Chapter 3

一些用于热身的问题

在这一章我们将介绍一些常见的问题用于热身，在解决这些问题的过程中我们将先使用蛮力法来尝试如何解决这些问题，蛮力法是一种比较简单的方法，它通过一些枚举和搜索的手段把解空间的每个成员都检查一遍，最后找出问题的解，但是也是因为蛮力法需要检查解空间内的每一个成员，因此往往是比较慢的。因此我们将尝试看是否能够利用问题本身的一些特性来得到复杂度更优的解决方案。

3.1 2-Sum 问题

假设有一个从小到大排好顺序的整型数组，问对于一个给定的 sum，是否在数组中存在两个元素 x_1, x_2，使得 $x_1 + x_2 = sum$? 例如: 输入数组为 $[1, 2, 3, 6, 8, 9, 12]$，$sum = 14$，返回 $True$，因为 $2 + 12 = 14$，然而当 $sum = 6$ 时返回 $False$，因为不存在 x_1, x_2，使得 $x_1 + x_2 = 6$。

3.1.1 蛮力法

首先我们先尝试使用最朴素的蛮力法来解决这个问题。我们直接枚举所有的 (x_1, x_2) 数对，然后逐个检查是否存在 $x_1 + x_2 = sum$ 的情况。蛮力法的代码如 TWO-SUM-BRUTE-FORCE 所示。

```
TWO-SUM-BRUTE-FORCE(A, sum)
1: for i = 1 to A.length do
2:     for j = 1 to A.length do
3:         if i ≠ j and A[i] + A[j] == sum then
4:             return True
5: return False
```

蛮力法的正确性是非常好证明的，因为算法的第 1 行和第 2 行两重循环遍历了解

空间内的每一个元素，然后判断是否有解，因此算法是正确的。但是蛮力法的缺点是时间复杂度偏高，对于这个算法来说，两重循环的时间复杂度是 $\Theta(n^2)$。

3.1.2 尝试使用二分查找

蛮力法能够得到正确的解，但是速度稍微有些慢，原因就是没有利用好数组是排好顺序的这个特点。我们在之前介绍过二分查找算法，可以在对数阶的时间复杂度内，查找有序数组中是否存在给定的元素。

```
BINARY-SEARCH(A, l, r, target)
1: while l ≤ r do
2:     mid = (l + r)/2
3:     if A[mid] == target then
4:         return True
5:     else if A[mid] > target then
6:         r = mid − 1
7:     else if A[mid] < target then
8:         l = mid + 1
9: return False
```

```
TWO-SUM-BINARY-SEARCH(A, sum)
1: for i = 1 to A.length do
2:     if BINARY-SEARCH(A, i+1, A.length, sum - A[i]) then
3:         return True
4: return False
```

该算法是枚举 A 数组中的每一个元素作为 x_1，然后在右半部分的子数组中二分查找 $x_2 = sum - x_1$，若能够找到则说明存在 $x_1 + x_2 = sum$。算法的第 1 行循环遍历整个数组，算法的第 2 行调用 BINARY-SEARCH 的时间复杂度是 $\Theta(\lg n)$，因此算法总的时间复杂度为 $\Theta(n \lg n)$。该算法比蛮力法要快。

3.1.3 线性时间复杂度的算法

我们接下来将尝试寻找比利用二分查找算法更快的算法。我们考虑 TWO-SUM-LINEAR 所示的算法，该算法由于每个元素都只可能被遍历一次，因此时间复杂度是 $O(n)$。

```
TWO-SUM-LINEAR(A, sum)
```

```
1: l = 1, r = A.length
2: while l < r do
3:     if A[l] + A[r] == sum then
4:         return True
5:     else if A[l] + A[r] < sum then
6:         l = l + 1
7:     else if A[l] + A[r] > sum then
8:         r = r − 1
9: return False
```

接下来我们考虑证明该算法的正确性，这里我们只说考虑的方式，而不去严格的使用数学归纳法来证明它的正确性。算法维护的循环不变式是：x_1 和 x_2 只可能在数组下标 $[l, r]$ 之间。初始情况下我们将 l 设置为 1，r 设置为 $A.length$，也就是说解可能存在于整个数组当中。随着算法的运行，下标范围越来越小，直到 $l = r$ 时，数组中只存在一个元素，此时不可能找到两个元素相加和为 sum，故算法在此时返回 $False$。

算法一开始，我们选择数组中最小的元素和最大的元素作为 x_1 和 x_2，如果此时，$x_1 + x_2 < sum$，此时，说明 x_1 不能与数组中的任何元素相加得到 sum，也就是说，x_1 可能的值应该排除掉当前元素了，也就是把 l 往右移动一位。反过来，如果此时 $x_1 + x_2 > sum$，说明 x_2 不能与数组中的任何元素相加得到 sum，那么我们就要把当前 x_2 从可能的解中排除出去，也就是把 r 往左移动一位，然后循环这个过程就可以了。

总结一下，对于 2-Sum 问题，就是考虑一个排好顺序的数组，我们选择数组中最小的元素和最大的元素相加。相加的结果无非只有等于 $target$ 和不等于 $target$ 两种。如果等于 $target$，说明我们已经找到了解。如果不等于 $target$，那么说明这两个元素中，有一个元素一定不是解，如果大于 $target$，那么最大的元素一定不是解，否则最小的元素一定不是解。算法每次循环，要么就找到了解，要么就一定排除一个元素，如果所有元素都一定不是解，则说明没有解。

正如我们一开始分析的，该算法的时间复杂度是 $O(n)$，比利用二分查找的速度还快。

3.2 两个排好序的数组求交集

接下来我们思考这个问题：给定两个排好序的整型数组 A 和 B，返回一个数组 C，该数组中的每个数字同时存在于两个输入数组之中。

3.2.1 蛮力法

首先我们先考虑使用蛮力法来解决这个问题。对于每个 A 数组中的元素 $A[i]$，我们遍历 B 数组中的元素 $B[j]$，如果 $A[i] == B[j]$，那么说明该元素同时存在于两个数组中，

于是我们将该元素放入 C 数组之中。

该算法的时间复杂度是 $\Theta(n \times m)$，其中 n 是 A 数组的元素个数，m 是 B 数组的元素个数。

3.2.2 利用二分查找优化时间复杂度

该问题也可以利用二分查找来优化时间复杂度，我们对于每个 A 数组中的元素 $A[i]$，在 B 数组中二分查找 $A[i]$，若二分查找成功找到了 $A[i]$，则说明 $A[i]$ 同时存在于 A 数组和 B 数组之中，于是我们可以把它放进 C 数组之中。该算法的时间复杂度是 $\Theta(n \lg m)$，比蛮力法要快。

3.2.3 线性时间复杂度的的算法

我们接下来将介绍如 INTERSECTION-LINEAR 所示的算法，该算法将两个数组中的元素最多遍历一次，因此时间复杂度是 $O(n+m)$，比利用二分查找优化的时间复杂度 $\Theta(n \lg m)$ 更快。

INTERSECTION-LINEAR(A, B)

```
1: i = 1, j = 1
2: while i ≤ A.length and j ≤ B.length do
3:     if A[i] == B[j] then
4:         AddTo(C, A[i])
5:         i = i + 1
6:         j = j + 1
7:     else if A[i] > B[j] then
8:         j = j + 1
9:     else if A[i] < B[j] then
10:        i = i + 1
11: return C
```

接下来我们考虑算法的正确性。算法在初始条件下选择两个数组中的最小值，如果这两个值相等，那么说明该值同时存在于两个数组之中。否则，如果 $A[i] < B[j]$，说明无论如何 $A[i]$ 都不可能同时存在于两个数组之中，因为 B 中的每个元素都比它大，于是我们需要将 $A[i]$ 排除在可能的值之外，也就是把 i 往右移动一位。反过来呢，如果 $A[i] > B[j]$，说明无论如何 $B[j]$ 都不可能同时存在于两个数组之中，因为 A 中的每个元素都比它大，于是此时我们需要将 $B[j]$ 排除在可能的值之外，也就是把 j 往右移动一位。

3.3 二维有序数组寻找元素

假设有一个维度为 $m \times n$ 的二维整型数组 A，数组中的每一行元素从左到右依次增大，同时数组中的每一列元素从上到下也依次增大。同时给你一个整数 *target*，请问数组中是否存在 *target*?

3.3.1 蛮力法

该问题的蛮力法解决方案就是遍历数组中的每个元素然后检查该元素是不是等于 *target*，如果找到了就返回 $True$，如果找不到就返回 $False$。该算法的时间复杂度是 $O(n \times m)$。

3.3.2 线性时间复杂度的算法

接下来我们考虑一个线性时间复杂度的算法 MATRIX-SEARCH。

```
MATRIX-SEARCH(A, target)
1: i = 1, j = n
2: while i ≤ m and j ≥ 1 do
3:     if A[i][j] == target then
4:         return True
5:     else if A[i][j] > target then
6:         j = j − 1
7:     else if A[i][j] < target then
8:         i = i + 1
9: return False
```

首先我们考虑算法的时间复杂度，i 最多遍历完所有行，j 最多遍历完所有列，因此时间复杂度为 $O(n + m)$，比蛮力法要快。

接下来我们考虑算法的正确性，算法维持的循环不变式是: *target* 只可能在范围 $(1, m)$ 到 (i, j) 确定的矩形范围内。初始条件下，(i, j) 标定的范围是整个二维数组。此时若 $target = A[i][j]$，那么我们找到了该元素。否则如果 $target < A[i][j]$，说明整个 i 行中都不可能存在 *target*，于是我们就需要将这一行排除出去，也就是将 i 往下移动一位。反过来我们考虑 $target > A[i][j]$，说明整个 j 列中都不可能存在 *target*，于是我们需要将这一列排除出去，也就是将 j 往左移动一位。算法每一步都将范围逐个变小，直到整个范围中没有元素，这时候算法应该返回 $False$。

总结一下，这个问题与 2-Sum 问题是类似的。算法的每一轮迭代，只有两种情况，要么指针指向的元素是 *target*，要么不是。如果是 *target* 我们就找到了该元素。如果不

是 *target*，说明一定有一行或者一定有一列里面没有 *target*，也就是说，算法的每次迭代如果没找到 *target*，就一定要排除一行或者一列。如果所有元素都被排除了，就说明这个数组里一定不存在 *target*。

3.4 全排列的生成

我们考虑这样一个问题：给定一个数组 $A[1..n]$，数组中的每个元素都不相同，且 $1 \le A[i] \le 9$。现在我们选择数组中的若干个元素，我们用这些整数构造一个整数 a，然后用数组中的剩余元素构造一个整数 b，求 $|a-b|$ 的最小值。

例如 $A = [1,2,3,4,5,6]$，我们有 $a = 412$，$b = 365$，此时 $|a-b| = 47$。除此之外任意解都比 47 要大。

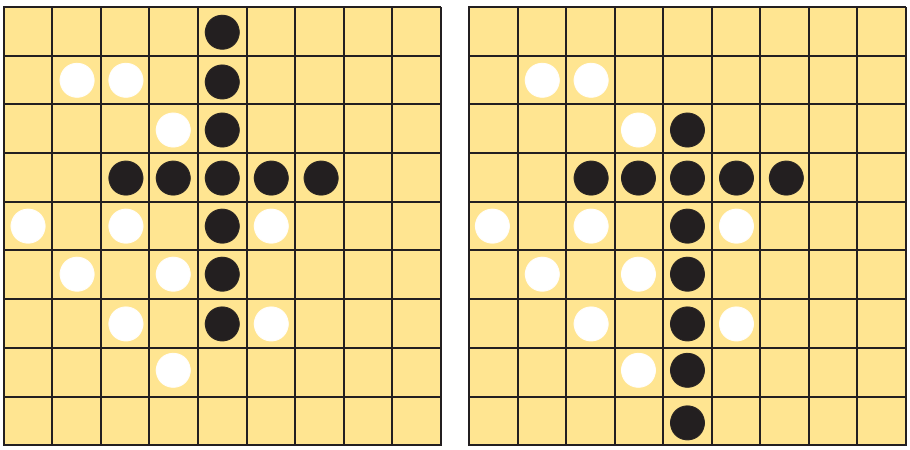
我们接下来考虑蛮力法：我们枚举 A 数组的全排列，然后再枚举 A 数组中 a 的位数并构造 a，然后剩下的数字就是 b，最后求解 $|a-b|$ 并记录最小值。

那么问题就变成了如何枚举 A 数组的全排列呢？我们可以考虑使用如 PERMUTATE 所示的算法，我们需要一个全局变量 $VIS[1..n]$，并在初始的时候将 VIS 中的每个元素都设置为 $False$，当我们调用 PERMUTATE(A, 1) 时，算法将生成 A 的每个排列，并放入 $PERMUTATION$ 数组中，并调用 Handle 函数来处理这个排列。对于这个问题，Handle 函数是构造可能的 a 和 b，并求解最小的 $|a-b|$。

```
PERMUTATE(A, i)
 1: if i == A.length + 1 then
 2:     Handle(PERMUTATION)
 3:     return
 4: for j = 1 to A.length do
 5:     if not VIS[j] then
 6:         VIS[j] = True
 7:         PERMUTATION[i] = A[j]
 8:         PERMUTATE(A, i + 1)
 9:         VIS[j] = False
```

接下来我们考虑证明算法的正确性，算法将逐个选择当前未被选择的数字放入到 $PERMUTATION[i]$ 上，于是当填满 $PERMUTATION$ 时，得到的是 A 数组的一个排列。

接下来我们考虑算法的时间复杂度，算法将生成 A 数组的全排列，因为数组 A 的全排列个数是 $n!$ 个，因此算法的时间复杂度是 $\Theta(n!)$。



(a) 合法的局面　　(b) 不合法的局面

Figure 3.1: 五子棋局面

3.5 五子棋的局面合法性判断

在一个 $n \times n$ 的棋盘上，黑棋和白棋轮流落子，黑棋先下，当有一方有横着、竖着、或者斜着连成 5 个或以上时游戏结束，最后落子的一方获胜，我们在这个问题中不考虑禁手。现在给定一个五子棋的局面，请问该局面是否是一个合法的局面？合法的含义是指能够从空白棋盘开始，黑白轮流落子能够到达的局面。

如图3.1(a)所示，是一种合法的局面，因为我们可以将最后一步棋视作黑棋交叉的那一个。然而如图3.1(b)所示的局面是非法的，因为我们无论如何都无法从游戏开始，黑白轮流落子下到这一步。

我们可以使用如下的蛮力法来解这个题目:

1. 对黑棋和白棋分别计数，当黑棋和白棋数量相等时，游戏的最后一步一定是白棋，当黑棋比白棋数量多一个时，游戏的最后一步一定是黑棋，除此之外，棋盘的局面是不合法的。

2. 若最后一步棋是黑棋，那么如果此时白色的有五个或者五个以上的连珠时，局面一定不合法。反过来，如果最后一步是白棋，但是有黑色的五连珠时，局面一定不合法。

3. 如果黑棋和白棋都不存在五连珠时，局面一定是合法的。

4. 否则，一定存在最后一步棋有五连珠，不妨假设最后一步棋是黑棋，我们枚举每一个黑棋判断它可不可能是最后一步棋，如果所有黑棋都不可能是最后一步棋，那么局面不合法，否则局面是合法的。判断一个棋可不可能是最后一步棋的方法是:去掉这步棋后，棋盘不存在五连珠。

接下来我们计算上面的这个算法的时间复杂度: 第 1 步数棋子, 时间复杂度是 $O(n^2)$, 第 2 步判断五连珠, 时间复杂度是 $O(n^2)$, 第 3 步可以在第 2 步进行的过程中顺便完成, 因此不花时间, 第 4 步需要枚举每个黑色棋子, 用 $O(n^2)$ 的时间, 去掉这个棋子后还要再看是否有五连珠, 又是 $O(n^2)$ 的时间, 因此总共是 $O(n^4)$ 的时间复杂度。

我们接下来考虑如何优化上面的时间复杂度。此时我们考虑既然黑棋至少存在一个五连珠, 那么我们只需要判断这五连珠中是否存在最后一步棋即可, 否则如果其它黑棋是最后一步棋, 一定存在刚才判断过的五连珠, 这样一来我们最多只需要枚举 5 个棋子来判断它们是不是最后一步棋, 此时时间复杂度是 $O(5 \times n^2) = O(n^2)$。