符的子字符串是"acfr"。

下面是参考代码：

```
int longestSubstringWithoutDuplication(const std::string& str)
{
    int curLength = 0;
    int maxLength = 0;

    int* position = new int[26];
    for(int i = 0; i < 26; ++i)
        position[i] = -1;

    for(int i = 0; i < str.length(); ++i)
    {
        int prevIndex = position[str[i] - 'a'];
        if(prevIndex < 0 || i - prevIndex > curLength)
            ++curLength;
        else
        {
            if(curLength > maxLength)
                maxLength = curLength;

            curLength = i - prevIndex;
        }
        position[str[i] - 'a'] = i;
    }
}
```

```

    }

    if(curLength > maxLength)
        maxLength = curLength;

    delete[] position;
    return maxLength;
}

```

在上述代码中，我们创建了一个长度为 26 的数组 `position` 用来存储每个字符上次出现在字符串中位置的下标。该数组所有元素的值都初始化为 -1，负数表示该元素对应的字符在字符串中还没有出现过。我们在扫描字符串时遇到某个字符，就把该字符在字符串中的位置存储到数组对应的元素中。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/48_LongestSubstringWithoutDup



测试用例：

- 功能测试（包含多个字符的字符串；只有一个字符的字符串；所有字符都唯一的字符串；所有字符都相同的字符串）。
- 特殊输入测试（空字符串）。



本题考点：

- 考查应聘者用动态规划分析问题的能力。应聘者能够熟练应用动态规划分析问题是解答这道面试题的前提。
- 考查应聘者对递归及时间效率的理解。如果只是能够把递归分析转换为递归代码，则应聘者不一定能够通过这道题的面试。面试官期待应聘者能够用基于循环的代码来避免不必要的重复计算。

5.3 时间效率与空间效率的平衡

硬件的发展一直遵循摩尔定律，内存的容量基本上每隔 18 个月就会翻一番。由于内存的容量增加迅速，在软件开发的过程中我们允许以牺牲一定的空间为代价来优化时间性能，以尽可能地缩短软件的响应时间。这就是我们通常所说的“以空间换时间”。

在面试的时候，如果我们分配少量的辅助空间来保存计算的中间结果以提高时间效率，则通常是可以被接受的。本书中收集的面试题中有不少这种类型的题目，比如在面试题 49 “丑数” 中用一个数组按照从小到大的顺序保存已经求出的丑数；在面试题 60 “ n 个骰子的点数” 中交替使用两个数组求骰子每个点数出现的次数。

值得注意的是，“以空间换时间”的策略并不一定都是可行的，在面试的时候要具体问题具体分析。我们都应该知道在 n 个无序的元素里执行查找操作，需要 $O(n)$ 的时间。但如果我们将这些元素放进一个哈希表，那么在哈希表内就能实现时间复杂度为 $O(1)$ 的查找。但同时实现一个哈希表是有空间消耗的，是不是值得以多消耗空间为前提来换取时间性能的提升，我们需要根据实际情况仔细权衡。在面试题 50 “第一个只出现一次的字符” 中，我们用数组实现了一个简易哈希表，有了这个哈希表就能实现在 $O(1)$ 时间内查找任意字符。对于 ASCII 码的字符而言，总共只有 256 个字符，因此只需要 1KB 的辅助内存。这点内存消耗对于绝大多数硬件来说是完全可以接受的。但如果是 16 位的 Unicode 的字符，创建这样一个长度为 2^{16} 的整型数组需要 4×2^{16} 也就是 256KB 的内存。这对于个人计算机来说也是可以接受的，但对于一些嵌入式的开发就要慎重了。

很多时候时间效率和空间效率存在类似于鱼与熊掌的关系，我们需要在它们之间有所取舍。在面试的时候究竟是“以时间换空间”还是“以空间换时间”，我们可以和面试官进行探讨。多和面试官进行这方面的讨论是很有必要的，这既能显示我们的沟通能力，又能展示我们对软件性能全方位的把握能力。

面试题 49：丑数

题目：我们把只包含因子 2、3 和 5 的数称作丑数（Ugly Number）。求按从小到大的顺序的第 1500 个丑数。例如，6、8 都是丑数，但 14 不是，因为它包含因子 7。习惯上我们把 1 当作第一个丑数。

❖ 逐个判断每个整数是不是丑数的解法，直观但不够高效

所谓一个数 m 是另一个数 n 的因子，是指 n 能被 m 整除，也就是 $n \% m = 0$ 。根据丑数的定义，丑数只能被 2、3 和 5 整除。也就是说，如果一个数能被 2 整除，就连续除以 2；如果能被 3 整除，就连续除以 3；如果能被 5 整除，就除以连续 5。如果最后得到的是 1，那么这个数就是丑数；否则不是。

因此，我们可以写出下面的函数来判断一个数是不是丑数：

```
bool IsUgly(int number)
{
    while(number % 2 == 0)
        number /= 2;
    while(number % 3 == 0)
        number /= 3;
    while(number % 5 == 0)
        number /= 5;

    return (number == 1) ? true : false;
}
```

接下来，我们只需要按照顺序判断每个整数是不是丑数，即：

```
int GetUglyNumber(int index)
{
    if(index <= 0)
        return 0;

    int number = 0;
    int uglyFound = 0;
    while(uglyFound < index)
    {
        ++number;

        if(IsUgly(number))
        {
            ++uglyFound;
        }
    }

    return number;
}
```

我们只需要在函数 GetUglyNumber 中传入参数 1500，就能得到第 1500 个丑数。该算法非常直观，代码也非常简洁，但最大的问题是每个整数都需要计算。即使一个数字不是丑数，我们还是需要对它执行求余数和除法操作。因此该算法的时间效率不是很高，面试官也不会就此满足，他会提示我们还有更高效的算法。

❖ 创建数组保存已经找到的丑数，用空间换时间的解法

前面的算法之所以效率低，很大程度上是因为不管一个数是不是丑数，我们都要对它进行计算。接下来我们试着找到一种只计算丑数的方法，而不在非丑数的整数上花费时间。根据丑数的定义，丑数应该是另一个丑数乘以 2、3 或者 5 的结果（1 除外）。因此，我们可以创建一个数组，里面的数字是排好序的丑数，每个丑数都是前面的丑数乘以 2、3 或者 5 得到的。

这种思路的关键在于怎样确保数组里面的丑数是排好序的。假设数组中已经有若干个排好序的丑数，并且把已有最大的丑数记作 M ，接下来分析如何生成下一个丑数。该丑数肯定是前面某一个丑数乘以 2、3 或者 5 的结果，所以我们首先考虑把已有的每个丑数乘以 2。在乘以 2 的时候，能得到若干个小于或等于 M 的结果。由于是按照顺序生成的，小于或者等于 M 肯定已经在数组中了，我们不需再次考虑；还会得到若干个大于 M 的结果，但我们只需要第一个大于 M 的结果，因为我们希望丑数是按从小到大的顺序生成的，其他更大的结果以后再说。我们把得到的第一个乘以 2 后大于 M 的结果记为 M_2 。同样，我们把已有的每个丑数乘以 3 和 5，能得到第一个大于 M 的结果 M_3 和 M_5 。那么下一个丑数应该是 M_2 、 M_3 和 M_5 这 3 个数的最小者。

在前面分析的时候提到把已有的每个丑数分别乘以 2、3 和 5。事实上这不是必需的，因为已有的丑数是按顺序存放在数组中的。对于乘以 2 而言，肯定存在某一个丑数 T_2 ，排在它之前的每个丑数乘以 2 得到的结果都会小于已有最大的丑数，在它之后的每个丑数乘以 2 得到的结果都会太大。我们只需记下这个丑数的位置，同时每次生成新的丑数的时候去更新这个 T_2 即可。对于乘以 3 和 5 而言，也存在同样的 T_3 和 T_5 。

有了这些分析，我们就可以写出如下代码：

```
int GetUglyNumber_Solution2(int index)
{
    if(index <= 0)
```

```

    return 0;

    int *pUglyNumbers = new int[index];
    pUglyNumbers[0] = 1;
    int nextUglyIndex = 1;

    int *pMultiply2 = pUglyNumbers;
    int *pMultiply3 = pUglyNumbers;
    int *pMultiply5 = pUglyNumbers;

    while(nextUglyIndex < index)
    {
        int min = Min(*pMultiply2 * 2, *pMultiply3 * 3, *pMultiply5 * 5);
        pUglyNumbers[nextUglyIndex] = min;

        while(*pMultiply2 * 2 <= pUglyNumbers[nextUglyIndex])
            ++pMultiply2;
        while(*pMultiply3 * 3 <= pUglyNumbers[nextUglyIndex])
            ++pMultiply3;
        while(*pMultiply5 * 5 <= pUglyNumbers[nextUglyIndex])
            ++pMultiply5;

        ++nextUglyIndex;
    }

    int ugly = pUglyNumbers[nextUglyIndex - 1];
    delete[] pUglyNumbers;
    return ugly;
}

int Min(int number1, int number2, int number3)
{
    int min = (number1 < number2) ? number1 : number2;
    min = (min < number3) ? min : number3;

    return min;
}

```

和第一种思路相比，第二种思路不需要在非丑数的整数上进行任何计算，因此时间效率有明显提升。但也需要指出，第二种算法由于需要保存已经生成的丑数，则因此需要一个数组，从而增加了空间消耗。如果是求第1500个丑数，则将创建一个能容纳1500个丑数的数组，这个数组占据6KB的内容空间。而第一种思路没有这样的内存开销。总的来说，第二种思路相当于用较小的空间消耗换取了时间效率的提升。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/49_UglyNumber



测试用例：

- 功能测试（输入 2、3、4、5、6 等）。
- 特殊输入测试（边界值 1；无效输入 0）。
- 性能测试（输入较大的数字，如 1500）。



本题考点：

- 考查应聘者对时间复杂度的理解。绝大部分应聘者都能想出第一种思路。在面试官提示还有更快的解法之后，应聘者能否分析出时间效率的瓶颈，并找出解决方案，是能否通过这轮面试的关键。
- 考查应聘者的学习能力和沟通能力。丑数对很多人而言是一个新概念。有些面试官喜欢在面试的时候定义一个新概念，然后针对这个新概念出面试题。这就要求应聘者听到不熟悉的概念之后，要有主动积极的态度，大胆向面试官提问，经过几次思考、提问、再思考的循环，在短时间内理解这个新概念。这个过程就体现了应聘者的学习能力和沟通能力。

面试题 50：第一个只出现一次的字符

题目一：字符串中第一个只出现一次的字符。

在字符串中找出第一个只出现一次的字符。如输入 "abaccdeff"，则输出 'b'。

看到这道题时，我们最直观的想法是从头开始扫描这个字符串中的每个字符。当访问到某字符时，拿这个字符和后面的每个字符相比较，如果在后面没有发现重复的字符，则该字符就是只出现一次的字符。如果字符串有 n 个字符，则每个字符可能与后面的 $O(n)$ 个字符相比较，因此这种思路的时间复杂度是 $O(n^2)$ 。面试官不会满意这种思路，他会提示我们还有更快的方法。

由于题目与字符出现的次数相关，那么我们是不是可以统计每个字符

在该字符串中出现的次数？要达到这个目的，我们需要一个数据容器来存放每个字符的出现次数。在这个数据容器中，可以根据字符来查找它出现的次数，也就是说这个容器的作用是把一个字符映射成一个数字。在常用的数据容器中，哈希表正是这个用途。

为了解决这个问题，我们可以定义哈希表的键值（Key）是字符，而值（Value）是该字符出现的次数。同时我们还需要从头开始扫描字符串两次。第一次扫描字符串时，每扫描到一个字符，就在哈希表的对应项中把次数加1。接下来第二次扫描时，每扫描到一个字符，就能从哈希表中得到该字符出现的次数。这样，第一个只出现一次的字符就是符合要求的输出。

哈希表是一种比较复杂的数据结构，C++标准模板库中的 `map` 和 `unordered_map` 实现了哈希表的功能，我们可以直接拿过来用。由于本题的特殊性，我们其实只需要一个非常简单的哈希表就能满足要求，因此我们可以考虑实现一个简单的哈希表。字符（char）是一个长度为8的数据类型，因此总共有256种可能。于是我们创建一个长度为256的数组，每个字母根据其ASCII码值作为数组的下标对应数组的一个数字，而数组中存储的是每个字符出现的次数。这样我们就创建了一个大小为256、以字符ASCII码为键值的哈希表。

第一次扫描时，在哈希表中更新一个字符出现的次数的时间是 $O(1)$ 。如果字符串长度为 n ，那么第一次扫描的时间复杂度是 $O(n)$ 。第二次扫描时，同样在 $O(1)$ 时间内能读出一个字符出现的次数，所以时间复杂度仍然是 $O(n)$ 。这样算起来，总的时间复杂度是 $O(n)$ 。同时，我们需要一个包含256个字符的辅助数组，它的大小是1KB。由于这个数组的大小是一个常数，因此可以认为这种算法的空间复杂度是 $O(1)$ 。

当我们向面试官讲述清楚这种思路并得到面试官的首肯之后，就可以动手写代码了。下面是一段参考代码：

```
char FirstNotRepeatingChar(char* pString)
{
    if(pString == nullptr)
        return '\0';
    const int tableSize = 256;
    unsigned int hashTable[tableSize];
    for(unsigned int i = 0; i < tableSize; ++ i)
        hashTable[i] = 0;

    char* pHashKey = pString;
    while(*(pHashKey) != '\0')
        hashTable[*pHashKey]++;
}
```

```

pHashKey = pString;
while(*pHashKey != '\0')
{
    if(hashTable[*pHashKey] == 1)
        return *pHashKey;

    pHashKey++;
}

return '\0';
}

```



源代码:

本题完整的源代码:

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/50_01_FirstNotRepeatingChar



测试用例:

- 功能测试（字符串中存在只出现一次的字符；字符串中不存在只出现一次的字符；字符串中所有字符都只出现一次）。
- 特殊输入测试（字符串为 nullptr 指针）。



本题考点:

- 考查应聘者对数组和字符串的编程能力。
- 考查应聘者对哈希表的理解及运用。
- 考查应聘者对时间效率及空间效率的分析能力。当面试官提示最直观的算法不是最优解的时候，应聘者需要立即分析出这种算法的时间效率。在想出基于哈希表的算法之后，应聘者也应该分析出该方法的时间效率和空间效率分别是 $O(n)$ 和 $O(1)$ 。



本题扩展:

在前面的例子中，我们之所以可以把哈希表的大小设为 256，是因为字符（char）是 8bit 的类型，总共只有 256 个字符。但实际上字符不只是 256

个，比如中文就有几千个汉字。如果题目要求考虑汉字，那么前面的算法是不是有问题？如果有，则可以怎么解决？



相关题目：

- 定义一个函数，输入两个字符串，从第一个字符串中删除在第二个字符串中出现过的所有字符。例如，从第一个字符串"We are students."中删除在第二个字符串"aeiou"中出现过的字符得到的结果是"W r Stdnts."。为了解决这个问题，我们可以创建一个用数组实现的简单哈希表来存储第二个字符串。这样我们从头到尾扫描第一个字符串的每个字符时，用 $O(1)$ 时间就能判断出该字符是不是在第二个字符串中。如果第一个字符串的长度是 n ，那么总的时间复杂度是 $O(n)$ 。
- 定义一个函数，删除字符串中所有重复出现的字符。例如，输入"google"，删除重复的字符之后的结果是"gole"。这道题目和上面的问题比较类似，我们可以创建一个用布尔型数组实现的简单的哈希表。数组中的元素的意义是其下标看作 ASCII 码后对应的字母在字符串中是否已经出现。我们先把数组中所有的元素都设为 false。以"google"为例，当扫描到第一个 g 时，g 的 ASCII 码是 103，那么我们把数组中下标为 103 的元素设为 true。当扫描到第二个 g 时，我们发现数组中下标为 103 的元素的值是 true，就知道 g 在前面已经出现过。也就是说，我们用 $O(1)$ 时间就能判断出每个字符是否在前面已经出现过。如果字符串的长度是 n ，那么总的时间复杂度是 $O(n)$ 。
- 在英语中，如果两个单词中出现的字母相同，并且每个字母出现的次数也相同，那么这两个单词互为变位词 (Anagram)。例如，silent 与 listen、evil 与 live 等互为变位词。请完成一个函数，判断输入的两个字符串是不是互为变位词。我们可以创建一个用数组实现的简单哈希表，用来统计字符串中每个字符出现的次数。当扫描到第一个字符串中的每个字符时，为哈希表对应的项的值增加 1。接下来扫描第二个字符串，当扫描到每个字符时，为哈希表对应的项的值减去 1。如果扫描完第二个字符串后，哈希表中所有的值都是 0，那么这两个字符串就互为变位词。



举一反三：

如果需要判断多个字符是不是在某个字符串里出现过或者统计多个字符在某个字符串中出现的次数，那么我们可以考虑基于数组创建一个简单的哈希表，这样可以用很小的空间消耗换来时间效率的提升。

题目二：字符流中第一个只出现一次的字符。

请实现一个函数，用来找出字符流中第一个只出现一次的字符。例如，当从字符流中只读出前两个字符"go"时，第一个只出现一次的字符是'g'；当从该字符流中读出前 6 个字符"google"时，第一个只出现一次的字符是'l'。

字符只能一个接着一个从字符流中读出来。可以定义一个数据容器来保存字符在字符流中的位置。当一个字符第一次从字符流中读出来时，把它在字符流中的位置保存到数据容器里。当这个字符再次从字符流中读出来时，那么它就不是只出现一次的字符，也就可以被忽略了。这时把它在数据容器里保存的值更新成一个特殊的值（如负数值）。

为了尽可能高效地解决这个问题，需要在 $O(1)$ 时间内往数据容器里插入一个字符，以及更新一个字符对应的值。受面试题 50 的启发，这个数据容器可以用哈希表来实现。用字符的 ASCII 码作为哈希表的键值，而把字符对应的位置作为哈希表的值。实现这种思路的参考代码如下：

```
class CharStatistics
{
public:
    CharStatistics() : index (0)
    {
        for(int i = 0; i < 256; ++i)
            occurrence[i] = -1;
    }

    void Insert(char ch)
    {
        if(occurrence[ch] == -1)
            occurrence[ch] = index;
        else if(occurrence[ch] >= 0)
            occurrence[ch] = -2;

        index++;
    }

    char FirstAppearingOnce()
    {
        char ch = '\0';
        ...
    }
}
```

```

int minIndex = numeric_limits<int>::max();
for(int i = 0; i < 256; ++i)
{
    if(occurrence[i] >= 0 && occurrence[i] < minIndex)
    {
        ch = (char)i;
        minIndex = occurrence[i];
    }
}
return ch;
}

private:
// occurrence[i]: A character with ASCII value i;
// occurrence[i] = -1: The character has not found;
// occurrence[i] = -2: The character has been found for mutlple times
// occurrence[i] >= 0: The character has been found only once
int occurrence[256];
int index;
};

```

在上述代码中，哈希表用数组 `occurrence` 实现。数组中的元素 `occurrence[i]` 和 ASCII 码的值为 i 的字符相对应。最开始的时候，数组中的所有元素都初始化为-1。当一个 ASCII 码为 i 的字符第一次从字符流中读出时，`occurrence[i]` 的值更新为它在字符流中的位置。当这个字符再次从字符流中读出时（`occurrence[i]` 大于或者等于 0），`occurrence[i]` 的值更新为-2。

当我们需要找到目前为止从字符流里读出的所有字符中第一个不重复的字符时，只需要扫描整个数组，并从中找出最小的大于等于 0 的值对应的字符即可。这就是函数 `FirstAppearingOnce` 的功能。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/50_02_FirstCharacterInStream



测试用例：

- 功能测试（读入一个字符；读入多个字符；读入的所有字符都是唯一的；读入的所有字符都是重复出现的）。
- 特殊输入测试（读入 0 个字符）。



本题考点：

- 考查应聘者对数组和字符串的编程能力。
- 考查应聘者对哈希表的理解及运用。
- 考查应聘者对时间效率及空间效率的分析能力。当面试官提示最直观的算法不是最优解的时候，应聘者需要立即分析出这种算法的时间效率。在想出基于哈希表的算法之后，应聘者也应该分析出该方法的时间效率和空间效率分别是 $O(n)$ 和 $O(1)$ 。

面试题 51：数组中的逆序对

题目：在数组中的两个数字，如果前面一个数字大于后面的数字，则这两个数字组成一个逆序对。输入一个数组，求出这个数组中的逆序对的总数。例如，在数组{7, 5, 6, 4}中，一共存在 5 个逆序对，分别是(7, 6)、(7, 5)、(7, 4)、(6, 4)和(5, 4)。

看到这道题目，我们的第一反应是顺序扫描整个数组。每扫描到一个数字，逐个比较该数字和它后面的数字的大小。如果后面的数字比它小，则这两个数字就组成一个逆序对。假设数组中含有 n 个数字。由于每个数字都要和 $O(n)$ 个数字进行比较，因此这种算法的时间复杂度是 $O(n^2)$ 。我们再尝试找找更快的算法。

我们以数组{7, 5, 6, 4}为例来分析统计逆序对的过程。每扫描到一个数字的时候，我们不能拿它和后面的每一个数字进行比较，否则时间复杂度就是 $O(n^2)$ ，因此，我们可以考虑先比较两个相邻的数字。

如图 5.2 (a) 和图 5.2 (b) 所示，我们先把数组分解成两个长度为 2 的子数组，再把这两个子数组分别拆分成两个长度为 1 的子数组。接下来一边合并相邻的子数组，一边统计逆序对的数目。在第一对长度为 1 的子数组{7}、{5}中，7 大于 5，因此(7, 5)组成一个逆序对。同样，在第二对长度为 1 的子数组{6}、{4}中，也有逆序对(6, 4)。由于我们已经统计了这两对子数组内部的逆序对，因此需要把这两对子数组排序，如图 5.2 (c) 所示，以免在以后的统计过程中再重复统计。

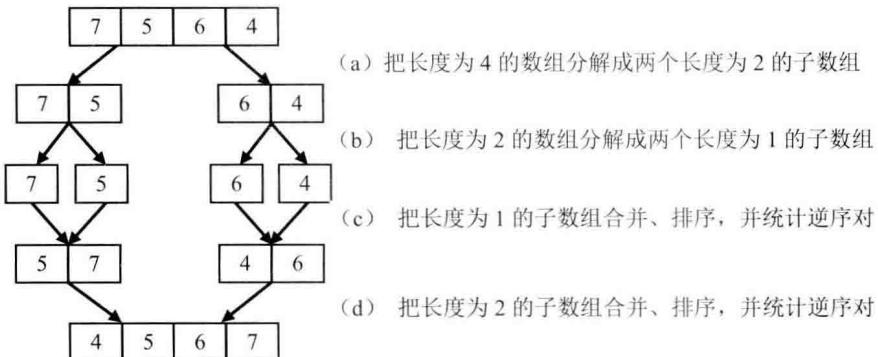


图 5.2 统计数组{7, 5, 6, 4} 中逆序对的过程

接下来我们统计两个长度为 2 的子数组之间的逆序对。我们在图 5.3 中细分图 5.2 (d) 的合并子数组及统计逆序对的过程。

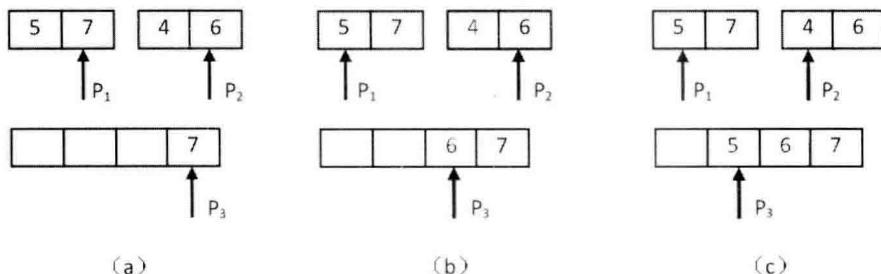


图 5.3 图 5.2 (d) 中合并两个子数组并统计逆序对的过程

注：图中省略了最后一步，即复制第二个子数组最后剩余的 4 到辅助数组。
(a) P_1 指向的数字大于 P_2 指向的数字，表明数组中存在逆序对。 P_2 指向的数字是第二个子数组的第二个数字，因此第二个子数组中有两个数字比 7 小。把逆序对数目加 2，并把 7 复制到辅助数组，向前移动 P_1 和 P_3 。
(b) P_1 指向的数字小于 P_2 指向的数字，没有逆序对。把 P_2 指向的数字复制到辅助数组，并向前移动 P_2 和 P_3 。
(c) P_1 指向的数字大于 P_2 指向的数字，因此存在逆序对。由于 P_2 指向的数字是第二个子数组的第一个数字，子数组中只有一个数字比 5 小。把逆序对数目加 1，并把 5 复制到辅助数组，向前移动 P_1 和 P_3 。

我们先用两个指针分别指向两个子数组的末尾，并每次比较两个指针指向的数字。如果第一个子数组中的数字大于第二个子数组中的数字，则构成逆序对，并且逆序对的数目等于第二个子数组中剩余数字的个数，如图 5.3 (a) 和图 5.3 (c) 所示。如果第一个数组中的数字小于或等于第二个

数组中的数字，则不构成逆序对，如图 5.3 (b) 所示。每次比较的时候，我们都把较大的数字从后往前复制到一个辅助数组，确保辅助数组中的数字是递增排序的。在把较大的数字复制到辅助数组之后，把对应的指针向前移动一位，接下来进行下一轮比较。

经过前面详细的讨论，我们可以总结出统计逆序对的过程：先把数组分隔成子数组，统计出子数组内部的逆序对的数目，然后再统计出两个相邻子数组之间的逆序对的数目。在统计逆序对的过程中，还需要对数组进行排序。如果对排序算法很熟悉，那么我们不难发现这个排序的过程实际上就是归并排序。我们可以基于归并排序写出如下代码：

```
int InversePairs(int* data, int length)
{
    if(data == nullptr || length < 0)
        return 0;

    int* copy = new int[length];
    for(int i = 0; i < length; ++ i)
        copy[i] = data[i];

    int count = InversePairsCore(data, copy, 0, length - 1);
    delete[] copy;

    return count;
}

int InversePairsCore(int* data, int* copy, int start, int end)
{
    if(start == end)
    {
        copy[start] = data[start];
        return 0;
    }

    int length = (end - start) / 2;

    int left = InversePairsCore(copy, data, start, start + length);
    int right = InversePairsCore(copy, data, start + length + 1, end);

    // i 初始化为前半段最后一个数字的下标
    int i = start + length;
    // j 初始化为后半段最后一个数字的下标
    int j = end;
    int indexCopy = end;
    int count = 0;
    while(i >= start && j >= start + length + 1)
    {
        if(data[i] > data[j])
        {
            copy[indexCopy] = data[j];
            j--;
            count++;
        }
        else
        {
            copy[indexCopy] = data[i];
            i--;
        }
        indexCopy--;
    }

    for(int k = start; k < end; k++)
        data[k] = copy[k];
}
```

```

        copy[indexCopy--] = data[i--];
        count += j - start - length;
    }
} else
{
    copy[indexCopy--] = data[j--];
}
}

for( i >= start; --i)
    copy[indexCopy--] = data[i];

for( j >= start + length + 1; --j)
    copy[indexCopy--] = data[j];

return left + right + count;
}

```

我们知道，归并排序的时间复杂度是 $O(n \log n)$ ，比最直观的 $O(n^2)$ 要快，但同时归并排序需要一个长度为 n 的辅助数组，相当于我们用 $O(n)$ 的空间消耗换来了时间效率的提升，因此这是一种用空间换时间的算法。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/51_InversePairs



测试用例：

- 功能测试（输入未经排序的数组、递增排序的数组、递减排序的数组；输入的数组中包含重复的数字）。
- 边界值测试（输入的数组中只有两个数字；输入的数组中只有一个数字）。
- 特殊输入测试（表示数组的指针为 nullptr 指针）。



本题考点：

- 考查应聘者分析复杂问题的能力。统计逆序对的过程很复杂，如何发现逆序对的规律，是应聘者解决这道题目关键。

- 考查应聘者对归并排序的掌握程度。如果应聘者在分析统计逆序对的过程中发现问题与归并排序的相似性，并能基于归并排序形成解题思路，那通过这轮面试的概率就很大了。

面试题 52：两个链表的第一个公共节点

题目：输入两个链表，找出它们的第一个公共节点。链表节点定义如下：

```
struct ListNode
{
    int      m_nKey;
    ListNode* m_pNext;
};
```

面试的时候碰到这道题，很多应聘者的第一反应就是蛮力法：在第一个链表上顺序遍历每个节点，每遍历到一个节点，就在第二个链表上顺序遍历每个节点。如果在第二个链表上有一个节点和第一个链表上的节点一样，则说明两个链表在这个节点上重合，于是就找到了它们的公共节点。如果第一个链表的长度为 m ，第二个链表的长度为 n ，那么，显然该方法的时间复杂度是 $O(mn)$ 。

通常蛮力法不会是最好的办法，我们接下来试着分析有公共节点的两个链表有哪些特点。从链表节点的定义可以看出，这两个链表是单向链表。如果两个单向链表有公共的节点，那么这两个链表从某一节点开始，它们的 m_pNext 都指向同一个节点。但由于是单向链表的节点，每个节点只有一个 m_pNext ，因此从第一个公共节点开始，之后它们所有的节点都是重合的，不可能再出现分叉。所以两个有公共节点而部分重合的链表，其拓扑形状看起来像一个 Y，而不可能像 X，如图 5.4 所示。

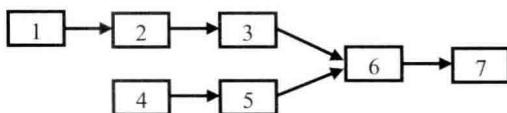


图 5.4 两个链表在值为 6 的节点处交汇

经过分析我们发现，如果两个链表有公共节点，那么公共节点出现在两个链表的尾部。如果我们从两个链表的尾部开始往前比较，那么最后一个相同的节点就是我们要找的节点。可问题是，在单向链表中，我们只能从头节点开始按顺序遍历，最后才能到达尾节点。最后到达的尾节点却要最先被比较，这听起来是不是像“后进先出”？于是我们就能想到用栈的特点来

解决这个问题：分别把两个链表的节点放入两个栈里，这样两个链表的尾节点就位于两个栈的栈顶，接下来比较两个栈顶的节点是否相同。如果相同，则把栈顶弹出接着比较下一个栈顶，直到找到最后一个相同的节点。

在上述思路中，我们需要用两个辅助栈。如果链表的长度分别为 m 和 n ，那么空间复杂度是 $O(m+n)$ 。这种思路的时间复杂度也是 $O(m+n)$ 。和最开始的蛮力法相比，时间效率得到了提高，相当于用空间消耗换取了时间效率。

之所以需要用到栈，是因为我们想同时遍历到达两个栈的尾节点。当两个链表的长度不相同时，如果我们从头开始遍历，那么到达尾节点的时间就不一致。其实解决这个问题还有一种更简单的办法：首先遍历两个链表得到它们的长度，就能知道哪个链表比较长，以及长的链表比短的链表多几个节点。在第二次遍历的时候，在较长的链表上先走若干步，接着同时在两个链表上遍历，找到的第一个相同的节点就是它们的第一个公共节点。

比如在图 5.4 的两个链表中，我们可以先遍历一次得到它们的长度分别为 5 和 4，也就是较长的链表与较短的链表相比多一个节点。第二次先在长的链表上走 1 步，到达节点 2。接下来分别从节点 2 和节点 4 出发同时遍历两个节点，直到找到它们第一个相同的节点 6，这就是我们想要的结果。

第三种思路和第二种思路相比，时间复杂度都是 $O(m+n)$ ，但我们不再需要辅助栈，因此提高了空间效率。当面试官首肯了我们的最后一种思路之后，就可以动手写代码了。下面是一段参考代码：

```
ListNode* FindFirstCommonNode( ListNode *pHead1, ListNode *pHead2)
{
    // 得到两个链表的长度
    unsigned int nLength1 = GetListLength(pHead1);
    unsigned int nLength2 = GetListLength(pHead2);
    int nLengthDif = nLength1 - nLength2;

    ListNode* pListHeadLong = pHead1;
    ListNode* pListHeadShort = pHead2;
    if(nLength2 > nLength1)
    {
        pListHeadLong = pHead2;
        pListHeadShort = pHead1;
        nLengthDif = nLength2 - nLength1;
    }

    // 先在长链表上走几步，再同时在两个链表上遍历
    for(int i = 0; i < nLengthDif; ++ i)
        pListHeadLong = pListHeadLong->m_pNext;

    while((pListHeadLong != nullptr) &&
```

```

(pListHeadShort != nullptr) &&
(pListHeadLong != pListHeadShort))
{
    pListHeadLong = pListHeadLong->m_pNext;
    pListHeadShort = pListHeadShort->m_pNext;
}

// 得到第一个公共节点
ListNode* pFirstCommonNode = pListHeadLong;

return pFirstCommonNode;
}

unsigned int GetListLength(ListNode* pHead)
{
    unsigned int nLength = 0;
    ListNode* pNode = pHead;
    while(pNode != nullptr)
    {
        ++nLength;
        pNode = pNode->m_pNext;
    }

    return nLength;
}

```



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/52_FirstCommonNodesInLists



测试用例：

- 功能测试（输入的两个链表有公共节点：第一个公共节点在链表的中间，第一个公共节点在链表的末尾，第一个公共节点是链表的头节点；输入的两个链表没有公共节点）。
- 特殊输入测试（输入的链表头节点是 `nullptr` 指针）。



本题考点：

- 考查应聘者对时间复杂度和空间复杂度的理解及分析能力。解决这道题有多种不同的思路。每当应聘者想到一种思路的时候，都要很

快分析出这种思路的时间复杂度和空间复杂度各是多少，并找到可以优化的地方。

- 考查应聘者对链表的编程能力。



相关题目：

如果把图 5.4 逆时针旋转 90° ，我们就会发现两个链表的拓扑形状和一棵树的形状非常相似，只是这里的指针是从叶节点指向根节点的。两个链表的第一个公共节点正好就是二叉树中两个叶节点的最低公共祖先。在本书 7.2 节，我们将详细讨论如何求两个节点的最低公共祖先。

5.4 本章小结

在编程面试的时候，面试官通常对时间复杂度和空间复杂度都会有要求，并且一般情况下面试官更加关注时间复杂度。

降低时间复杂度的第一种方法是改用更加高效的算法。比如我们用动态规划解答面试题 42 “连续子数组的最大和”能够把时间复杂度降低到 $O(n)$ ，利用快速排序的 Partition 函数也能在 $O(n)$ 时间内解决面试题 39 “数组中出现次数超过一半的数字”和面试题 40 “最小的 k 个数”。

降低时间复杂度的第二种方法是用空间换取时间。在解决面试题 50 “第一个只出现一次的字符”的时候，我们用数组实现一个简单的哈希表，于是用 $O(1)$ 时间就能知道任意字符出现的次数。这种思路可以解决很多同类型的题目。另外，我们可以创建一个缓存保存中间的计算结果，从而避免重复的计算。面试题 49 “丑数”就是这方面的一个例子。在用递归的思路求解问题的时候，如果有重复的子问题，那么我们也可以通过保存求解子问题的结果来避免重复计算。更多关于递归的讨论请参考本书的 2.4.1 节及面试题 10 “斐波那契数列”。

值得注意的是，以空间换取时间并不一定都是可行的方案。我们要注意需要的辅助空间的大小，消耗太多的内存可能得不偿失。另外，我们还要关注问题的背景。如果面试题是有关嵌入式开发的，那么对空间消耗就要格外留心，因为通常嵌入式系统的内存很有限。

第6章

面试中的各项能力

6.1

面试官谈能力

“应聘者能够礼貌平和、不卑不亢地和面试官交流，逻辑清晰、详略得当当地介绍自己及项目经历，谈论题目时能够发现问题的细节并向面试官进行询问，这些都是比较好的沟通表现。对自己做的项目能够了解得很深入、对面试题能够快速寻找解决方法是判断应聘者学习能力的一种方法。这两个能力都很重要，基本能够起到一票否决的作用。”

——殷焰（支付宝，高级安全测试工程师）

“有时候会问一些应聘者不是很熟悉的领域，看应聘者在遇到难题时的反应，在他们回答不出时会有人员提供解答，在解答过程中观察他的沟通能力及求知欲。”

——朱麟（交通银行，项目经理）

“沟通能力其实在整个面试过程中都在考核，包括询问他过往的经历，也通常会涉及沟通能力。学习能力是在考查算法或者项目经验的过程中，

通过提问，尤其是一些他没有接触过的问题来考核的。沟通能力和学习能力很重要，在某种程度上这些都是潜力。如果应聘者沟通能力不行、难以合作，那么我们不会录取。”

——何幸杰（SAP，高级工程师）

“让其介绍过往项目其实就在考查沟通和表达能力。学习能力通过问其看书和关注什么来考查。沟通能力、学习能力对最终面试结果会有一定的影响。对于资深的应聘者，影响要大些。”

——韩伟东（盛大，高级研究员）

“应聘者会被问及一些需求不是很明确的问题，解决这些问题需要应聘者和面试官进行沟通，以及在讲解设计思路和代码的过程中也需要和面试官交流互动。沟通及学习能力是面试成绩中关键的考查点。”

——尧敏（淘宝，资深经理）

“沟通、学习能力就是看面试者能否清晰、有条理地表达自己，是否会在自己所得到的信息不足的情况下主动发问澄清，能否在得到一些暗示之后迅速做出反应纠正错误。”

——陈黎明（微软，SDE II）

6.2 沟通能力和学习能力

1. 沟通能力

随着软件、系统功能越来越复杂，开发团队的规模也随之扩张，开发者、测试者和项目经理之间的沟通交流也变得越来越重要。也正因如此，很多公司在面试的时候都会注意考查应聘者的沟通能力。这就要求应聘者

无论是在介绍项目经验还是在介绍解题思路的时候，都需要逻辑清晰明了，语言详略得当，表述的时候重点突出、观点明确。

我们不能把好的沟通能力理解成夸夸其谈。在面试的时候，知之为知之，不知为不知，对于不清楚的知识点，要勇敢承认，千万别不懂装懂。通常当应聘者说自己很懂某一领域的时候，面试官都会跟进几个问题。如果应聘者在不懂装懂，那么面试官迟早会发现，他可能就会觉得应聘者在其他的地方也有浮报虚夸的成分，这将是得不偿失的。

有意向加入外企的应聘者要注意提高自己英文交流的能力。不少外企的面试部分甚至全部采用英语面试，这对英语的要求就很高。我们通过了英语的四六级考试未必能用英语对话。如果觉得自己英语的听说能力还不够好，则建议花更多的时间来提高自己的听力。英语面试中最重要的是我们要听懂面试官的问题。通常采用英语面试的面试官自己的英语都比较好，即使我们的发音不够标准，对方一般也能听懂。这和我们能听懂普通话不标准的外国人说中文的道理是一样的。但如果我们没有听明白面试官的问题，那么说得再清楚也无济于事了。

2. 学习能力

计算机是一门更新速度很快的学科，每年都有新的技术不断涌现。因此，作为这个领域从业人员的软件工程师需要具备很强的学习能力，否则时间一长就会跟不上技术进步的步伐。也正是因为这个原因，IT公司在面试的时候，面试官都会重视考查应聘者的学习能力。只有具备很强的学习能力及学习愿望的人，才能不断完善自己的知识结构，不断学习新的先进技术，让自己的职业生涯保持长久的生命力。

通常面试官有两种方法考查应聘者的学习能力。第一种方法是询问应聘者最近在看什么书或者在做什么项目、从中学到了哪些新技术。面试官可以用这个问题了解应聘者的学习愿望和学习能力。学习能力强的人对各种新技术充满了兴趣，随时学习、吸收新知识，并把知识转换为自己的技能。第二种方法是抛出一个新概念，接下来观察应聘者能不能在较短时间内理解这个新概念并解决相关的问题。本书收集的面试题涉及诸如数组的旋转（面试题 11）、二叉树的镜像（面试题 27）、丑数（面试题 49）、逆序对（面试题 51）等新概念。当面试官提出这些新概念的时候，他期待应聘者能够通过思考、提问、再思考的过程，理解它们并最终解决问题。

3. 善于学习、沟通的人也善于提问

面试官有一个很重要的任务，就是考查应聘者的学习愿望及学习能力。学习能力怎么体现呢？面试官提出一个新概念，应聘者没有听说过它，于是他在已有的理解的基础上提出进一步的问题，在得到面试官的答复之后，思考再提问，几个来回之后掌握了这个概念。这个过程能够体现应聘者的学习能力。通常学习能力强的人具有主动积极的态度，对未知的领域有强烈的求知欲望。因此，建议应聘者在面试过程中遇到不明白的地方多提问，这样面试官就会觉得你态度积极、求知欲望强烈，会给面试结果加分。



面试小提示：

面试是一个双向交流的过程，面试官可以问应聘者问题，同样应聘者也可以向面试官提问。如果应聘者能够针对面试题主动地提出几个高质量的问题，那么面试官就会觉得他有很强的沟通能力和学习能力。

举个例子，Google 曾经有一道面试题：找出第 1500 个丑数。很多人都不知道丑数是什么。不知道怎么办？面试官就坐在对面，可以问他。面试官会告诉你只含有 2、3、5 三个因子的数就是丑数。你听了后，觉得听明白了，但不太确定，于是可以举几个例子并让面试官确认你的理解是不是正确：6、8、10、12 都是丑数，但 14 就不是，对吗？当面试官给出肯定的答复后，你就知道自己的理解是对的。问题问的是第 1500 个丑数，与顺序有关。可是哪个数字是第一个丑数呢，1 是不是第一个？这个你可能也不能确定，怎么办？还是问面试官，他会告诉你 1 是或者不是丑数。题目是他出的，他有责任把题目解释清楚。

有些面试官故意一开始不把题目描述清楚，让题目存在一定的二义性。他期待应聘者能够一步步通过提问来弄明白题目的要求。这也是在考查应聘者的沟通能力。为什么要这样考查？因为实际工作也是这样，不是一开始项目需求就定义得很清楚，程序员需要与项目经理甚至客户反复沟通才能把需求弄清楚。如果没有一定的沟通能力，那么当程序员面对一个模糊的客户需求时，他就会觉得无从下手。

比如最近很流行的一道面试题，面试官最开始问：如何求树中两个节点的最低公共祖先。此时面试官对题目中的树的特点完全没有给出描述，他希望应聘者在听到问题后会提出几个问题，比如这棵树是二叉树还是普通的树。

如果面试官说是二叉树，则应聘者可以继续问该树是不是排序的二叉树。面试官回答是排序的。听到这里，应聘者才能确定思路：从树的根节点出发遍历树，如果当前节点都大于输入的两个节点，则下一步遍历当前节点的左子树；如果当前节点都小于输入的两个节点，则下一步遍历当前节点的右子树。一直遍历到当前节点比一个输入节点大而比另一个小，此时当前节点就是符合要求的最低公共祖先。

在应聘者问树是不是二叉树的时候，如果面试官回答是任意的树，那么应聘者可以接着提问在树的节点中有没有指向父节点的指针。如果面试官给出肯定的回答，也就是树的节点中有指向父节点的指针，那么从输入的节点出发，沿着指向父节点的指针一直到树的根节点，可以看作一个链表，因此这道题目的解法就和求两个链表的第一个公共节点的解法是一样的。如果面试官给出的是否定的回答，也就是树的节点没有指向父节点的指针，那么我们可以在遍历的时候用一个栈来保存从根节点到当前节点的路径，最终把它转换为求两个路径的最后一个公共节点。详细的解题过程请参考本书的 7.2 节。

面试官给出不同的条件，这将是 3 道完全不一样的题目。如果一开始应聘者没有弄清楚面试官的意图就贸然动手解题，那结果很有可能是离题千里。从中我们也可以看出在面试过程中沟通的重要性。当觉得题目的条件、要求不够明确的时候，我们一定要多提问以消除自己的疑惑。

6.3

知识迁移能力

所谓学习能力，很重要的一点就是根据已经掌握的知识、技术，能够迅速学习、理解新的技术并运用到实际工作中去。大部分新的技术都不是凭空产生的，而是在已有技术的基础上发展起来的。这就要求我们能够把对已有技术的理解迁移到学习新技术的过程中去，也就是要具备很强的知识迁移能力。以学习编程语言为例，如果全面理解了 C++ 的面向对象的思想，那么学习下一门面向对象的语言 Java 就不会很难；在深刻理解了 Java 的垃圾回收机制之后，再去学习另一门托管语言比如 C#，也会很容易。

面试官考查应聘者知识迁移能力的一种方法是把经典的问题稍作变换。这时候面试官期待应聘者能够找到和经典问题的联系，并从中受到启

发，把解决经典问题的思路迁移过来解决新的问题。比如，如果遇到面试题53“在排序数组中查找数字”，我们看到“排序数组”就可以想到二分查找算法。通常二分查找算法用来在一个排序数组中查找一个数字。我们可以把二分查找算法的思想迁移过来稍作变换，用二分查找算法在排序数组中查找重复数字的第一个和最后一个，从而得到数字在数组中出现的次数。

面试官考查应聘者知识迁移能力的另一种方法就是先问一个简单的问题，在应聘者解答完这个简单的问题之后，再追问一个相关的同时难度也更大的问题。这时候面试官希望应聘者能够总结前面解决简单问题的经验，把前面的思路、方法迁移过来。比如在面试题56“数组中数字出现的次数”中，面试官先问一个简单的问题，即数组中只有一个数字只出现一次的情况。在应聘者想出用异或的办法找到这个只出现一次的数字之后，他再追问如果数组中有两个数字只出现一次，那么该怎么找出这两个数字？这时候应聘者要从前面的思路中得到启发：既然有办法找到数组中只出现一次的一个数字，那么当数组中有两个数字只出现一次的时候，我们就可以把整个数组一分为二，每个子数组中包含一个只出现一次的数字，这样我们就能在两个子数组中分别找到那两个只出现一次的数字。接下来我们就可以集中精力去想办法把数组一分为二，这样就能找到解决问题的窍门，整道题目的难度系数就降低了不少。

知识迁移能力的一种通俗的说法是“举一反三”的能力。我们在去面试之前，通常都会看一些经典的面试题。然而题目总是做不完的，我们不可能把所有的面试题都准备一遍。因此，更重要的是每做一道面试题的时候，都要总结这道题的解法有什么特点，有哪些思路是可以应用到同类型的题目中去的。比如，为了解决面试题“翻转单词顺序”，我们先翻转整个句子的所有字符，再分别翻转每个单词中的字符。这样多次翻转字符的思路也可以运用到面试题“左旋转字符串”中（详见面试题58）。在解决面试题38“字符串的排列”之后，我们发现“八皇后问题”其实归根结底就是数组的排列问题。本书中很多章节在分析了一道题之后，列举了与这道题相关的题目，读者可以通过分析这些题目的相关性来提高举一反三的能力。

面试题 53：在排序数组中查找数字

题目一：数字在排序数组中出现的次数。

统计一个数字在排序数组中出现的次数。例如，输入排序数组{1, 2, 3, 3, 3, 3, 4, 5}和数字3，由于3在这个数组中出现了4次，因此输出4。

既然输入的数组是排序的，那么我们就能很自然地想到用二分查找算法。在题目给出的例子中，我们可以先用二分查找算法找到一个3。由于3可能出现多次，因此我们找到的3的左右两边可能都有3，于是在找到的3的左右两边顺序扫描，分别找出第一个3和最后一个3。因为要查找的数字在长度为n的数组中有可能出现 $O(n)$ 次，所以顺序扫描的时间复杂度是 $O(n)$ 。因此，这种算法的效率和直接从头到尾顺序扫描整个数组统计3出现的次数的方法是一样的。显然，面试官不会满意这个算法，他会提示我们还有更快的算法。

接下来我们思考如何更好地利用二分查找算法。假设我们要统计数字k在排序数组中出现的次数。在前面的算法中，时间主要消耗在如何确定重复出现的数字的第一个k和最后一个k的位置上，有没有可能用二分查找算法直接找到第一个k及最后一个k呢？

我们先分析如何用二分查找算法在数组中找到第一个k。二分查找算法总是先拿数组中间的数字和k作比较。如果中间的数字比k大，那么k只可能出现在数组的前半段，下一轮我们只在数组的前半段查找就可以了。如果中间的数字比k小，那么k只可能出现在数组的后半段，下一轮我们只在数组的后半段查找就可以了。如果中间的数字和k相等呢？我们先判断这个数字是不是第一个k。如果中间数字的前面一个数字不是k，那么此时中间的数字刚好就是第一个k；如果中间数字的前面一个数字也是k，那么第一个k肯定在数组的前半段，下一轮我们仍然需要在数组的前半段查找。

基于这种思路，我们可以很容易地写出递归的代码找到排序数组中的第一个k。下面是一段参考代码：

```
int GetFirstK(int* data, int length, int k, int start, int end)
{
    if(start > end)
        return -1;

    int middleIndex = (start + end) / 2;
    int middleData = data[middleIndex];
```

```

if(middleData == k)
{
    if((middleIndex > 0 && data[middleIndex - 1] != k)
       || middleIndex == 0)
        return middleIndex;
    else
        end = middleIndex - 1;
}
else if(middleData > k)
    end = middleIndex - 1;
else
    start = middleIndex + 1;

return GetFirstK(data, length, k, start, end);
}

```

在函数 GetFirstK 中，如果数组中不包含数字 k ，那么返回-1；如果数组中包含至少一个 k ，那么返回第一个 k 在数组中的下标。

我们可以用同样的思路在排序数组中找到最后一个 k 。如果中间数字比 k 大，那么 k 只能出现在数组的前半段。如果中间数字比 k 小，那么 k 只能出现在数组的后半段。如果中间数字等于 k 呢？我们需要判断这个 k 是不是最后一个 k ，也就是中间数字的下一个数字是不是也等于 k 。如果下一个数字不是 k ，则中间数字就是最后一个 k ；否则下一轮我们还是要在数组的后半段中去查找。我们同样可以基于递归写出如下代码：

```

int GetLastK(int* data, int length, int k, int start, int end)
{
    if(start > end)
        return -1;

    int middleIndex = (start + end) / 2;
    int middleData = data[middleIndex];

    if(middleData == k)
    {
        if((middleIndex < length - 1 && data[middleIndex + 1] != k)
           || middleIndex == length - 1)
            return middleIndex;
        else
            start = middleIndex + 1;
    }
    else if(middleData < k)
        start = middleIndex + 1;
    else
        end = middleIndex - 1;

    return GetLastK(data, length, k, start, end);
}

```

和函数 GetFirstK 类似，如果数组中不包含数字 k ，那么 GetLastK 返回 -1；否则返回最后一个 k 在数组中的下标。

在分别找到第一个 k 和最后一个 k 的下标之后，我们就能计算出 k 在数组中出现的次数。相应的代码如下：

```
int GetNumberOfK(int* data, int length, int k)
{
    int number = 0;

    if(data != nullptr && length > 0)
    {
        int first = GetFirstK(data, length, k, 0, length - 1);
        int last = GetLastK(data, length, k, 0, length - 1);

        if(first > -1 && last > -1)
            number = last - first + 1;
    }

    return number;
}
```

在上述代码中，GetFirstK 和 GetLastK 都是用二分查找算法在数组中查找一个合乎要求的数字的，它们的时间复杂度都是 $O(\log n)$ ，因此 GetNumberOfK 的总的时间复杂度也只有 $O(\log n)$ 。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/53_01_NumberOfK



测试用例：

- 功能测试（数组中包含要查找的数字；数组中没有要查找的数字；要查找的数字在数组中出现一次/多次）。
- 边界值测试（查找数组中的最大值、最小值；数组中只有一个数字）。
- 特殊输入测试（表示数组的指针为 `nullptr` 指针）。

题目二：0~n-1 中缺失的数字。

一个长度为 $n-1$ 的递增排序数组中的所有数字都是唯一的，并且每个数字都在范围 $0 \sim n-1$ 之内。在范围 $0 \sim n-1$ 内的 n 个数字中有且只有一个数字不在该数组中，请找出这个数字。

这个问题有一个直观的解决方案。我们可以先用公式 $n(n-1)/2$ 求出数字 $0 \sim n-1$ 的所有数字之和，记为 s_1 。接着求出数组中所有数字的和，记为 s_2 。那个不在数组中的数字就是 s_1-s_2 的差。这种解法需要 $O(n)$ 的时间求数组中所有数字的和。显然，该解法没有有效利用数组是递增排序的这一特点。

因为 $0 \sim n-1$ 这些数字在数组中是排序的，因此数组中开始的一些数字与它们的下标相同。也就是说，0 在下标为 0 的位置，1 在下标为 1 的位置，以此类推。如果不在数组中的那个数字记为 m ，那么所有比 m 小的数字的下标都与它们的值相同。

由于 m 不在数组中，那么 $m+1$ 处在下标为 m 的位置， $m+2$ 处在下标为 $m+1$ 的位置，以此类推。我们发现 m 正好是数组中第一个数值和下标不相等的下标，因此这个问题转换成在排序数组中找出第一个值和下标不相等的元素。

我们可以基于二分查找的算法用如下过程查找：如果中间元素的值和下标相等，那么下一轮查找只需要查找右半边；如果中间元素的值和下标不相等，并且它前面一个元素和它的下标相等，这意味着这个中间的数字正好是第一个值和下标不相等的元素，它的下标就是在数组中不存在的数字；如果中间元素的值和下标不相等，并且它前面一个元素和它的下标不相等，这意味着下一轮查找我们只需要在左半边查找即可。

下面是基于二分查找算法的参考代码：

```
int GetMissingNumber(const int* numbers, int length)
{
    if(numbers == nullptr || length <= 0)
        return -1;

    int left = 0;
    int right = length - 1;
    while(left <= right)
    {
        int middle = (right + left) >> 1;
        if(numbers[middle] != middle)
        {
```

```

        if(middle == 0 || numbers[middle - 1] == middle - 1)
            return middle;
        right = middle - 1;
    }
    else
        left = middle + 1;
}

if(left == length)
    return length;

// 无效的输入，比如数组不是按要求排序的，
// 或者有数字不在 0~n-1 范围之内
return -1;
}

```



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/53_02_MissingNumber



测试用例：

- 功能测试（缺失的数字位于数组的开始、中间或者末尾）。
- 边界值测试（数组中只有一个数字 0）
- 特殊输入测试（表示数组的指针为 nullptr 指针）。

题目三：数组中数值和下标相等的元素。

假设一个单调递增的数组里的每个元素都是整数并且是唯一的。请编程实现一个函数，找出数组中任意一个数值等于其下标的元素。例如，在数组 {−3, −1, 1, 3, 5} 中，数字 3 和它的下标相等。

我们很容易就能想到最直观的解法：从头到尾依次扫描数组中的数字，并逐一检验数字是不是和下标相等。显然，这种算法的时间复杂度是 $O(n)$ 。

由于数组是单调递增排序的，因此我们可以尝试用二分查找算法来进行优化。假设我们某一步抵达数组中的第 i 个数字。如果我们很幸运，该数字的值刚好也是 i ，那么我们就找到了一个数字和其下标相等。

那么当数字的值和下标不相等的时候该怎么办呢？假设数字的值为 m 。我们先考虑 m 大于 i 的情形，即数字的值大于它的下标。由于数组中的所有数字都唯一并且单调递增，那么对于任意大于0的 k ，位于下标 $i+k$ 的数字的值大于或等于 $m+k$ 。另外，因为 $m>i$ ，所以 $m+k>i+k$ 。因此，位于下标 $i+k$ 的数字的值一定大于它的下标。这意味着如果第 i 个数字的值大于 i ，那么它右边的数字都大于对应的下标，我们都可以忽略。下一轮查找我们只需要从它左边的数字中查找即可。

数字的值 m 小于它的下标 i 的情形和上面类似。它左边的所有数字的值都小于对应的下标，我们也可以忽略。感兴趣的读者请自己证明。

由于我们在每一步查找时都可以把查找的范围缩小一半，这是典型的二分查找的过程。下面是基于二分查找的参考代码：

```
int GetNumberSameAsIndex(const int* numbers, int length)
{
    if(numbers == nullptr || length <= 0)
        return -1;

    int left = 0;
    int right = length - 1;
    while(left <= right)
    {
        int middle = left + ((right - left) >> 1);
        if(numbers[middle] == middle)
            return middle;

        if(numbers[middle] > middle)
            right = middle - 1;
        else
            left = middle + 1;
    }

    return -1;
}
```



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/53_03_IntegerIdenticalToIndex



测试用例：

- 功能测试（数组中包含或者不包含数值和下标相等的元素）。
- 边界值测试（数组中只有一个数字；数值和下标相等的元素位于数组的开头或者结尾）。
- 特殊输入测试（表示数组的指针为 `nullptr` 指针）。



本题考点：

- 考查应聘者的知识迁移能力。我们都应该知道，二分查找算法可以用来在排序数组中查找一个数字。应聘者如果能够运用知识迁移能力，把问题转换成用二分查找算法在排序数组中查找某些特定的数字，那么这些问题也就解决了一大半。
- 考查应聘者对二分查找算法的理解程度。这道题实际上是二分查找算法的加强版。只有对二分查找算法有着深刻的理解，应聘者才有可能解决这个问题。

面试题 54：二叉搜索树的第 k 大节点

题目：给定一棵二叉搜索树，请找出其中第 k 大的节点。例如，在图 6.1 中的二叉搜索树里，按节点数值大小顺序，第三大节点的值是 4。

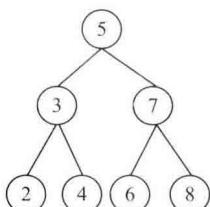


图 6.1 一棵有 7 个节点的二叉搜索树，其中按节点数值大小顺序，第三大节点的值是 4

如果按照中序遍历的顺序遍历一棵二叉搜索树，则遍历序列的数值是递增排序的。例如，图 6.1 中二叉搜索树的中序遍历序列是 {2, 3, 4, 5, 6, 7, 8}。因此，只需要用中序遍历算法遍历一棵二叉搜索树，我们就很容易找出它的第 k 大节点。

这道题实质上考查应聘者对中序遍历的理解。只要熟练掌握了中序遍历，应聘者很快就能写出如下代码：

```
BinaryTreeNode* KthNode(BinaryTreeNode* pRoot, unsigned int k)
{
    if(pRoot == nullptr || k == 0)
        return nullptr;

    return KthNodeCore(pRoot, k);
}

BinaryTreeNode* KthNodeCore(BinaryTreeNode* pRoot, unsigned int& k)
{
    BinaryTreeNode* target = nullptr;

    if(pRoot->m_pLeft != nullptr)
        target = KthNodeCore(pRoot->m_pLeft, k);

    if(target == nullptr)
    {
        if(k == 1)
            target = pRoot;

        k--;
    }

    if(target == nullptr && pRoot->m_pRight != nullptr)
        target = KthNodeCore(pRoot->m_pRight, k);

    return target;
}
```



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/54_KthNodeInBST



测试用例：

- 功能测试（各种形态不同的二叉搜索树）。
- 边界值测试（输入 k 为 0、1、二叉搜索树的节点数、二叉搜索树的节点数加 1）。
- 特殊输入测试（指向二叉搜索树根节点的指针为 `nullptr` 指针）。



本题考点：

- 考查应聘者的知识迁移能力。面试官期待应聘者能够运用中序遍历算法来解决这道面试题。
- 考查应聘者对二叉搜索树和中序遍历的特点的理解。如果应聘者理解二叉搜索树的中序遍历序列是递增的，那么他/她很容易就能找出第 k 大的节点。

面试题 55：二叉树的深度

题目一：二叉树的深度。

输入一棵二叉树的根节点，求该树的深度。从根节点到叶节点依次经过的节点（含根、叶节点）形成树的一条路径，最长路径的长度为树的深度。

二叉树的节点定义如下：

```
struct BinaryTreeNode
{
    int m_nValue;
    BinaryTreeNode* m_pLeft;
    BinaryTreeNode* m_pRight;
};
```

例如，在图 6.2 中的二叉树的深度为 4，因为它从根节点到叶节点最长的路径包含 4 个节点（从根节点 1 开始，经过节点 2 和节点 5，最终到达叶节点 7）。

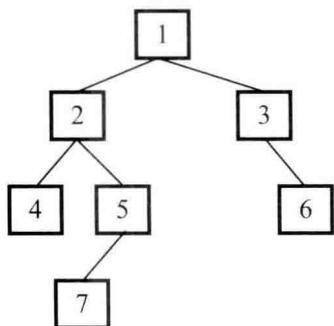


图 6.2 深度为 4 的二叉树

在本题中，面试官给出了一种树的深度的定义，我们可以根据这个定

义去得到树的所有路径，也就能得到最长的路径及它的长度。在面试题34“二叉树中和为某一值的路径”中我们详细讨论了如何记录树中的路径。这种思路的代码量比较大，我们可以尝试更加简洁的方法。

我们还可以从另外一个角度来理解树的深度。如果一棵树只有一个节点，那么它的深度为1。如果根节点只有左子树而没有右子树，那么树的深度应该是其左子树的深度加1；同样，如果根节点只有右子树而没有左子树，那么树的深度应该是其右子树的深度加1。如果既有右子树又有左子树，那么该树的深度就是其左、右子树深度的较大值再加1。比如，在图6.2所示的二叉树中，根节点为1的树有左、右两棵子树，其左、右子树的根节点分别为节点2和节点3。根节点为2的左子树的深度为3，而根节点为3的右子树的深度为2，因此，根节点为1的树的深度就是4。

这种思路用递归的方法很容易实现，只需对遍历的代码稍作修改即可。参考代码如下：

```
int TreeDepth(BinaryTreeNode* pRoot)
{
    if(pRoot == nullptr)
        return 0;

    int nLeft = TreeDepth(pRoot->m_pLeft);
    int nRight = TreeDepth(pRoot->m_pRight);

    return (nLeft > nRight) ? (nLeft + 1) : (nRight + 1);
}
```



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/55_01_TreeDepth



测试用例：

- 功能测试（输入普通的二叉树；二叉树中所有节点都没有左/右子树）。
- 特殊输入测试（二叉树只有一个节点；二叉树的头节点为nullptr指针）。

只要应聘者对二叉树这一数据结构很熟悉，就能很快写出上面的代码。如果公司对编程能力有较高的要求，那么面试官可能会追加一个与前面问题相关但难度更大的问题。

题目二：平衡二叉树。

输入一棵二叉树的根节点，判断该树是不是平衡二叉树。如果某二叉树中任意节点的左、右子树的深度相差不超过 1，那么它就是一棵平衡二叉树。例如，图 6.2 中的二叉树就是一棵平衡二叉树。

❖ 需要重复遍历节点多次的解法，简单但不足以打动面试官

有了求二叉树的深度的经验之后再解决这个问题，我们很容易就能想到一种思路：在遍历树的每个节点的时候，调用函数 TreeDepth 得到它的左、右子树的深度。如果每个节点的左、右子树的深度相差都不超过 1，那么按照定义它就是一棵平衡二叉树。这种思路对应的代码如下：

```
bool IsBalanced(BinaryTreeNode* pRoot)
{
    if(pRoot == nullptr)
        return true;

    int left = TreeDepth(pRoot->m_pLeft);
    int right = TreeDepth(pRoot->m_pRight);
    int diff = left - right;
    if(diff > 1 || diff < -1)
        return false;

    return IsBalanced(pRoot->m_pLeft) && IsBalanced(pRoot->m_pRight);
}
```

上面的代码固然简洁，但我们也要注意到由于一个节点会被重复遍历多次，这种思路的时间效率不高。例如，在函数 IsBalance 中输入图 6.2 中的二叉树，我们将首先判断根节点（节点 1）是不是平衡的。此时我们往函数 TreeDepth 里输入左子树的根节点（节点 2）时，需要遍历节点 4、5、7。接下来判断以节点 2 为根节点的子树是不是平衡树的时候，仍然会遍历节点 4、5、7。毫无疑问，重复遍历同一个节点会影响性能。接下来我们寻找不需要重复遍历的算法。

❖ 每个节点只遍历一次的解法，正是面试官喜欢的

如果我们用后序遍历的方式遍历二叉树的每个节点，那么在遍历到一个节点之前我们就已经遍历了它的左、右子树。只要在遍历每个节点的时候记录它的深度（某一节点的深度等于它到叶节点的路径的长度），我们就可以一边遍历一边判断每个节点是不是平衡的。下面是这种思路的参考代码：

```
bool IsBalanced(BinaryTreeNode* pRoot, int* pDepth)
{
    if(pRoot == nullptr)
    {
        *pDepth = 0;
        return true;
    }

    int left, right;
    if(IsBalanced(pRoot->m_pLeft, &left) && IsBalanced(pRoot->m_pRight, &right))
    {
        int diff = left - right;
        if(diff <= 1 && diff >= -1)
        {
            *pDepth = 1 + (left > right ? left : right);
            return true;
        }
    }

    return false;
}
```

我们只需给上面的函数传入二叉树的根节点及一个表示节点深度的整型变量即可。

```
bool IsBalanced(BinaryTreeNode* pRoot)
{
    int depth = 0;
    return IsBalanced(pRoot, &depth);
}
```

在上面的代码中，我们用后序遍历的方式遍历整棵二叉树。在遍历某节点的左、右子节点之后，我们可以根据它的左、右子节点的深度判断它是不是平衡的，并得到当前节点的深度。当最后遍历到树的根节点的时候，也就判断了整棵二叉树是不是平衡二叉树。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/55_02_BalancedBinaryTree



测试用例：

- 功能测试（平衡的二叉树；不是平衡的二叉树；二叉树中所有节点都没有左/右子树）。
- 特殊输入测试（二叉树中只有一个节点；二叉树的头节点为 `nullptr` 指针）。



本题考点：

- 考查应聘者对二叉树的理解及编程能力。这两道题的解法实际上只是树的遍历算法的应用。
- 考查应聘者对新概念的学习能力。面试官提出一个新的概念即树的深度，这就要求我们在较短的时间内理解这个概念并解决相关的问题。这是一种常见的面试题型。能在较短时间内掌握、理解新概念的能力就是一种学习能力。
- 考查应聘者的知识迁移能力。如果面试官先问如何求二叉树的深度，再问如何判断一棵二叉树是不是平衡的，那么应聘者应该从求二叉树深度的分析过程中得到启发，找到判断平衡二叉树的突破口。

面试题 56：数组中数字出现的次数

题目一：数组中只出现一次的两个数字。

一个整型数组里除两个数字之外，其他数字都出现了两次。请写程序找出这两个只出现一次的数字。要求时间复杂度是 $O(n)$ ，空间复杂度是 $O(1)$ 。

例如，输入数组 {2, 4, 3, 6, 3, 2, 5, 5}，因为只有 4 和 6 这两个数字只出现了一次，其他数字都出现了两次，所以输出 4 和 6。

这是一道比较难的题目，很少有人能在面试的时候不需要提示一下子想到最好的解决办法。一般当应聘者想了几分钟后还没有思路，面试官会

给出一些提示。面试官很有可能会说：你可以先考虑这个数组中只有一个数字只出现了一次，其他数字都出现了两次，怎么找出这个数字？

这两道题目都在强调一个（或两个）数字只出现一次，其他数字出现两次。这有什么意义呢？我们想到异或运算的一个性质：任何一个数字异或它自己都等于0。也就是说，如果我们从头到尾依次异或数组中的每个数字，那么最终的结果刚好是那个只出现一次的数字，因为那些成对出现两次的数字全部在异或中抵消了。

想明白怎么解决这个简单的问题之后，我们再回到原始的问题，看看能不能运用相同的思路。我们试着把原数组分成两个子数组，使得每个子数组包含一个只出现一次的数字，而其他数字都成对出现两次。如果能够这样拆分成两个数组，那么我们就可以按照前面的办法分别找出两个只出现一次的数字了。

我们还是从头到尾依次异或数组中的每个数字，那么最终得到的结果就是两个只出现一次的数字的异或结果，因为其他数字都出现了两次，在异或中全部抵消了。由于这两个数字肯定不一样，那么异或的结果肯定不为0，也就是说，在这个结果数字的二进制表示中至少有一位为1。我们在结果数字中找到第一个为1的位的位置，记为第n位。现在我们以第n位是不是1为标准把原数组中的数字分成两个子数组，第一个子数组中每个数字的第n位都是1，而第二个子数组中每个数字的第n位都是0。由于我们分组的标准是数字中的某一位是1还是0，那么出现了两次的数字肯定被分配到同一个子数组。因为两个相同的数字的任意一位都是相同的，我们不可能把两个相同的数字分配到两个子数组中去，于是我们已经把原数组分成了两个子数组，每个子数组都包含一个只出现一次的数字，而其他数字都出现了两次。我们已经知道如何在数组中找出唯一一个只出现一次的数字，因此，到此为止所有的问题都已经解决了。

举个例子，假设输入数组{2, 4, 3, 6, 3, 2, 5, 5}。当我们依次对数组中的每个数字进行异或运算之后，得到的结果用二进制表示是0010。异或得到的结果中的倒数第二位是1，于是我们根据数字的倒数第二位是不是1将该数组分为两个子数组。第一个子数组{2, 3, 6, 3, 2}中所有数字的倒数第二位都是1，而第二个子数组{4, 5, 5}中所有数字的倒数第二位都是0。接下来只要分别对这两个子数组求异或，就能找出第一个子数组中只出现一次的数字是6，而第二个子数组中只出现一次的数字是4。

想清楚整个过程之后再写代码就不难了。下面是参考代码：

```
void FindNumsAppearOnce(int data[], int length, int* num1, int* num2)
{
    if (data == nullptr || length < 2)
        return;

    int resultExclusiveOR = 0;
    for (int i = 0; i < length; ++ i)
        resultExclusiveOR ^= data[i];

    unsigned int indexOf1 = FindFirstBitIs1(resultExclusiveOR);

    *num1 = *num2 = 0;
    for (int j = 0; j < length; ++ j)
    {
        if(IsBit1(data[j], indexOf1))
            *num1 ^= data[j];
        else
            *num2 ^= data[j];
    }
}

unsigned int FindFirstBitIs1(int num)
{
    int indexBit = 0;
    while (((num & 1) == 0) && (indexBit < 8 * sizeof(int)))
    {
        num = num >> 1;
        ++ indexBit;
    }

    return indexBit;
}

bool IsBit1(int num, unsigned int indexBit)
{
    num = num >> indexBit;
    return (num & 1);
}
```

在上述代码中，FindFirstBitIs1 用来在整数 num 的二进制表示中找到最右边是 1 的位，IsBit1 的作用是判断在 num 的二进制表示中从右边数起的 indexBit 位是不是 1。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/56_01_NumbersAppearOnce



测试用例：

功能测试（数组中有多对重复的数字；数组中没有重复的数字）。

题目二：数组中唯一只出现一次的数字。

在一个数组中除一个数字只出现一次之外，其他数字都出现了三次。请找出那个只出现一次的数字。

如果我们把题目稍微改一改，那么就会容易很多：如果数组中的数字除一个只出现一次之外，其他数字都出现了两次。我们可以用 XOR 异或位运算解决这个简化的问题。由于两个相同的数字的异或结果是 0，我们把数组中所有数字异或的结果就是那个唯一只出现一次的数字。

可惜这种思路不能解决这里的问题，因为三个相同的数字的异或结果还是该数字。尽管我们这里不能应用异或运算，我们还是可以沿用位运算的思路。如果一个数字出现三次，那么它的二进制表示的每一位（0 或者 1）也出现三次。如果把所有出现三次的数字的二进制表示的每一位都分别加起来，那么每一位的和都能被 3 整除。

我们把数组中所有数字的二进制表示的每一位都加起来。如果某一位的和能被 3 整除，那么那个只出现一次的数字二进制表示中对应的那一位是 0；否则就是 1。

这种思路的参考代码如下：

```
int FindNumberAppearingOnce(int numbers[], int length)
{
    if(numbers == nullptr || length <= 0)
        throw new std::exception("Invalid input.");

    int bitSum[32] = {0};
    for(int i = 0; i < length; ++i)
    {
        int bitMask = 1;
        for(int j = 31; j >= 0; --j)
        {
            int bit = numbers[i] & bitMask;
            if(bit != 0)
                bitSum[j] += 1;
        }
    }
}
```

```

        bitMask = bitMask << 1;
    }

    int result = 0;
    for(int i = 0; i < 32; ++i)
    {
        result = result << 1;
        result += bitSum[i] % 3;
    }

    return result;
}

```

这种解法的时间效率是 $O(n)$ 。我们需要一个长度为 32 的辅助数组存储二进制表示的每一位的和。由于数组的长度是固定的，因此空间效率是 $O(1)$ 。该解法比其他两种直观的解法效率都要高：(1) 我们很容易就能从排序的数组中找到只出现一次的数字，但排序需要 $O(n \log n)$ 时间；(2) 我们也可以用一个哈希表来记录数组中每个数字出现的次数，但这个哈希表需要 $O(n)$ 的空间。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/56_02_NumberAppearingOnce



测试用例：

功能测试（唯一只出现一次的数字分别是 0、正数、负数；重复出现三次的数字分别是 0、正数、负数）。



本题考点：

- 考查应聘者的知识迁移能力。其他数字都出现两次而只有一个数字出现一次这个问题，很多应聘者都能想到解决办法。能不能把解决简单问题的思路迁移到复杂问题上，继续从位运算上想办法，是应聘者能否通过这轮面试的关键。
- 考查应聘者对二进制和位运算的理解。

面试题 57：和为 s 的数字

题目一：和为 s 的两个数字。

输入一个递增排序的数组和一个数字 s ，在数组中查找两个数，使得它们的和正好是 s 。如果有对数字的和等于 s ，则输出任意一对即可。

例如，输入数组 {1,2,4,7,11,15} 和数字 15。由于 $4+11=15$ ，因此输出 4 和 11。

在面试的时候，很重要的一点是应聘者要表现出很快的反应能力。只要想到一种方法，应聘者就可以马上告诉面试官，即使这种方法不一定是最好的。比如这个问题，很多人都能立即想到 $O(n^2)$ 的方法，也就是先在数组中固定一个数字，再依次判断数组中其余的 $n-1$ 个数字与它的和是不是等于 s 。面试官会告诉我们这不是最好的办法。不过这没有关系，至少面试官知道我们的思维还是比较敏捷的。

接着我们寻找更好的算法。我们先在数组中选择两个数字，如果它们的和等于输入的 s ，那么我们就找到了要找的两个数字。如果和小于 s 呢？我们希望两个数字的和再大一点。由于数组已经排好序了，我们可以考虑选择较小的数字后面的数字。因为排在后面的数字要大一些，那么两个数字的和也要大一些，就有可能等于输入的数字 s 了。同样，当两个数字的和大于输入的数字的时候，我们可以选择较大数字前面的数字，因为排在数组前面的数字要小一些。

我们以数组 {1,2,4,7,11,15} 及期待的和 15 为例详细分析一下这个过程。首先定义两个指针，第一个指针指向数组的第一个（最小的）数字 1，第二个指针指向数组的最后一个（最大的）数字 15。这两个数字的和 16 大于 15，因此我们把第二个指针向前移动一个数字，让它指向 11。这时候两个数字 1 与 11 的和是 12，小于 15。接下来我们把第一个指针向后移动一个数字指向 2，此时两个数字 2 与 11 的和是 13，还是小于 15。我们再次向后移动第一个指针，让它指向数字 4。数字 4 与 11 的和是 15，正是我们期待的结果。表 6.1 总结了在数组 {1,2,4,7,11,15} 中查找和为 15 的数对的过程。

表 6.1 在数组{1,2,4,7,11,15}中查找和为 15 的数对

| 步 骤 | 较小的数字 | 较大的数字 | 和 | 与 s 相比较 | 下一步操作 |
|-----|-------|-------|----|---------|-------------|
| 1 | 1 | 15 | 16 | 大于 | 选择 15 之前的数字 |
| 2 | 1 | 11 | 12 | 小于 | 选择 1 之后的数字 |
| 3 | 2 | 11 | 13 | 小于 | 选择 2 之后的数字 |
| 4 | 4 | 11 | 15 | 等于 | |

这一次面试官会首肯我们的思路，于是就可以动手写代码了。下面是一段参考代码：

```
bool FindNumbersWithSum(int data[], int length, int sum,
                        int* num1, int* num2)
{
    bool found = false;
    if(length < 1 || num1 == nullptr || num2 == nullptr)
        return found;

    int ahead = length - 1;
    int behind = 0;

    while(ahead > behind)
    {
        long long curSum = data[ahead] + data[behind];

        if(curSum == sum)
        {
            *num1 = data[behind];
            *num2 = data[ahead];
            found = true;
            break;
        }
        else if(curSum > sum)
            ahead--;
        else
            behind++;
    }

    return found;
}
```

在上述代码中，`ahead` 为较小的数字的下标，`behind` 为较大的数字的下标。由于数组是排序的，因此较小数字一定位于较大数字的前面，这就是 `while` 循环继续的条件是 `ahead>behind` 的原因。代码中只有一个 `while` 循环从两端向中间扫描数组，因此这种算法的时间复杂度是 $O(n)$ 。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/57_01_TwoNumbersWithSum



测试用例：

- 功能测试（数组中存在和为 s 的两个数；数组中不存在和为 s 的两个数）。
- 特殊输入测试（表示数组的指针为 `nullptr` 指针）。

看到应聘者比较轻松地解决了问题还有时间剩余，有些面试官喜欢追问和前面问题相关但稍微难一些的问题。

题目二：和为 s 的连续正数序列。

输入一个正数 s ，打印出所有和为 s 的连续正数序列（至少含有两个数）。例如，输入 15，由于 $1+2+3+4+5=4+5+6=7+8=15$ ，所以打印出 3 个连续序列 1~5、4~6 和 7~8。

有了解决前面问题的经验，我们也考虑用两个数 `small` 和 `big` 分别表示序列的最小值和最大值。首先把 `small` 初始化为 1，`big` 初始化为 2。如果从 `small` 到 `big` 的序列的和大于 s ，则可以从序列中去掉较小的值，也就是增大 `small` 的值。如果从 `small` 到 `big` 的序列的和小于 s ，则可以增大 `big`，让这个序列包含更多的数字。因为这个序列至少要有两个数字，我们一直增加 `small` 到 $(1+s)/2$ 为止。

以求和为 9 的所有连续序列为为例，我们先把 `small` 初始化为 1，`big` 初始化为 2。此时介于 `small` 和 `big` 之间的序列是 {1,2}，序列的和为 3，小于 9，所以我们下一步要让序列包含更多的数字。我们把 `big` 增加 1 变成 3，此时序列为 {1,2,3}。由于序列的和是 6，仍然小于 9，我们接下来再增加 `big` 变成 4，介于 `small` 和 `big` 之间的序列也随之变成 {1,2,3,4}。由于序列的和 10 大于 9，我们要删去序列中的一些数字，于是我们增加 `small` 变成 2，此时得到的序列为 {2,3,4}，序列的和正好是 9。我们找到了第一个和为 9 的连续序列，把它打印出来。接下来我们再增加 `big`，重复前面的过程，可以找到第二个和为 9 的连续序列 {4,5}。可以用表 6.2 总结整个过程。

表 6.2 求取和为 9 的连续序列的过程

| 步 骤 | small | big | 序 列 | 序列和 | 与 s 相比较 | 下一部操作 |
|-----|-------|-----|------------|-----|---------|--------------|
| 1 | 1 | 2 | 1, 2 | 3 | 小于 | 增加 big |
| 2 | 1 | 3 | 1, 2, 3 | 6 | 小于 | 增加 big |
| 3 | 1 | 4 | 1, 2, 3, 4 | 10 | 大于 | 增加 small |
| 4 | 2 | 4 | 2, 3, 4 | 9 | 等于 | 打印序列, 增加 big |
| 5 | 2 | 5 | 2, 3, 4, 5 | 14 | 大于 | 增加 small |
| 6 | 3 | 5 | 3, 4, 5 | 12 | 大于 | 增加 small |
| 7 | 4 | 5 | 4, 5 | 9 | 等于 | 打印序列 |

形成了清晰的解题思路之后，我们就可以开始写代码了。下面是这种思路的参考代码：

```

void FindContinuousSequence(int sum)
{
    if(sum < 3)
        return;

    int small = 1;
    int big = 2;
    int middle = (1 + sum) / 2;
    int curSum = small + big;

    while(small < middle)
    {
        if(curSum == sum)
            PrintContinuousSequence(small, big);

        while(curSum > sum && small < middle)
        {
            curSum -= small;
            small++;
        }

        if(curSum == sum)
            PrintContinuousSequence(small, big);
    }

    big++;
    curSum += big;
}

void PrintContinuousSequence(int small, int big)
{
    for(int i = small; i <= big; ++ i)
        printf("%d ", i);
}

```

```

    printf("\n");
}

```

在上述代码中，求连续序列的和应用了一个小技巧。通常我们可以用循环求一个连续序列的和，但考虑到每次操作之后的序列和操作之前的序列相比大部分数字都是一样的，只是增加或者减少了一个数字，因此我们可以在前一个序列的和的基础上求操作之后的序列的和。这样可以减少很多不必要的运算，从而提高代码的效率。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/57_02_ContinuousSquenceWithSum



测试用例：

- 功能测试（存在和为 s 的连续序列，如 9、100 等；不存在和为 s 的连续序列，如 4、0 等）。
- 边界值测试（连续序列的最小和 3）。



本题考点：

- 考查应聘者思考复杂问题的思维能力。应聘者如果能够通过一两个具体的例子找到规律，那么解决这个问题就容易多了。
- 考查应聘者的知识迁移能力。应聘者面对第二个问题的时候，能不能把解决第一个问题的思路应用到新的题目上，是面试官考查知识迁移能力的重要指标。

面试题 58：翻转字符串

题目一：翻转单词顺序。

输入一个英文句子，翻转句子中单词的顺序，但单词内字符的顺序不变。为简单起见，标点符号和普通字母一样处理。例如输入字符串 "I am a student."，则输出 "student. a am I"。

这道题目流传甚广，很多公司多次拿来作为面试题，很多应聘者也多次在各种博客或者书籍上看到通过两次翻转字符串的解法，于是很快就可以跟面试官解释清楚解题思路：第一步翻转句子中所有的字符。比如翻转 "I am a student." 中所有的字符得到 ".tneduts a ma I"，此时不但翻转了句子中单词的顺序，连单词内的字符顺序也被翻转了。第二步再翻转每个单词中字符的顺序，就得到了 "student. a am I"。这正是符合题目要求的输出。

这种思路的关键在于实现一个函数以翻转字符串中的一段。下面的函数 Reverse 可以完成这一功能：

```
void Reverse(char *pBegin, char *pEnd)
{
    if(pBegin == nullptr || pEnd == nullptr)
        return;

    while(pBegin < pEnd)
    {
        char temp = *pBegin;
        *pBegin = *pEnd;
        *pEnd = temp;

        pBegin++, pEnd--;
    }
}
```

接着我们可以用这个函数先翻转整个句子，再翻转句子中的每个单词。这种思路的参考代码如下：

```
char* ReverseSentence(char *pData)
{
    if(pData == nullptr)
        return nullptr;

    char *pBegin = pData;
    char *pEnd = pData;
    while(*pEnd != '0')
        pEnd++;
    pEnd--;

    // 翻转整个句子
    Reverse(pBegin, pEnd);

    // 翻转句子中的每个单词
    pBegin = pEnd = pData;
    while(*pBegin != '0')
    {
        if(*pBegin == ' ')
        {
            pBegin++;
            pEnd++;
        }
    }
}
```

```

    }
    else if(*pEnd == ' ' || *pEnd == '\0')
    {
        Reverse(pBegin, --pEnd);
        pBegin = ++pEnd;
    }
    else
    {
        pEnd++;
    }
}

return pData;
}

```

在英语句子中，单词被空格符号分隔，因此我们可以通过扫描空格来确定每个单词的起始和终止位置。在上述代码的翻转每个单词阶段，指针 pBegin 指向单词的第一个字符，而指针 pEnd 指向单词的最后一个字符。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/58_01_ReverseWordsInSentence



测试用例：

- 功能测试（句子中有多个单词；句子中只有一个单词）。
- 特殊输入测试（字符串指针为 nullptr 指针；字符串的内容为空；字符串中只有空格）。

有经验的面试官看到一个应聘者几乎不假思索就能想出一种比较巧妙的算法，就会觉得他之前可能见过这个题目。这时候很多面试官都会再问一个问题，以考查他是不是真的理解了这种算法。一种常见的考查办法就是问一个类似的但更难一点的问题。以这道题为例，如果面试官觉得应聘者之前看过这种思路，那么他可能再问第二个问题。

题目二：左旋转字符串。

字符串的左旋转操作是把字符串前面的若干个字符转移到字符串的尾部。请定义一个函数实现字符串左旋转操作的功能。比如，输入字符串 "abcdefg" 和数字 2，该函数将返回左旋转两位得到的结果 "cdefgab"。

要找到字符串旋转时每个字符移动的规律，不是一件轻松的事情。那我们是不是可以从解决第一个问题的思路中找到启发？在第一个问题中，如果输入的字符串之中只有两个单词，比如"hello world"，那么翻转这个句子中的单词顺序就得到了"world hello"。比较这两个字符串，我们是不是可以把"world hello"看成把原始字符串"hello world"的前面若干个字符转移到后面？也就是说这两个问题是非常相似的，我们同样可以通过翻转字符串的办法来解决第二个问题。

以"abcdefg"为例，我们可以把它分为两部分。由于想把它的前两个字符移到后面，我们就把前两个字符分到第一部分，把后面的所有字符分到第二部分。我们先分别翻转这两部分，于是就得到"bagfedc"。接下来翻转整个字符串，得到的"cdefgab"刚好就是把原始字符串左旋转两位的结果。

通过前面的分析，我们发现只需要调用3次前面的Reverse函数就可以实现字符串的左旋转功能。参考代码如下：

```
char* LeftRotateString(char* pStr, int n)
{
    if(pStr != nullptr)
    {
        int nLength = static_cast<int>(strlen(pStr));
        if(nLength > 0 && n > 0 && n < nLength)
        {
            char* pFirstStart = pStr;
            char* pFirstEnd = pStr + n - 1;
            char* pSecondStart = pStr + n;
            char* pSecondEnd = pStr + nLength - 1;

            // 翻转字符串的前面 n 个字符
            Reverse(pFirstStart, pFirstEnd);
            // 翻转字符串的后面部分
            Reverse(pSecondStart, pSecondEnd);
            // 翻转整个字符串
            Reverse(pFirstStart, pSecondEnd);
        }
    }

    return pStr;
}
```

想清楚思路之后再写代码是一件很容易的事情，但我们也不能掉以轻心。面试官在检查与字符串相关的代码时经常会发现两种问题：一是输入空指针nullptr时程序会崩溃；二是内存访问越界的问题，也就是试图访问不属于字符串的内存。例如，如果输入的n小于0，那么指针pStr+n指向的内存就不属于字符串。如果我们不排除这种情况，试图访问不属于字符串的内存时，程序可能会崩溃。

串的内存，就会留下严重的内存越界的安全隐患。在前面的代码中，我们添加了两个 if 判断语句，就是为了防止出现这两种问题。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/58_02_LeftRotateString



测试用例：

- 功能测试（把长度为 n 的字符串左旋转 0 个字符、1 个字符、2 个字符、 $n-1$ 个字符、 n 个字符、 $n+1$ 个字符）。
- 特殊输入测试（字符串的指针为 `nullptr` 指针）。



本题考点：

- 考查应聘者的知识迁移能力。当面试的时候遇到第二个问题，而之前我们做过“翻转句子中单词的顺序”这道题目，如果能够把多次翻转字符串的思路迁移过来，就能很轻易地解决字符串左旋转的问题。
- 考查应聘者对字符串的编程能力。

面试题 59：队列的最大值

题目一：滑动窗口的最大值。

给定一个数组和滑动窗口的大小，请找出所有滑动窗口里的最大值。例如，如果输入数组 {2, 3, 4, 2, 6, 2, 5, 1} 及滑动窗口的大小 3，那么一共有 6 个滑动窗口，它们的最大值分别为 {4, 4, 6, 6, 6, 5}，如表 6.3 所示。

如果采用蛮力法，那么这个问题似乎不难解决：可以扫描每个滑动窗口的所有数字并找出其中的最大值。如果滑动窗口的大小为 k ，则需要 $O(k)$ 时间才能找出滑动窗口里的最大值。对于长度为 n 的输入数组，这种算法的总时间复杂度是 $O(nk)$ 。

实际上，一个滑动窗口可以看成一个队列。当窗口滑动时，处于窗口的第一个数字被删除，同时在窗口的末尾添加一个新的数字。这符合队列的“先进先出”特性。如果能从队列中找出它的最大数，那么这个问题也就解决了。

在面试题 30 中，我们实现了一个可以用 $O(1)$ 时间得到最小值的栈。同样，也可以用 $O(1)$ 时间得到栈的最大值。同时在面试题 9 中，我们讨论了如何用两个栈实现一个队列。综合这两个问题的解决方案，我们发现，如果把队列用两个栈实现，由于可以用 $O(1)$ 时间得到栈中的最大值，那么也就可以用 $O(1)$ 时间得到队列的最大值，因此总的时间复杂度也就降到了 $O(n)$ 。

表 6.3 数组{2, 3, 4, 2, 6, 2, 5, 1}里大小为 3 的滑动窗口的最大值（滑动窗口用一对中括号表示）

| 数组中的滑动窗口 | 滑动窗口中的最大值 |
|--------------------------|-----------|
| [2, 3, 4], 2, 6, 2, 5, 1 | 4 |
| 2, [3, 4, 2], 6, 2, 5, 1 | 4 |
| 2, 3, [4, 2, 6], 2, 5, 1 | 6 |
| 2, 3, 4, [2, 6, 2], 5, 1 | 6 |
| 2, 3, 4, 2, [6, 2, 5], 1 | 6 |
| 2, 3, 4, 2, 6, [2, 5, 1] | 5 |

我们可以用这种方法来解决本题。不过这样就相当于在一轮面试的时间内要做两道面试题，时间未必够用。再来看看有没有其他的方法。

下面换一种思路。我们并不把滑动窗口的每个数值都存入队列，而是只把有可能成为滑动窗口最大值的数值存入一个两端开口的队列（如 C++ 标准模板库中的 deque）。接着以输入数组 {2, 3, 4, 2, 6, 2, 5, 1} 为例一步步分析。

数组的第一个数字是 2，把它存入队列。第二个数字是 3，由于它比前一个数字 2 大，因此 2 不可能成为滑动窗口中的最大值。先把 2 从队列里删除，再把 3 存入队列。此时队列中只有一个数字 3。针对第三个数字 4 的步骤类似，最终在队列中只剩下一个数字 4。此时滑动窗口中已经有 3 个数字，而它的最大值 4 位于队列的头部。

接下来处理第四个数字 2。2 比队列中的数字 4 小。当 4 滑出窗口之后，2 还有可能成为滑动窗口中的最大值，因此把 2 存入队列的尾部。现在队

列中有两个数字4和2，其中最大值4仍然位于队列的头部。

第五个数字是6。由于它比队列中已有的两个数字4和2都大，因此这时4和2已经不可能成为滑动窗口中的最大值了。先把4和2从队列中删除，再把数字6存入队列。这时候最大值6仍然位于队列的头部。

第六个数字是2。由于它比队列中已有的数字6小，所以把2也存入队列的尾部。此时队列中有两个数字，其中最大值6位于队列的头部。

第七个数字是5。在队列中已有的两个数字6和2里，2小于5，因此2不可能是一个滑动窗口的最大值，可以把它从队列的尾部删除。删除数字2之后，再把数字5存入队列。此时队列里剩下两个数字6和5，其中位于队列头部的是最大值6。

数组最后一个数字是1，把1存入队列的尾部。注意到位于队列头部的数字6是数组的第五个数字，此时的滑动窗口已经不包括这个数字了，因此应该把数字6从队列中删除。那么怎么知道滑动窗口是否包括一个数字？应该在队列里存入数字在数组里的下标，而不是数值。当一个数字的下标与当前处理的数字的下标之差大于或者等于滑动窗口的大小时，这个数字已经从窗口中滑出，可以从队列中删除了。

表6.4总结了上述步骤。我们注意到滑动窗口的最大值总是位于队列的头部。

表6.4 找出数组{2, 3, 4, 2, 6, 2, 5, 1}中大小为3的滑动窗口的最大值的步骤（在“队列中的下标”一列中，小括号前面的数字表示一个数字在输入数组中的下标。为了方便读者理解，下标对应的在数组中的数字在后面的小括号中标出）

| 步 骤 | 插入数字 | 滑动窗口 | 队列中的下标 | 最 大 值 |
|-----|------|---------|------------|-------|
| 1 | 2 | 2 | 0(2) | N/A |
| 2 | 3 | 2, 3 | 1(3) | N/A |
| 3 | 4 | 2, 3, 4 | 2(4) | 4 |
| 4 | 2 | 3, 4, 2 | 2(4), 3(2) | 4 |
| 5 | 6 | 4, 2, 6 | 4(6) | 6 |
| 6 | 2 | 2, 6, 2 | 4(6), 5(2) | 6 |
| 7 | 5 | 6, 2, 5 | 4(6), 6(5) | 6 |
| 8 | 1 | 2, 5, 1 | 6(5), 7(1) | 5 |

可以用如下 C++ 代码实现这个解决方案：

```
vector<int> maxInWindows(const vector<int>& num, unsigned int size)
{
    vector<int> maxInWindows;
    if(num.size() >= size && size >= 1)
    {
        deque<int> index;

        for(unsigned int i = 0; i < size; ++i)
        {
            while(!index.empty() && num[i] >= num[index.back()])
                index.pop_back();

            index.push_back(i);
        }

        for(unsigned int i = size; i < num.size(); ++i)
        {
            maxInWindows.push_back(num[index.front()]);

            while(!index.empty() && num[i] >= num[index.back()])
                index.pop_back();
            if(!index.empty() && index.front() <= (int)(i - size))
                index.pop_front();

            index.push_back(i);
        }
        maxInWindows.push_back(num[index.front()]);
    }

    return maxInWindows;
}
```

在上述代码中，`index` 是一个两端开口的队列，用来保存有可能是滑动窗口最大值的数字的下标。在存入一个数字的下标之前，首先要判断队列里已有数字是否小于待存入的数字。如果已有的数字小于待存入的数字，那么这些数字已经不可能是滑动窗口的最大值，因此它们将会被依次从队列的尾部删除（调用函数 `pop_back`）。同时，如果队列头部的数字已经从窗口里滑出，那么滑出的数字也需要从队列的头部删除（调用函数 `pop_front`）。由于队列的头部和尾部都有可能删除数字，这也是需要两端开口的队列的原因。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/59_01_MaxInSlidingWindow



测试用例：

- 功能测试（输入数组的数字大小无序；输入数组的数字单调递增；输入数组的数字单调递减）。
- 边界值测试（滑动窗口的大小为0、1、等于输入数组的长度、大于输入数组的长度）。
- 特殊输入测试（输入数组为空）。

题目二：队列的最大值。

请定义一个队列并实现函数 `max` 得到队列里的最大值，要求函数 `max`、`push_back` 和 `pop_front` 的时间复杂度都是 $O(1)$ 。

如前所述，滑动窗口可以看成一个队列，因此上题的解法可以用来实现带 `max` 函数的队列。下面是实现队列的参考代码：

```
template<typename T> class QueueWithMax
{
public:
    QueueWithMax() : currentIndex(0)
    {
    }

    void push_back(T number)
    {
        while(!maximums.empty() && number >= maximums.back().number)
            maximums.pop_back();

        InternalData internalData = { number, currentIndex };
        data.push_back(internalData);
        maximums.push_back(internalData);

        ++currentIndex;
    }

    void pop_front()
    {
        if(maximums.empty())
            throw new exception("queue is empty");

        if(maximums.front().index == data.front().index)
            maximums.pop_front();

        data.pop_front();
    }

    T max() const
    {
        return maximums.front().number;
    }
}
```

```

    {
        if(maximums.empty())
            throw new exception("queue is empty");

        return maximums.front().number;
    }

private:
    struct InternalData
    {
        T number;
        int index;
    };

    deque<InternalData> data;
    deque<InternalData> maximums;
    int currentIndex;
};

```

由于该解法和上题找滑动窗口的最大值类似，因此我们不再逐步分析队列插入和删除的操作，感兴趣的读者请自己分析。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/59_02_QueueWithMax



测试用例：

往队列末尾插入不同大小的数字并求最大值；从队列头部删除数字并求最大值。



本题考点：

- 考查应聘者分析问题的能力。无论是求滑动窗口的最大值还是求队列的最大值，都不是一道容易的面试题，应聘者可以通过举例的方法一步步分析，找出其中的规律。
- 考查应聘者的知识迁移能力。如果应聘者深入理解了滑动窗口最大值和队列最大值之间的联系，那么掌握了上述两道题中任意一道题的解法，就能顺利解答另外一道题。

6.4 抽象建模能力

计算机只是一种工具，它被用来解决实际生产生活中的问题。程序员的工作就是把各种现实问题抽象成数学模型并用计算机的编程语言表达出来，因此有些面试官喜欢从日常生活中抽取提炼出问题考查应聘者是否能建立数学模型并解决问题。要想顺利解决这种类型的问题，应聘者除了需要具备扎实的数学基础和编程能力，还需要具有敏锐的洞察力和丰富的想象力。

建模的第一步是选择合理的数据结构来表述问题。实际生产生活中的问题千变万化，而常用的数据结构只有有限的几种。我们在根据问题的特点综合考虑性能、编程难度等因素之后，选择最合适的数据结构来表达问题，也就是建立模型。比如在面试题61“扑克牌中的顺子”中，我们用一个数组表示一副牌，用11、12和13分别表示J、Q、K，并且用0表示大小王。在面试题62“圆圈中最后剩下的数字”中，我们可以用一个环形链表模拟一个圆圈。

建模的第二步是分析模型中的内在规律，并用编程语言表述这种规律。我们只有对现实问题进行深入细致的观察分析之后，才能找到模型中的规律，才有可能编程解决问题。例如，在本书2.4.1节提到的“青蛙跳台阶”问题中，它内在的规律是斐波那契数列。再比如面试题60“ n 个骰子的点数”问题，其本质是求数列 $f(n)=f(n-1)+f(n-2)+f(n-3)+f(n-4)+f(n-5)+f(n-6)$ 。找到这个规律之后，我们就可以分别用递归和循环两种不同的方法去写代码。然而，并不是所有问题的内在规律都是显而易见的。在面试题62“圆圈中最后剩下的数字”中，我们经过严密的数学分析之后才能找到每次从圆圈中删除数字的规律，从而找到一种不需要辅助环形链表的快速方法来解决问题。

面试题60： n 个骰子的点数

题目：把 n 个骰子扔在地上，所有骰子朝上一面的点数之和为 s 。输入 n ，打印出 s 的所有可能的值出现的概率。

玩过麻将的人都知道，骰子一共有6个面，每个面上都有一个点数，

对应的是1~6之间的一个数字。所以 n 个骰子的点数和的最小值为 n ，最大值为 $6n$ 。另外，根据排列组合的知识，我们还知道 n 个骰子的所有点数的排列数为 6^n 。要解决这个问题，我们需要先统计出每个点数出现的次数，然后把每个点数出现的次数除以 6^n ，就能求出每个点数出现的概率。

❖ 解法一：基于递归求骰子点数，时间效率不够高

现在我们考虑如何统计每个点数出现的次数。要想求出 n 个骰子的点数和，可以先把 n 个骰子分为两堆：第一堆只有一个；另一堆有 $n-1$ 个。单独的那一个有可能出现1~6的点数。我们需要计算1~6的每一种点数和剩下的 $n-1$ 个骰子来计算点数和。接下来把剩下的 $n-1$ 个骰子仍然分成两堆：第一堆只有一个；第二堆有 $n-2$ 个。我们把上一轮那个单独骰子的点数和这一轮单独骰子的点数相加，再和剩下的 $n-2$ 个骰子来计算点数和。分析到这里，我们不难发现这是一种递归的思路，递归结束的条件就是最后只剩下一个骰子。

我们可以定义一个长度为 $6n-n+1$ 的数组，将和为 s 的点数出现的次数保存到数组的第 $s-n$ 个元素里。基于这种思路，我们可以写出如下代码：

```
int g_maxValue = 6;

void PrintProbability(int number)
{
    if(number < 1)
        return;

    int maxSum = number * g_maxValue;
    int* pProbabilities = new int[maxSum - number + 1];
    for(int i = number; i <= maxSum; ++i)
        pProbabilities[i - number] = 0;

    Probability(number, pProbabilities);

    int total = pow((double)g_maxValue, number);
    for(int i = number; i <= maxSum; ++i)
    {
        double ratio = (double)pProbabilities[i - number] / total;
        printf("%d: %e\n", i, ratio);
    }

    delete[] pProbabilities;
}

void Probability(int number, int* pProbabilities)
{
    for(int i = 1; i <= g_maxValue; ++i)
```

```

        Probability(number, number, i, pProbabilities);
    }

void Probability(int original, int current, int sum,
                int* pProbabilities)
{
    if(current == 1)
    {
        pProbabilities[sum - original]++;
    }
    else
    {
        for(int i = 1; i <= g_maxValue; ++i)
        {
            Probability(original, current - 1, i + sum, pProbabilities);
        }
    }
}

```

上述思路很简洁，实现起来也容易。但由于是基于递归的实现，它有很多计算是重复的，从而导致当 `number` 变大时性能慢得让人不能接受。关于递归的性能讨论，详见本书 2.4.1 节。

❖ 解法二：基于循环求骰子点数，时间性能好

可以换一种思路来解决这个问题。我们可以考虑用两个数组来存储骰子点数的每个总数出现的次数。在一轮循环中，第一个数组中的第 n 个数字表示骰子和为 n 出现的次数。在下一轮循环中，我们加上一个新的骰子，此时和为 n 的骰子出现的次数应该等于上一轮循环中骰子点数和为 $n-1$ 、 $n-2$ 、 $n-3$ 、 $n-4$ 、 $n-5$ 与 $n-6$ 的次数的总和，所以我们把另一个数组的第 n 个数字设为前一个数组对应的第 $n-1$ 、 $n-2$ 、 $n-3$ 、 $n-4$ 、 $n-5$ 与 $n-6$ 个数字之和。基于这种思路，我们可以写出如下代码：

```

void PrintProbability(int number)
{
    if(number < 1)
        return;

    int* pProbabilities[2];
    pProbabilities[0] = new int[g_maxValue * number + 1];
    pProbabilities[1] = new int[g_maxValue * number + 1];
    for(int i = 0; i < g_maxValue * number + 1; ++i)
    {
        pProbabilities[0][i] = 0;
        pProbabilities[1][i] = 0;
    }

    int flag = 0;

```

```

for (int i = 1; i <= g_maxValue; ++i)
    pProbabilities[flag][i] = 1;

for (int k = 2; k <= number; ++k)
{
    for(int i = 0; i < k; ++i)
        pProbabilities[1 - flag][i] = 0;

    for (int i = k; i <= g_maxValue * k; ++i)
    {
        pProbabilities[1 - flag][i] = 0;
        for(int j = 1; j <= i && j <= g_maxValue; ++j)
            pProbabilities[1-flag][i]+=pProbabilities[flag][i-j];
    }

    flag = 1 - flag;
}

double total = pow((double)g_maxValue, number);
for(int i = number; i <= g_maxValue * number; ++i)
{
    double ratio = (double)pProbabilities[flag][i] / total;
    printf("%d: %e\n", i, ratio);
}

delete[] pProbabilities[0];
delete[] pProbabilities[1];
}

```

在上述代码中，我们定义了两个数组 `pProbabilities[0]` 和 `pProbabilities[1]` 来存储骰子的点数之和。一轮循环中，一个数组的第 n 项等于另一个数组的第 $n-1$ 、 $n-2$ 、 $n-3$ 、 $n-4$ 、 $n-5$ 及 $n-6$ 项的和。在下一轮循环中，我们交换这两个数组（通过改变变量 `flag` 实现）再重复这一计算过程。

值得注意的是，上述代码没有在函数里把一个骰子的最大点数硬编码 (Hard Code) 为 6，而是用一个变量 `g_maxValue` 来表示。这样做的好处是，如果某个厂家生产了其他点数的骰子，那么我们只需要在代码中修改一个地方，扩展起来很方便。如果在面试的时候我们能对面试官提起对程序扩展性的考虑，则一定能给面试官留下很好的印象。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/60_DicesProbability



测试用例：

- 功能测试（1、2、3、4个骰子的各点数的概率）。
- 特殊输入测试（输入0）。
- 性能测试（输入较大的数字，如11）。



本题考点：

- 考查应聘者的数学建模能力。不管采用哪种思路解决问题，我们都要先想到用数组来存放 n 个骰子的每个点数出现的次数，并通过分析点数的规律建立模型，最终找到解决方案。
- 考查应聘者对递归和循环的性能的理解。

面试题 61：扑克牌中的顺子

题目：从扑克牌中随机抽5张牌，判断是不是一个顺子，即这5张牌是不是连续的。2~10为数字本身，A为1，J为11，Q为12，K为13，而大、小王可以看成任意数字。

我们需要把扑克牌的背景抽象成计算机语言。不难想象，我们可以把5张牌看成由5个数字组成的数组。大、小王是特殊的数字，我们不妨把它们都定义为0，这样就能和其他扑克牌区分开来了。

接下来我们分析怎样判断5个数字是不是连续的，最直观的方法是把数组排序。值得注意的是，由于0可以当成任意数字，我们可以用0去补满数组中的空缺。如果排序之后的数组不是连续的，即相邻的两个数字相隔若干个数字，那么只要我们有足够的0可以补满这两个数字的空缺，这个数组实际上还是连续的。举个例子，数组排序之后为{0,1,3,4,5}，在1和3之间空缺了一个2，刚好我们有一个0，也就是我们可以把它当成2去填补这个空缺。

于是我们需要做3件事情：首先把数组排序；其次统计数组中0的个数；最后统计排序之后的数组中相邻数字之间的空缺总数。如果空缺的总数小于或者等于0的个数，那么这个数组就是连续的；反之则不连续。

最后我们还需要注意一点：如果数组中的非0数字重复出现，则该数

组不是连续的。换成扑克牌的描述方式就是：如果一副牌里含有对子，则不可能是顺子。

基于这种思路，我们可以写出如下代码：

```
bool IsContinuous(int* numbers, int length)
{
    if(numbers == nullptr || length < 1)
        return false;

    qsort(numbers, length, sizeof(int), compare);

    int numberOfZero = 0;
    int numberGap = 0;

    // 统计数组中 0 的个数
    for(int i = 0; i < length && numbers[i] == 0; ++i)
        ++numberOfZero;

    // 统计数组中的间隔数目
    int small = numberOfZero;
    int big = small + 1;
    while(big < length)
    {
        // 两个数相等，有对子，不可能是顺子
        if(numbers[small] == numbers[big])
            return false;

        numberGap += numbers[big] - numbers[small] - 1;
        small = big;
        ++big;
    }

    return (numberGap > numberOfZero) ? false : true;
}

int compare(const void *arg1, const void *arg2)
{
    return *(int*)arg1 - *(int*)arg2;
}
```

为了让代码显得简洁，上述代码调用 C 的库函数 `qsort` 排序。可能有人担心 `qsort` 的时间复杂度是 $O(n \log n)$ ，还不够快。由于扑克牌的值出现在 0~13 之间，我们可以定义一个长度为 14 的哈希表，这样在 $O(n)$ 时间内就能完成排序（本书 2.4.1 节有这种思路的例子）。通常我们认为不同级别的时间复杂度只有当 n 足够大的时候才有意义。由于本题中数组的长度是固定的，只有 5 张牌，那么 $O(n)$ 和 $O(n \log n)$ 不会有多少区别，我们可以选用简洁易懂的方法来实现算法。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/61_ContinousCards



测试用例：

- 功能测试（抽出的牌中有一个或者多个大、小王；抽出的牌中没有大、小王；抽出的牌中有对子）。
- 特殊输入测试（输入 `nullptr` 指针）。



本题考点：

考查应聘者的抽象建模能力。这道题目要求我们把熟悉的扑克牌转换为数组，把找顺子的过程通过排序、计数等步骤实现。这些都是把生活中的模型用程序语言来表达的例子。

面试题 62：圆圈中最后剩下的数字

题目： $0, 1, \dots, n-1$ 这 n 个数字排成一个圆圈，从数字 0 开始，每次从这个圆圈里删除第 m 个数字。求出这个圆圈里剩下的最后一个数字。

例如， $0, 1, 2, 3, 4$ 这 5 个数字组成一个圆圈（如图 6.3 所示），从数字 0 开始每次删除第 3 个数字，则删除的前 4 个数字依次是 2、0、4、1，因此最后剩下的数字是 3。

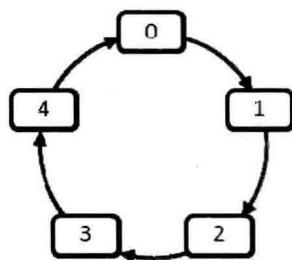


图 6.3 由 $0 \sim 4$ 这 5 个数字组成的圆圈

本题就是有名的约瑟夫（Josephuse）环问题。我们介绍两种解题方法：

一种方法是用环形链表模拟圆圈的经典解法；第二种方法是分析每次被删除的数字的规律并直接计算出圆圈中最后剩下的数字。

❖ 经典的解法，用环形链表模拟圆圈

既然题目中有一个数字圆圈，很自然的想法就是用一个数据结构来模拟这个圆圈。在常用的数据结构中，我们很容易想到环形链表。我们可以创建一个共有 n 个节点的环形链表，然后每次在这个链表中删除第 m 个节点。

如果面试官要求我们不能使用标准模板库里的数据容器来模拟环形链表，那么我们自己实现一个链表也不是很难的事情。如果面试官没有特殊要求，那么可以用模板库中的 `std::list` 来模拟一个环形链表。由于 `std::list` 本身并不是一个环形结构，因此每当迭代器（Iterator）扫描到链表末尾的时候，我们要记得把迭代器移到链表的头部，这样就相当于按照顺序在一个圆圈里遍历了。这种思路的代码如下：

```
int LastRemaining(unsigned int n, unsigned int m)
{
    if(n < 1 || m < 1)
        return -1;

    unsigned int i = 0;

    list<int> numbers;
    for(i = 0; i < n; ++ i)
        numbers.push_back(i);

    list<int>::iterator current = numbers.begin();
    while(numbers.size() > 1)
    {
        for(int i = 1; i < m; ++ i)
        {
            current++;
            if(current == numbers.end())
                current = numbers.begin();
        }

        list<int>::iterator next = ++ current;
        if(next == numbers.end())
            next = numbers.begin();

        -- current;
        numbers.erase(current);
        current = next;
    }

    return *(current);
}
```

如果我们用一两个例子仔细分析上述代码的运行过程，就会发现，实际上需要在环形链表里重复遍历很多遍。重复的遍历当然对时间效率有负面影响。这种方法每删除一个数字需要 m 步运算，共有 n 个数字，因此总的时间复杂度是 $O(mn)$ 。同时这种思路还需要一个辅助链表来模拟圆圈，其空间复杂度是 $O(n)$ 。接下来我们试着找到每次被删除的数字有哪些规律，希望能够找到更加高效的算法。

❖ 创新的解法，拿到 Offer 不在话下

首先我们定义一个关于 n 和 m 的方程 $f(n,m)$ ，表示每次在 n 个数字 $0, 1, \dots, n-1$ 中删除第 m 个数字最后剩下的数字。

在这 n 个数字中，第一个被删除的数字是 $(m-1)\%n$ 。为了简单起见，我们把 $(m-1)\%n$ 记为 k ，那么删除 k 之后剩下的 $n-1$ 个数字为 $0, 1, \dots, k-1, k+1, \dots, n-1$ ，并且下一次删除从数字 $k+1$ 开始计数。相当于在剩下的序列中， $k+1$ 排在最前面，从而形成 $k+1, \dots, n-1, 0, 1, \dots, k-1$ 。该序列最后剩下的数字也应该是关于 n 和 m 的函数。由于这个序列的规律和前面最初的序列不一样（最初的序列是从 0 开始的连续序列），因此该函数不同于前面的函数，记为 $f'(n-1, m)$ 。最初序列最后剩下的数字 $f(n, m)$ 一定是删除一个数字之后的序列最后剩下的数字，即 $f(n, m) = f'(n-1, m)$ 。

接下来我们把剩下的这 $n-1$ 个数字的序列 $k+1, \dots, n-1, 0, 1, \dots, k-1$ 进行映射，映射的结果是形成一个 $0 \sim n-2$ 的序列。

$$\begin{aligned}
 k+1 &\rightarrow 0 \\
 k+2 &\rightarrow 1 \\
 &\cdots \\
 n-1 &\rightarrow n-k-2 \\
 0 &\rightarrow n-k-1 \\
 1 &\rightarrow n-k \\
 &\cdots \\
 k-1 &\rightarrow n-2
 \end{aligned}$$

我们把映射定义为 p ，则 $p(x)=(x-k-1)\%n$ 。它表示如果映射前的数字是 x ，

那么映射后的数字是 $(x-k-1)\%n$ 。该映射的逆映射是 $p^{-1}(x)=(x+k+1)\%n$ 。

由于映射之后的序列和最初的序列具有同样的形式，即都是从0开始的连续序列，因此仍然可以用函数 f 来表示，记为 $f(n-1, m)$ 。根据我们的映射规则，映射之前的序列中最后剩下的数字 $f'(n-1, m)=p^{-1}[f(n-1, m)]=[f(n-1, m)+k+1]\%n$ ，把 $k=(m-1)\%n$ 代入得到 $f(n, m)=f'(n-1, m)=[f(n-1, m)+m]\%n$ 。

经过上面复杂的分析，我们终于找到了一个递归公式。要得到 n 个数字的序列中最后剩下的数字，只需要得到 $n-1$ 个数字的序列中最后剩下的数字，并以此类推。当 $n=1$ 时，也就是序列中开始只有一个数字0，那么很显然最后剩下的数字就是0。我们把这种关系表示为：

$$f(n, m) = \begin{cases} 0 & n = 1 \\ [f(n-1, m) + m] \% n & n > 1 \end{cases}$$

这个公式无论是用递归还是用循环，都很容易实现。下面是一段基于循环实现的代码：

```
int LastRemaining(unsigned int n, unsigned int m)
{
    if(n < 1 || m < 1)
        return -1;

    int last = 0;
    for (int i = 2; i <= n; i++)
        last = (last + m) % i;

    return last;
}
```

可以看出，这种思路的分析过程尽管非常复杂，但写出的代码却非常简洁，这就是数学的魅力。最重要的是，这种算法的时间复杂度是 $O(n)$ ，空间复杂度是 $O(1)$ ，因此，无论是在时间效率还是在空间效率上都优于第一种方法。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/62_LastNumberInCircle



测试用例：

- 功能测试（输入的 m 小于 n ，比如从最初有 5 个数字的圆圈中每次删除第 2、3 个数字；输入的 m 大于或者等于 n ，比如从最初有 6 个数字的圆圈中每次删除第 6、7 个数字）。
- 特殊输入测试（圆圈中有 0 个数字）。
- 性能测试（从最初有 4000 个数字的圆圈中每次删除第 997 个数字）。



本题考点：

- 考查应聘者的抽象建模能力。不管应聘者是用环形链表来模拟圆圈，还是分析被删除数字的规律，都要深刻理解这个问题的特点并编程实现自己的解决方案。
- 考查应聘者对环形链表的理解及应用能力。大部分面试官只要求应聘者基于环形链表的方法解决这个问题。
- 考查应聘者的数学功底及逻辑思维能力。少数对算法和数学基础要求很高的公司，面试官会要求应聘者不能使用 $O(n)$ 的辅助内存，这时候应聘者就只能静下心来一步步推导出每次删除的数字有哪些规律。

面试题 63：股票的最大利润

题目：假设把某股票的价格按照时间先后顺序存储在数组中，请问买卖该股票一次可能获得的最大利润是多少？例如，一只股票在某些时间节点的价格为 $\{9, 11, 8, 5, 7, 12, 16, 14\}$ 。如果我们能在价格为 5 的时候买入并在价格为 16 时卖出，则能收获最大的利润 11。

股票交易的利润来自股票买入和卖出价格的差价。当然，我们只能在买入某只股票之后才能卖出。如果把股票的买入价和卖出价两个数字组成一个数对，那么利润就是这个数对的差值。因此，最大的利润就是数组中所有数对的最大差值。

我们不难想到用蛮力法来解决这个问题，也就是找出数组中所有的数对，并逐一求出它们的差值。由于长度为 n 的数组中存在 $O(n^2)$ 个数对，因此该算法的时间复杂度是 $O(n^2)$ 。

我们也可以换一种思路。我们先定义函数 $\text{diff}(i)$ 为当卖出价为数组中第 i 个数字时可能获得的最大利润。显然，在卖出价固定时，买入价越低获得的利润越大。也就是说，如果在扫描到数组中的第 i 个数字时，只要我们能够记住之前的 $i-1$ 个数字中的最小值，就能算出在当前价位卖出时可能得到的最大利润。基于这种思路的代码如下：

```
int MaxDiff(const int* numbers, unsigned length)
{
    if(numbers == nullptr && length < 2)
        return 0;

    int min = numbers[0];
    int maxDiff = numbers[1] - min;

    for(int i = 2; i < length; ++i)
    {
        if(numbers[i - 1] < min)
            min = numbers[i - 1];

        int currentDiff = numbers[i] - min;
        if(currentDiff > maxDiff)
            maxDiff = currentDiff;
    }

    return maxDiff;
}
```

在上述代码中，变量 min 保存了数组前 $i-1$ 个数字的最小值，也就是之前股票的最低价。

由于我们只需要扫描数组一次，因此该算法的时间复杂度是 $O(n)$ ，比蛮力法的效率要高。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/63_MaximalProfit



测试用例：

- 功能测试（存储股票价格的数组无序、单调递增、单调递减）。
- 边界值测试（存储股票价格的数组中只有两个数字）。
- 特殊输入测试（指向数组的指针为 nullptr）。



本题考点：

- 考查应聘者的抽象建模能力。应聘者需要从股票买卖的特点入手总结出股票交易获得最大利润的条件。
- 考查应聘者对数组的编程能力。

6.5

发散思维能力

发散思维的特点是思维活动的多向性和变通性，也就是我们在思考问题时注重运用多思路、多方案、多途径来解决问题。对于同一个问题，我们可以从不同的方向、侧面和层次，采用探索、转换、迁移、组合和分解等方法，提出多种创新的解法。

通过考查发散思维能力，面试官能够了解应聘者探索新思路的激情。面试时面试官故意限制应聘者不能使用常规的思路，此时他在观察应聘者有没有积极的心态，是不是能够主动跳出常规思维的束缚从多角度去思考问题。比如在面试题 64 “求 $1+2+\dots+n$ ” 中，面试官有意限制不能使用乘除法及与循环、条件判断、选择相关的关键字。这个问题应该说是很难的。在难题面前，应聘者是轻言放弃，还是充满激情地寻找新思路、新方法，具有不同心态的应聘者在面试中的表现是大不一样的。

通过考查发散思维能力，面试官能够了解应聘者的灵活性和变通性。当常规思路遇到阻碍的时候，应聘者能不能及时地从另一个角度用不同的方法去分析问题，这些都能体现应聘者的创造力。在面试题 65 “不用加减乘除做加法” 中，当四则运算被限制使用的时候，应聘者能不能迅速地从二进制和位运算这个方向寻找突破口，都是其思维灵活性的直接体现。

通过考查发散思维能力，面试官还能了解应聘者知识面的广度和深度。

面试实际上是一个厚积薄发的过程。在遇到问题之后，应聘者如果具有广泛的知识面并且对各领域有较深的理解，那么他就更容易从不同的角度去思考问题。比如我们可以从构造函数、虚函数、函数指针及模板参数的实例化等不同角度去解决面试题 64 “求 $1+2+\dots+n$ ”。只有对 C++ 各方面的特性了如指掌，我们才能在遇到问题的时候将各个知识点信手拈来。同样，如果我们在学习数字电路相关课程的时候对 CPU 中加法器的原理有深刻的理解，那么自然就会想到从二进制和位运算的角度去思考解决面试题 65“不用加减乘除做加法”。

面试题 64：求 $1+2+\dots+n$

题目：求 $1+2+\dots+n$ ，要求不能使用乘除法、for、while、if、else、switch、case 等关键字及条件判断语句 (A?B:C)。

这个问题本身没有太多的实际意义，因为在软件开发中不可能有这么苛刻的限制。但不少面试官认为这是一道不错的能够考查应聘者发散思维能力的题目，而发散思维能够反映出应聘者知识面的宽度，以及对编程相关技术理解的深度。

通常求 $1+2+\dots+n$ 除了用公式 $n(n+1)/2$ ，无外乎循环和递归两种思路。由于已经明确限制 for 和 while 的使用，循环已经不能再用了。递归函数也需要用 if 语句或者条件判断语句来判断是继续递归下去还是终止递归，但现在题目已经不允许使用这两种语句了。

❖ 解法一：利用构造函数求解

我们仍然围绕循环做文章。循环只是让相同的代码重复执行 n 遍而已，我们完全可以不用 for 和 while 来达到这个效果。比如我们先定义一个类型，接着创建 n 个该类型的实例，那么这个类型的构造函数将确定会被调用 n 次。我们可以将与累加相关的代码放到构造函数里。如下代码正是基于这种思路：

```
class Temp
{
public:
    Temp() { ++N; Sum += N; }

    static void Reset() { N = 0; Sum = 0; }
    static unsigned int GetSum() { return Sum; }
}
```

```

private:
    static unsigned int N;
    static unsigned int Sum;
};

unsigned int Temp::N = 0;
unsigned int Temp::Sum = 0;

unsigned int Sum_Solution1(unsigned int n)
{
    Temp::Reset();

    Temp *a = new Temp[n];
    delete []a;
    a = nullptr;

    return Temp::GetSum();
}

```

❖ 解法二：利用虚函数求解

我们同样可以围绕递归做文章。既然不能在一个函数中判断是不是应该终止递归，那么我们不妨定义两个函数，一个函数充当递归函数的角色，另一个函数处理终止递归的情况，我们需要做的就是在两个函数里二选一。从二选一我们很自然地想到布尔变量，比如值为 `true(1)` 的时候调用第一个函数，值为 `false(0)` 的时候调用第二个函数。那现在的问题是如何把数值变量 `n` 转换成布尔值。如果对 `n` 连续做两次反运算，即`!!n`，那么非零的 `n` 转换为 `true`，`0` 转换为 `false`。有了上述分析，我们再来看下面的代码：

```

class A;
A* Array[2];

class A
{
public:
    virtual unsigned int Sum (unsigned int n)
    {
        return 0;
    }
};

class B: public A
{
public:
    virtual unsigned int Sum (unsigned int n)
    {
        return Array[!!n]->Sum(n-1) + n;
    }
};

```

```

};

int Sum_Solution2(int n)
{
    A a;
    B b;
    Array[0] = &a;
    Array[1] = &b;

    int value = Array[1]->Sum(n);

    return value;
}

```

这种思路是用虚函数来实现函数的选择。当 n 不为零时，调用函数 $B::Sum$ ；当 n 等于 0 时，调用函数 $A::Sum$ 。

❖ 解法三：利用函数指针求解

在纯 C 语言的编程环境中，我们不能使用虚函数，此时可以用函数指针来模拟，这样代码可能还更加直观一些。

```

typedef unsigned int (*fun)(unsigned int);

unsigned int Solution3_Tominator(unsigned int n)
{
    return 0;
}

unsigned int Sum_Solution3(unsigned int n)
{
    static fun f[2] = {Solution3_Tominator, Sum_Solution3};
    return n + f[!n](n - 1);
}

```

❖ 解法四：利用模板类型求解

另外，我们还可以让编译器帮助完成类似于递归的计算。看如下代码：

```

template <unsigned int n> struct Sum_Solution4
{
    enum Value { N = Sum_Solution4<n - 1>::N + n };
};

template <> struct Sum_Solution4<1>
{
    enum Value { N = 1 };
};

```

$\text{Sum_Solution4}<100>::N$ 就是 $1+2+\dots+100$ 的结果。当编译器看到 Sum_

Solution4<100>时，就会为模板类 Sum_Solution4 以参数 100 生成该类型的代码。但以 100 为参数的类型需要得到以 99 为参数的类型，因为 $\text{Sum}_\text{Solution4}<\!\!100\!\!>::\text{N} = \text{Sum}_\text{Solution4}<\!\!99\!\!>::\text{N} + 100$ 。这个过程会一直递归到参数为 1 的类型，由于该类型已经显式定义，编译器无须生成，递归编译到此结束。由于这个过程是在编译过程中完成的，因此要求输入 n 必须是在编译期间就能确定的常量，不能动态输入，这是该方法最大的缺点。而且编译器对递归编译代码的递归深度是有限制的，也就是要求 n 不能太大。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/64_Accumulate



测试用例：

- 功能测试（输入 5、10 求 $1+2+\dots+5$ 和 $1+2+\dots+10$ ）。
- 边界值测试（输入 0 和 1）。



本题考点：

- 考查应聘者的发散思维能力。当习以为常的方法被限制使用的时候，应聘者是否能发挥创造力，打开思路想出新的办法，是能否通过面试的关键所在。
- 考查应聘者的知识面的广度和深度。上面提供的几种解法涉及构造函数、静态变量、虚拟函数、函数指针、模板类型的实例化等知识点。只有深刻理解了相关的概念，才能在需要的时候信手拈来。这就是厚积薄发的过程。

面试题 65：不用加减乘除做加法

题目：写一个函数，求两个整数之和，要求在函数体内不得使用“+”、“-”、“×”、“÷”四则运算符号。

面试的时候被问到这个问题，很多人都在想：四则运算都不能用，那

还能用什么啊？可是问题总是要解决的，我们只能打开思路去思考各种可能性。首先我们可以分析人们是如何做十进制加法的，比如是如何得出 $5+17=22$ 这个结果的。实际上，我们可以分成三步进行：第一步只做各位相加不进位，此时相加的结果是 12（个位数 5 和 7 相加不要进位是 2，十位数 0 和 1 相加结果是 1）；第二步做进位， $5+7$ 中有进位，进位的值是 10；第三步把前面两个结果加起来， $12+10$ 的结果是 22，刚好 $5+17=22$ 。

我们一直在想，求两数之和四则运算都不能用，那还能用什么？对数字做运算，除四则运算之外，也就只剩下位运算了。位运算是针对二进制的，我们就以二进制再来分析一下前面的“三步走”策略对二进制是不是也适用。

5 的二进制是 101，17 的二进制是 10001。我们还是试着把计算分成三步：第一步各位相加但不计进位，得到的结果是 10100（最后一位两个数都是 1，相加的结果是二进制的 10。这一步不计进位，因此结果仍然是 0）；第二步记下进位，在这个例子中只在最后一位相加时产生一个进位，结果是二进制的 10；第三步把前两步的结果相加，得到的结果是 10110，转换成十进制正好是 22。由此可见“三步走”策略对二进制也是适用的。

接下来我们试着把二进制的加法用位运算来替代。第一步不考虑进位对每一位相加。0 加 0、1 加 1 的结果都是 0，0 加 1、1 加 0 的结果都是 1。我们注意到，这和异或的结果是一样的。对异或而言，0 和 0、1 和 1 的异或结果是 0，而 0 和 1、1 和 0 的异或结果是 1。接着考虑第二步进位，对 0 加 0、0 加 1、1 加 0 而言，都不会产生进位，只有 1 加 1 时，会向前产生一个进位。此时我们可以想象成两个数先做位与运算，然后再向左移动一位。只有两个数都是 1 的时候，位与得到的结果是 1，其余都是 0。第三步把前两个步骤的结果相加。第三步相加的过程依然是重复前面两步，直到不产生进位为止。

把这个过程想清楚之后，写出的代码非常简洁。下面是一段基于循环实现的参考代码：

```
int Add(int num1, int num2)
{
    int sum, carry;
    do
    {
        sum = num1 ^ num2;
        carry = (num1 & num2) << 1;
```

```

        num1 = sum;
        num2 = carry;
    }
    while(num2 != 0);

    return num1;
}

```



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/65_AddTwoNumbers



测试用例：

输入正数、负数和0。



本题考点：

- 考查应聘者的发散思维能力。当“+”、“-”、“×”、“÷”运算符都不能使用时，应聘者能不能打开思路想到用位运算做加法，是能否顺利解决这个问题的关键。
- 考查应聘者对二进制和位运算的理解。



相关问题：

不使用新的变量，交换两个变量的值。比如有两个变量 a 、 b ，我们希望交换它们的值。有两种不同的方法：

| 基于加减法 | 基于异或运算 |
|--|---|
| $a = a + b;$ $b = a - b;$ $a = a - b;$ | $a = a \wedge b;$ $b = a \wedge b;$ $a = a \wedge b;$ |

面试题 66：构建乘积数组

题目：给定一个数组 $A[0, 1, \dots, n-1]$ ，请构建一个数组 $B[0, 1, \dots, n-1]$ ，其中 B 中的元素 $B[i] = A[0] \times A[1] \times \dots \times A[i-1] \times A[i+1] \times \dots \times A[n-1]$ 。不能使用除法。

如果没有不能使用除法的限制，则可以用公式 $\prod_{j=0}^{n-1} A[j] / A[i]$ 求得 $B[i]$ 。在使用除法时，要特别注意 $A[i]$ 等于 0 的情况。

现在要求不能使用除法，只能用其他方法。一种直观的解法是用连乘 $n-1$ 个数字得到 $B[i]$ 。显然这种方法需要 $O(n^2)$ 的时间构建整个数组 B 。

好在还有更高效的算法。可以把 $B[i] = A[0] \times A[1] \times \cdots \times A[i-1] \times A[i+1] \times \cdots \times A[n-1]$ 看成 $A[0] \times A[1] \times \cdots \times A[i-1]$ 和 $A[i+1] \times \cdots \times A[n-2] \times A[n-1]$ 两部分的乘积。因此，数组 B 可以用一个矩阵来创建（见图 6.4）。在图中， $B[i]$ 为矩阵中第 i 行所有元素的乘积。

| | | | | | | |
|-----------|-------|-------|-------|-----------|-----------|-----------|
| B_0 | 1 | A_1 | A_2 | … | A_{n-2} | A_{n-1} |
| B_1 | A_0 | 1 | A_2 | … | A_{n-2} | A_{n-1} |
| B_2 | A_0 | A_1 | 1 | … | A_{n-2} | A_{n-1} |
| … | A_0 | A_1 | … | 1 | A_{n-2} | A_{n-1} |
| B_{n-2} | A_0 | A_1 | … | A_{n-3} | 1 | A_{n-1} |
| B_{n-1} | A_0 | A_1 | … | A_{n-3} | A_{n-2} | 1 |

图 6.4 把数组 B 看成由一个矩阵来创建

不妨定义 $C[i] = A[0] \times A[1] \times \cdots \times A[i-1]$, $D[i] = A[i+1] \times \cdots \times A[n-2] \times A[n-1]$ 。
 $C[i]$ 可以用自上而下的顺序计算出来，即 $C[i] = C[i-1] \times A[i-1]$ 。类似的， $D[i]$ 也可以用自下而上的顺序计算出来，即 $D[i] = D[i+1] \times A[i+1]$ 。

下面是这种思路的 C++ 实现，数组用标准模板库中的 vector 表示。

```
void multiply(const vector<double>& array1, vector<double>& array2)
{
    int length1 = array1.size();
    int length2 = array2.size();

    if(length1 == length2 && length2 > 1)
    {
        array2[0] = 1;
        for(int i = 1; i < length1; ++i)
        {
            array2[i] = array2[i - 1] * array1[i - 1];
        }

        double temp = 1;
        for(int i = length1 - 2; i >= 0; --i)
        {

```

```

        temp *= array1[i + 1];
        array2[i] *= temp;
    }
}
}

```

显然这种思路的时间复杂度是 $O(n)$ ，这比前面提到的直观的解法效率要高。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/66_ConstuctArray



测试用例：

- 功能测试（输入数组包含正数、负数、一个0、多个0）。
- 边界值测试（输入数组的长度为0）。



本题考点：

- 考查应聘者的发散思维能力。这道题目有两种常规解法：一种是把所有数字都相乘再分别除以各个数字，但题目已经限定不能使用除法；另一种解法是连乘 $n-1$ 个数字得到 $B[i]$ 。通常面试官会告知应聘者还有比 $O(n^2)$ 更高效的算法。此时应聘者不能放弃，还要继续打开思路，多角度去分析解答问题。
- 考查应聘者对数组的理解和编程能力。

6.6

本章小结

面试是我们展示自己综合素质的时候。除了扎实的编程能力，我们还需要表现自己的沟通能力和学习能力，以及知识迁移能力、抽象建模能力和发散思维能力等方面的综合实力，如图6.5所示。

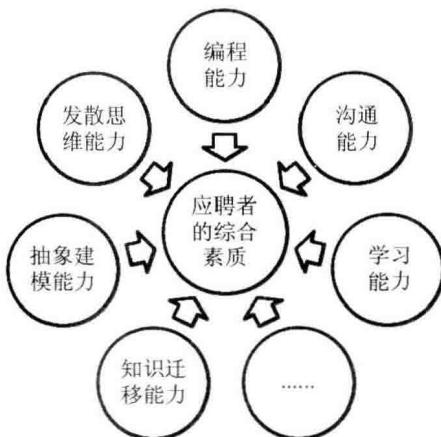


图 6.5 应聘者综合能力的组成

面试官对沟通能力、学习能力的考查贯穿面试的始终。面试官不仅会留意我们回答问题时的言语谈吐，还会关注我们是否能抓住问题的本质从而提出有针对性的问题。通常面试官认为善于提问的人有较好的沟通能力和学习能力。

知识迁移能力能帮助我们轻松地解决很多问题。有些面试官在提问一道难题之前，会问一道相关但比较简单的题目，他希望我们能够从解决简单问题的过程中受到启发，最终解决较为复杂的问题。另外，我们在面试之前可以做一些练习。如果面试的时候碰到类似的题目，就可以应用之前的方法。这要求我们平时要有一定的积累，并且每做完一道题之后都要总结解题方法。

有一类很有意思的面试题是从日常生活中提炼出来的，面试官用这种类型的问题来考查我们的抽象建模能力。为了解决这种类型的问题，我们先用适当的数据结构表述模型，再分析模型中的内在规律，从而确定计算方法。

有些面试官喜欢在面试的时候限制使用常规的思路。这时候就需要我们充分发挥发散思维能力，跳出常规思路的束缚，从不同的角度去尝试新的办法。

第 7 章

两个面试案例

在第 1 章中，我们讨论了面试的流程。通常一轮面试是从面试官对照简历了解应聘者的项目经历及掌握的技能开始的。在介绍自己的项目经历时，应聘者可以参照 STAR 模型，着重介绍自己完成的工作（包括基于什么平台、用了哪些技术、实现了哪些算法等），以及最终对项目组的贡献。

接着进入重头戏——技术面试环节。在这一环节中，面试官会从编程语言、数据结构和算法等方面考查应聘者的基础知识是否扎实全面（详见第 2 章），并且很有可能会要求应聘者编程实现一两个函数。如果碰到的面试题很简单，则应聘者也不能掉以轻心，一定要从基本功能、边界条件和错误处理等方面确保代码的完整性和鲁棒性（详见第 3 章）。如果碰到的题目很难，则应聘者可以尝试画图让抽象的问题变得形象化，也可以尝试举几个具体的例子去分析隐含的规律，还可以尝试把大的问题分解成两个或者多个小问题再递归地解决小问题。这 3 种方法能够帮助应聘者形成清晰的思路，从而解决复杂的难题（详见第 4 章）。很多面试题都不止一种解决方案，应聘者可以从时间复杂度和空间复杂度两个方面选择最优的解法（详见第 5 章）。在面试过程中，面试官除了关注应聘者的编程能力，还会关注应聘者的沟通能力和学习能力，并有可能考查应聘者的知识迁移能力、抽象建模能力和发散思维能力（详见第 6 章）。

在面试结束前的几分钟，面试官会给应聘者机会问几个最感兴趣的问题。应聘者可以从当前招聘的项目及其团队等方面提出几个问题。不建议应聘者在技术面试的时候向面试官询问薪资情况，或者立即打听面试结果。

接下来是两个典型的面试案例，我们从中可以直观地感受到面试的整个过程。在第一个案例（详见7.1节）中，我们将看到面试过程中很多应聘者都犯过的错误；而在第二个案例（详见7.2节）中，我们将看到面试官所认可的表现。我们希望应聘者能够少犯甚至不犯错误，在面试过程中充分表现出自己的综合素质，同时也衷心祝愿每名应聘者都能拿到自己心仪的Offer。

7.1

案例一：（面试题67）把字符串转换成整数

面试官：看你简历上写的是精通C/C++语言，这两门语言你用了几年？

应聘者：从大一算起的话，快六、七年了。

面试官：也是C/C++的老程序员了嘛（微笑），那先问一个C++的问题（递给应聘者一张A4纸，上面有一段打印的代码，如下面所示）。你能不能分析一下这段代码的输出？

```
class A
{
private:
    int n1;
    int n2;
public:
    A(): n2(0), n1(n2 + 2)
    {
    }

    void Print()
    {
        std::cout << "n1: " << n1 << ", n2: " << n2 << std::endl;
    }
};

int _tmain(int argc, _TCHAR* argv[])
{
    A a;
    a.Print();

    return 0;
}
```

应聘者：（看了一下代码，略作思考）n1是2，而n2是0。

面试官：为什么？

应聘者：在构造函数的初始化列表中，n2 先被初始化为 0，n2 的值就是 0 了。接下来再用 n2+2 初始化 n1，所以 n1 的值就是 2。

[注：应聘者这个问题的回答是错误的，详见后面的“面试官点评”]

面试官：C++是按照在初始化列表中的顺序初始化成员变量的吗？

应聘者：（一脸困惑）不是这样吗？我不太清楚。



面试官心理：

对成员变量的初始化顺序完全没有概念就号称自己“精通”C++，也太言过其实了。算了，C++就不接着问了，看看你的编程能力。

面试官：没关系，我们换一道题目。能不能介绍一下 C 语言的库函数中 atoi 的作用？

应聘者：atoi 用来把一个字符串转换成一个整数。比如，输入字符串 "123"，它的输出是数字 123。

面试官：对的。现在就请你写一个函数 StrToInt，实现把字符串转换成整数这个功能。当然，不能使用 atoi 或者其他类似的库函数。你看有没有问题？

应聘者：（嘴角出现一丝自信的笑容）没有问题。

应聘者马上开始在白纸上写出了如下代码：

```
int StrToInt(char* string)
{
    int number = 0;
    while(*string != 0)
    {
        number = number * 10 + *string - '0';
        ++string;
    }

    return number;
}
```

应聘者：（放下笔）我已经写好了。



面试官心理：

我出的题目有这么简单吗？你也太小看我了。

面试官：这么快？（稍微看了看代码）你觉得这代码有没有问题？仔细检查一下看看。

应聘者：（从头开始读代码）哦，不好意思，忘了检查字符串是空指针的情况。

应聘者拿起笔，在原来的代码上添加两行新的代码。修改之后的代码如下：

```
int StrToInt(char* string)
{
    if(string == nullptr)
        return 0;

    int number = 0;
    while(*string != 0)
    {
        number = number * 10 + *string - '0';
        ++string;
    }

    return number;
}
```

面试官：改好了？（看了一下新的代码）当字符串为空的时候，你的返回是0。如果输入的字符串是"0"，那么返回是什么？

应聘者：也是0。

面试官：两种情况都得到返回值0，那么当这个函数的调用者得到返回值0的时候，他怎么知道是哪种情况？

应聘者：（脸上表情有些困惑）不知道。

面试官：你知道atoi是怎么区分的吗？

应聘者：（努力回忆，有些慌张）不记得了。

面试官：atoi是通过一个全局变量来区分的。如果是非法输入，则返回0并把这个全局变量设为一个特殊标记。如果输入是"0"，则返回0，不会设置全局变量。这样，当atoi的调用者得到返回值0的时候，可以通过检查全局变量得知输入究竟是非法输入还是字符串"0"。

应聘者：哦。（拿起笔准备写代码）我马上修改。

面试官：等一下，除了空字符串，还有没有其他类型的非法输入？

应聘者：（陷入思考，额头上出现汗珠）如果字符串中含有'0'~'9'之外的字符，那么这样的输入也是非法的。

面试官：所有'0'~'9'之外的字符都是非法的吗？

应聘者：加号和减号应该也是合法的输入字符。

面试官：对的。先好好想想，想清楚了再开始写代码。

应聘者思考几分钟后，写下了如下代码：

```
enum Status {kValid = 0, kInvalid};
int g_nStatus = kValid;

int StrToInt(const char* str)
{
    g_nStatus = kInvalid;
    int num = 0;

    if(str != nullptr)
    {
        const char* digit = str;

        bool minus = false;
        if(*digit == '+')
            digit++;
        else if(*digit == '-')
        {
            digit++;
            minus = true;
        }

        while(*digit != '\0')
        {
            if(*digit >= '0' && *digit <= '9')
            {
                num = num * 10 + (*digit - '0');

                digit++;
            }
            else
            {
                num = 0;
                break;
            }
        }

        if(*digit == '\0')
        {
            g_nStatus = kValid;
            if(minus)
                num = 0 - num;
        }
    }

    return num;
}
```

面试官：（看到应聘者写完了）能不能简要地解释一下你的代码？

应聘者：我定义了一个全局变量 `g_nStatus` 来标记是不是遇到了非法输入。如果输入的字符串指针是空指针，则标记该全局变量然后直接返回。接下来我开始遍历字符串中的所有字符。由于正负号只可能出现在字符串的第一个字符，我们先处理字符串的第一个字符。如果第一个字符是负号，则标记当前的数字是负数，并在最后确保返回值是负数。在处理后续字符时，当遇到'0'~'9'之外的字符时，终止遍历。如果遇到了数字，则把数值累加上去。



面试官心理：

这段代码已经写得不错了，你的编程能力看起来还不错，只是编程的习惯不太好，不会在编码之前想好可能有哪些输入，从而在代码中留下太多的漏洞。这次修改的几个问题都是我提醒你的，没有提醒的你自己没有找出一个。

面试官：不错。觉得功能上还有什么遗漏吗？

应聘者：还有遗漏？（思索良久）要不要考虑溢出？

面试官：你觉得呢？如果输入的是一个空字符串""，你觉得应该输出什么？

应聘者：""不是一个数字，我想应该返回 0，同时把 `g_nStatus` 设为非法输入。

面试官：那你能分析一下你现在的输出是什么吗？

应聘者：（紧张，声音有些发抖）好像返回值是 0，但没有设置 `g_nStatus` 为非法输入？

面试官：嗯。我们再考虑一些有意思的输入，比如输入的字符串只有一个正号或者负号，你期待的输出是什么？

应聘者：如果只有一个正号或者负号，后面没有跟着数字，那么我想也不是有效的输入。（开始分析代码）我的返回值是 0，但不会设置 `g_nStatus`。

面试官：由于时间也差不多了，已经没有时间给你再进行修改了。我的问题问完了，你有什么问题需要问我的吗？

应聘者：你们公司工资待遇怎么样？

面试官：你的期望值是多少呢？

应聘者：我有不少同学的月薪税前超过 12000 元，我不想低于他们。



面试官心理：

我不是 HR，别和我谈工资。

面试官：我们公司由人事部门统一确定应届毕业生的工资，所以你的这个问题我不能直接回答，不过你的期望值我倒可以转告 HR。

应聘者：好的，谢谢。

面试官：还有其他问题吗？

应聘者：没有了。

面试官：那这轮面试就到这里结束吧。

面试官点评：

这名应聘者在简历中写他精通 C/C++，本来我对他的表现是充满了期待的。但在他回答错了第一道 C++ 的语法题之后，他给我留下的印象就不是很好了。这就是希望越大失望越大吧。实际上，构造函数的初始化列表是 C++ 中经常使用的一个概念。在 C++ 中，成员变量的初始化顺序只与它们在类中声明的顺序有关，而与在初始化列表中的顺序无关。在前面的问题中，n1 先于 n2 被声明，因此 n1 也会在 n2 之前被初始化，所以我们会先用 n2+2 去初始化 n1。由于 n2 这个时候还没有被初始化，因此它的值是随机的。用此时的 n2 加上 2 去初始化 n1，n1 的值只是一个随机值。接下来再用 0 初始化 n2，因此最终 n2 的值是 0。

接下来要求应聘者把一个字符串转换成整数，这看起来是一道很简单的题目，实现其基本功能，大部分人都能用 10 行之内的代码解决。可是，当我们把很多特殊情况即测试用例都考虑进去时，却不是一件容易的事情。解决数值转换问题本身不难，但我希望在写转换数值的代码之前，应聘者至少能把空指针 nullptr、空字符串 ""、正负号、溢出等方方面面的测试用例都考虑到，并在写代码的时候对这些特殊的输入都定义好合理的输出。当然，这些输出并不一定要和 atoi 完全保持一致，但必须要有显式的说明，和面试官沟通好。

这名应聘者最大的问题就是还没有养成在写代码之前考虑所有可能的

测试用例的习惯，逻辑不够严谨，因此一开始的代码只处理了最简单的数值转换。后来我每提醒他一处特殊的测试用例，他就改一处代码。尽管他已经进行了两次修改，但仍然有不少明显的漏洞，如特殊输入空字符串""、边界条件比如最大的正整数与最小的负整数等。由于这道题的思路本身不难，因此我希望他能把问题考虑得尽可能周到，代码尽量写完整。下面是参考代码：

```
enum Status {kValid = 0, kInvalid};
int g_nStatus = kValid;

int StrToInt(const char* str)
{
    g_nStatus = kInvalid;
    long long num = 0;

    if(str != nullptr && *str != '\0')
    {
        bool minus = false;
        if(*str == '+')
            str++;
        else if(*str == '-')
        {
            str++;
            minus = true;
        }

        if(*str != '\0')
        {
            num = StrToIntCore(str, minus);
        }
    }

    return (int)num;
}

long long StrToIntCore(const char* digit, bool minus)
{
    long long num = 0;

    while(*digit != '\0')
    {
        if(*digit >= '0' && *digit <= '9')
        {
            int flag = minus ? -1 : 1;
            num = num * 10 + flag * (*digit - '0');

            if((!minus && num > 0x7FFFFFFF)
               || (minus && num < (signed int)0x80000000))
            {
                num = 0;
                break;
            }
        }
    }
}
```

```

        }

        digit++;
    }
else
{
    num = 0;
    break;
}
}

if(*digit == '\0')
{
    g_nStatus = kValid;
}

return num;
}

```

在前面的代码中，把空字符串""和只有一个正号或者负号的情况都考虑到了。同时最大的正整数值是 0x7FFF FFFF，最小的负整数是 0x8000 0000，因此，我们需要分两种情况来判断整数是否发生上溢出或者下溢出。

最后，他在提问环节给我留下的印象不是很好。他只有一个问题，是关于薪水方面的。是不是反映出他找工作仅仅关心工资？通常只关心工资待遇的员工是非常容易流失的，而且他把期望值设在 12000 元的唯一理由是他有同学的工资超过这个数。他对自己有没有一个定位？

虽然从他后来改写代码的过程来看，他的编程能力还是不错的，但我担心以他现在的编程习惯，由于没有做出全面的考虑，他写出的代码将会漏洞百出，鲁棒性也得不到保证。总的来说，我的意见是我们不能录取这名应聘者。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/67_StringToInt



测试用例：

- 功能测试（输入的字符串表示正数、负数和 0）。

- 边界值测试（最大的正整数；最小的负整数）。
- 特殊输入测试（输入的字符串为 nullptr 指针；输入的字符串为空字符串；输入的字符串中有非数字字符等）。

7.2

案例二：(面试题 68) 树中两个节点的最低公共祖先

面试官：前面两轮面试下来感觉怎么样？

应聘者：感觉还好，只是大脑连续转了两个小时，有点累。

面试官：面试是一个体力活，是挺累的。不过程序员这个行当本身也是体力活，没有好的身体还真撑不住。面试中也要看看你们的体力怎么样。

应聘者：（笑笑并点点头）说得有道理。

面试官：开个玩笑。现在可以开始面试了吧？

应聘者：好的，我准备好了。

面试官：请简要介绍一下你最近的一个项目。

应聘者：我最近完成的项目是 Civil 3D（一款基于 AutoCAD 的土木设计软件）中的 Multi-Target。这个 Target 指的是道路的边缘。之前道路的边缘只能是 Civil 中的一个数据类型，叫 Alignment，我的工作是让 Civil 支持其他类型的数据作为道路的边缘，如 AutoCAD 中的 Polyline 等。

面试官：有没有考虑到以后有可能添加新的数据类型作为道路的边缘？

应聘者：这在开发的过程中就发生过。在第二版的需求文档中添加了一种叫作 Pipeline 的道路边缘。由于我的设计中考虑了扩展性，最后只要添加新的 class 就行了，几乎不需要对已有的代码进行任何修改。

面试官：你是怎么做到的？

应聘者在白纸上用 UML 画了一张类型关系图（图略）。

应聘者：（指着图解释）从这张图中可以看出，一旦需要支持新的道路边缘，如 Pipeline，我们只需继承出新的 class 就可以了，对已有的其他 class 没有影响。



面试官心理：

对自己的工作讲得很细致、很深入，这个项目的确是你设计和实现的。可以看出，你对面向对象的设计和开发有着较深的理解。

面试官：（点点头）的确是这样的。接下来我们做一道编程题吧。我的题目是输入两个树节点，求它们的最低公共祖先。

应聘者：这棵树是不是二叉树？

面试官：是又怎么样，不是又怎么样？

应聘者：如果是二叉树，并且是二叉搜索树，那么是可以找到公共节点的。

面试官：那假设是二叉搜索树，你怎么查找呢？

应聘者：（有些激动，说得很快）二叉搜索树是排序过的，位于左子树的节点都比父节点小，而位于右子树的节点都比父节点大，我们只需要从树的根节点开始和两个输入的节点进行比较。如果当前节点的值比两个节点的值都大，那么最低的共同父节点一定在当前节点的左子树中，于是下一步遍历当前节点的左子节点。如果当前节点的值比两个节点的值都小，那么最低的共同父节点一定在当前节点的右子树中，于是下一步遍历当前节点的右子节点。这样，在树中从上到下找到的第一个在两个输入节点的值之间的节点就是最低的公共祖先。

面试官：看起来你对这道题目很熟悉，是不是以前做过啊？

应聘者：（面露尴尬）这个……碰巧……

面试官：（笑）那咱们把题目稍微换一下。如果这棵树不是二叉搜索树，甚至连二叉树都不是，而只是普通的树，又该怎么办呢？

应聘者：（停下来想了十几秒）树的节点中有没有指向父节点的指针？



面试官心理：

反应挺快的，而且提的问题针对性很强。你的沟通能力不错。

面试官：为什么需要指向父节点的指针？

应聘者在白纸上画了一张图，如图 7.1 所示。

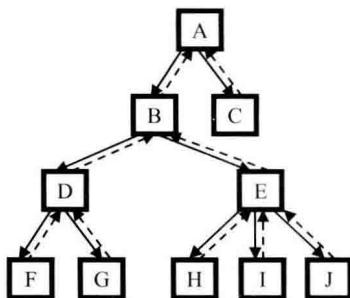


图 7.1 树中的节点有指向父节点的指针，用虚线箭头表示

应聘者：（指着自己画的图 7.1 解释）如果树中的每个节点（除根节点之外）都有一个指向父节点的指针，那么这个问题可以转换成求两个链表的第一个公共节点。假设树节点中指向父节点的指针是 pParent，那么从树的每个叶节点开始都有一个由指针 pParent 串起来的链表，这些链表的尾指针都是树的根节点。输入两个节点，那么这两个节点位于两个链表上，它们的最低公共祖先刚好就是这两个链表的第一个公共节点。比如输入的两个节点分别为 F 和 H，那么 F 在链表 F->D->B->A 上，而 H 在链表 H->E->B->A 上，这两个链表的第一个交点 B 刚好也是它们的最低公共祖先。

面试官：求两个链表的第一个共同节点这道题目你是不是之前也做过？

应聘者：（摸摸后脑勺，尴尬地笑笑）这个……又被您发现了……



面试官心理：

能够把这道题目转换成求两个链表的第一个公共节点，你的知识迁移能力不错。感觉你对数据结构很熟悉，基本上达到录用标准了。不过我很有兴趣看看你的极限在哪里。再加大点难度试试吧。

面试官：（笑）那只好再把题目的要求改变一下了。现在假设这棵树是普通的树，而且树中的节点没有指向父节点的指针。

应聘者：（稍微流露出一丝抓狂的表情，语气中透出失望）好吧，我再想想。

面试官：这道题目只比前面的两道稍微难一点点，你能搞定的。

应聘者：（静下来思考了两分钟）所谓两个节点的公共祖先，指的是这两个节点都出现在某个节点的子树中。我们可以从根节点开始遍历一棵树，每遍历到一个节点时，判断两个输入节点是不是在它的子树中。如果在子树中，则分别遍历它的所有子节点，并判断两个输入节点是不是在它们的子树中。这样从上到下一直找到的第一个节点，它自己的子树中同时包含两个输入的节点而它的子节点却没有，那么该节点就是最低的公共祖先。

面试官：能不能举个具体的例子说明你的思路？

应聘者：（一边在纸上画图 7.2 一边解释）假设还是输入节点 F 和 H。我们先判断 A 的子树中是否同时包含节点 F 和 H，得到的结果为 true。接着我们再先后判断 A 的两个子节点 B 和 C 的子树是不是同时包含 F 和 H，B 的结果是 true 而 C 的结果是 false。接下来我们再判断 B 的两个子节点 D 和 E，发现这两个节点得到的结果都是 false。于是 B 是最后一个公共祖先，即我们的输出。

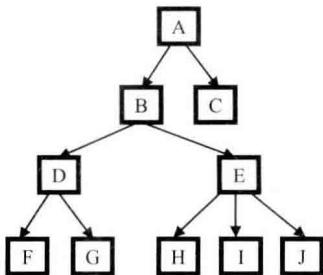


图 7.2 一棵普通的树，树中的节点没有指向父节点的指针

面试官：听起来不错。很明显，当我们判断以节点 A 为根的树中是否含有节点 F 的时候，我们需要对 D、E 等节点遍历一遍；接下来判断以节点 B 为根的树中是否含有节点 F 的时候，我们还是需要对 D、E 等节点再遍历一遍。这种思路会对同一节点重复遍历很多次。你想想看还有没有更快的算法？

应聘者：（双肘抵住桌子，双手抱住头顶，苦苦思索了两分钟）可以用辅助内存吗？

面试官：你需要多大的辅助内存？

应聘者：我的想法是用两个链表分别保存从根节点到输入的两个节点的路径，然后把问题转换成两个链表的最后公共节点。

面试官：（点头，面露赞许）嗯，具体说说。

应聘者：我们首先得到一条从根节点到树中某一节点的路径，这就要求在遍历的时候有一个辅助内存来保存路径。比如我们用前序遍历的方法来得到从根节点到H的路径的过程是这样的：(1) 遍历到A，把A存放到路径中，路径中只有一个节点A；(2) 遍历到B，把B存放到路径中，此时路径为A->B；(3) 遍历到D，把D存放到路径中，此时路径为A->B->D；(4) 遍历到F，把F存放到路径中，此时路径为A->B->D->F；(5) F已经没有子节点了，因此这条路径不可能到达节点H。把F从路径中删除，变成A->B->D；(6) 遍历G。和节点F一样，这条路径也不能到达H。遍历完G之后，路径仍然是A->B->D；(7) 由于D的所有子节点都遍历过了，不可能到达节点H，因此D不在从A到H的路径中，把D从路径中删除，变成A->B；(8) 遍历E，把E存放到路径中，此时路径变成A->B->E；(9) 遍历H，已经到达目标节点，A->B->E就是从根节点开始到达H必须经过的路径。

面试官：然后呢？

应聘者：同样，我们也可以得到从根节点开始到达F必须经过的路径是A->B->D。接着，我们求出这两条路径的最后一个公共节点，也就是B。B这个节点也是F和H的最低公共祖先。

面试官：这种思路的时间和空间效率是多少？

应聘者：为了得到从根节点开始到输入的两个节点的两条路径，需要遍历两次树，每遍历一次的时间复杂度是 $O(n)$ 。得到的两条路径的长度在最差情况时是 $O(n)$ ，通常情况下两条路径的长度是 $O(\log n)$ 。



面试官心理：

显然，你对数据结构的理解比大多数人要深刻得多，期待你的代码。

面试官：（微笑，点头）不错。根据这种思路写出C/C++代码，怎么样？

应聘者：好的，没问题。

应聘者先后下了3个函数：

```
bool GetNodePath(TreeNode* pRoot, TreeNode* pNode, list<TreeNode*>& path)
{
    if(pRoot == pNode)
        return true;
```

```
path.push_back(pRoot);

bool found = false;

vector<TreeNode*>::iterator i = pRoot->m_vChildren.begin();
while(!found && i < pRoot->m_vChildren.end())
{
    found = GetNodePath(*i, pNode, path);
    ++i;
}

if(!found)
    path.pop_back();

return found;
}

TreeNode* GetLastCommonNode
(
    const list<TreeNode*>& path1,
    const list<TreeNode*>& path2
)
{
    list<TreeNode*>::const_iterator iterator1 = path1.begin();
    list<TreeNode*>::const_iterator iterator2 = path2.begin();

    TreeNode* pLast = nullptr;

    while(iterator1 != path1.end() && iterator2 != path2.end())
    {
        if(*iterator1 == *iterator2)
            pLast = *iterator1;

        iterator1++;
        iterator2++;
    }

    return pLast;
}

TreeNode* GetLastCommonParent(TreeNode* pRoot, TreeNode* pNode1, TreeNode* pNode2)
{
    if(pRoot == nullptr || pNode1 == nullptr || pNode2 == nullptr)
        return nullptr;

    list<TreeNode*> path1;
    GetNodePath(pRoot, pNode1, path1);

    list<TreeNode*> path2;
    GetNodePath(pRoot, pNode2, path2);
```

```

        return GetLastCommonNode(path1, path2);
    }
}

```

应聘者：代码中 `GetNodePath` 用来得到从根节点 `pRoot` 开始到达节点 `pNode` 的路径，这条路径保存在 `path` 中。函数 `GetLastCommonNode` 用来得到两条路径 `path1` 和 `path2` 的最后一个公共节点。函数 `GetLastCommonParent` 先调用 `GetNodePath` 得到 `pRoot` 到达 `pNode1` 的路径 `path1`，再得到 `pRoot` 到达 `pNode2` 的路径 `path2`，接着调用 `GetLastCommonParent` 得到 `path1` 和 `path2` 的最后一个公共节点，即我们要找的最低公共祖先。

面试官：嗯，很好。这轮面试的时间已经很长了，我的问题就到这里。你有什么需要问我的吗？

应聘者：(略作思考)我想问问关于项目合作的事情。你们项目组和美国总部的同事是怎么合作的？是美国人做好设计，然后交给中国这边做具体实现吗？

面试官：理论上说中国同事与美国同事之间的合作是平等的，不全是美国说了算。我们中国的团队也有自己的项目经理做产品设计。现在两边的团队都有自己负责的功能。只是我们的团队成员和美国同事比起来，经验还不够，对产品的理解没有美国同事那么深刻。因此，当两边意见出现分歧的时候，他们的意见更能得到上层的重视。

应聘者：两边的团队都有哪些沟通的方式？

面试官：平时做相关工作的同事之间会有大量的 E-mail 交流。每周二的早上(美国时间是星期一的下午)我们有一个例会，所有同事都会参加。在会上大家会讨论项目的进度、遇到的困难等事项。另外，由于最近我们这边招了不少新员工，美国那边正计划选派一名资深的工程师过来给新员工进行培训。

应聘者：那中国这边的员工有机会去美国吗？

面试官：我们的人力资源部门有一个项目，让新员工在两年之内至少有机会去美国接受一次培训，以熟悉公司总部的文化。只是最近由于大的经济环境不是很好，公司在严格控制差旅费用，因此这个项目的执行受到了一点影响。还有其他问题吗？

应聘者：(想了一会儿)没有了。



面试官心理：

从最后几个问题可以看出，你对我们的项目和团队很有兴趣。同样，我也希望你能加入我们的团队一起做项目。这轮面试你通过了。

面试官：由于时间关系，这轮面试就到这里，怎么样？

应聘者：（摸摸额头，微笑中略显疲惫）谢谢。

面试官点评：

求树中两个节点的最低公共祖先，不能说只是一道题目，而应该说是一组题目，不同条件下的题目是完全不一样的。一开始的时候，我有意没有说明树的特点，比如树是不是二叉树、树中的节点是不是有一个指向父节点的指针。我把题目说得模棱两可是希望应聘者能够主动向我提出问题，一步一步弄清我的意图。如果一名应聘者能够在面试过程中主动问出高质量的问题以弄清楚题目的要求，那么我会觉得他态度积极，并且具有较强的沟通能力。

在这轮面试中，该应聘者表现得比较积极主动。一开始听到题目之后，他马上询问我树是不是二叉树。在我答复可以是二叉树之后，他立即给出了当树是二叉搜索树时的解法。我看出了他之前做过这个问题，于是就把题目的要求设为树只是普通的树而不一定是二叉树。他的反应很快，立即又问我树中的节点有没有指向父节点的指针。在第二个问题得到肯定的回答之后，他把问题转换成求两个链表的第一个公共节点。他这段的表现很好，问的两个问题都很有针对性，表明他对这种类型的问题有很深的理解，给我留下了很好的印象。

通常面试的时候让应聘者写出有指向父节点的指针这种情况的代码也就差不多了，但考虑到他之前做过类似的问题，同时我觉得他反应很快，功底不错，以他的能力应该可以挑战一下更高的难度。于是我接下来把指向父节点的指针去掉，决定再加大难度测试一下他的水平到底有多深。他再次表现出很快的反应能力，思考了一两分钟之后就想出了一种需要重复遍历一个节点多次的算法。在我提示出还有更快的算法之后，他再次把题目转换成求链表的共同节点的问题。在他解释其思路的过程中，可以看出他对树的遍历算法理解得很透彻，接下来写出的代码也很规范。综合这名应聘者在本轮面试中的表现，我强烈建议我们公司录用他。

如果面试官在面试的过程中逐步加大面试题的难度，那么通常对应聘者来说是一件好事，这说明应聘者一开始表现得很好，面试官对他的印象很好，并很有兴趣看看他的水平有多深，于是一步一步加大题目的难度。虽然最后应聘者可能不能很好地解决高难度的问题，但最终仍有可能拿到Offer。与此相反的是，有些应聘者在面试的时候很多问题都回答出来了，可最终被拒，觉得难以理解。其实这是因为面试官一开始问的问题他回答得很不好，面试官已经判断出他的能力有限，心里已经默默给出了NO的结论。但为了照顾应聘者的情面，也会问几个简单的问题。虽然这些问题应聘者可能都能答对，但前面的结果已经不会改变。

在这轮面试中，由于该应聘者一开始的表现很好，我才决定加大难度考考他。假如他对于普通树中节点没有指向父节点的指针这个问题没有很好地解决，那么我会让他回头去写普通树中节点有指向父节点的指针这个问题的代码。只要他的代码写得完整正确，我仍然会让他通过这轮面试，尽管我对他的评价可能没有现在这么高。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/68_CommonParentInTree



测试用例：

- 功能测试（普通形态的树；形状退化成链状的树）。
- 特殊输入测试（指向树根节点的指针为nullptr指针）。

面试官的视角

从面试官视角剖析考题构思、现场心理、题解优劣与面试心得，尚属首例。

80余道编程题

本书精选谷歌、微软等知名IT企业的80余道典型面试题，提供多角度的解题辅导。这些题目现今仍被大量面试官反复采用，实战参考价值颇高。

系统的解题方法

本书系统地总结了如何在面试时写出高质量代码，如何优化代码效率，以及分析、解决难题的常用方法。

超写实体验与感悟

Autodesk→微软→思科→美国微软总部，作者一路跳槽一路“面”，既亲历被考，也做过考官，更是资深程序员，大量的面试与编程经验，足当确保本书品质。

本书涉及程序源代码请到

<http://www.broadview.com.cn/31092> 进行下载。



上架建议：程序设计 / 求职面试



博文视点Broadview



@博文视点Broadview



策划编辑：张春雨

责任编辑：徐津平

封面设计：李玲

ISBN 978-7-121-31092-8



9 787121 310928 >

定价：65.00元