20241217-Week15 编程练习

Updated 1256 GMT+8 Dec 17, 2024

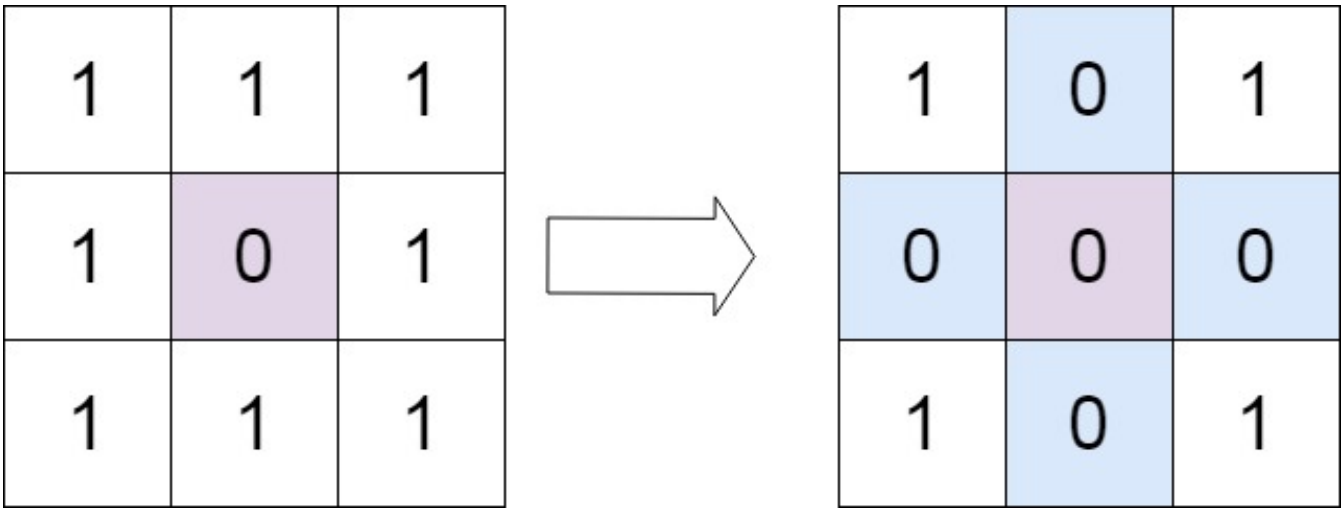
2024 fall, Complied by Hongfei Yan

73.矩阵置零

https://leetcode.cn/problems/set-matrix-zeroes/

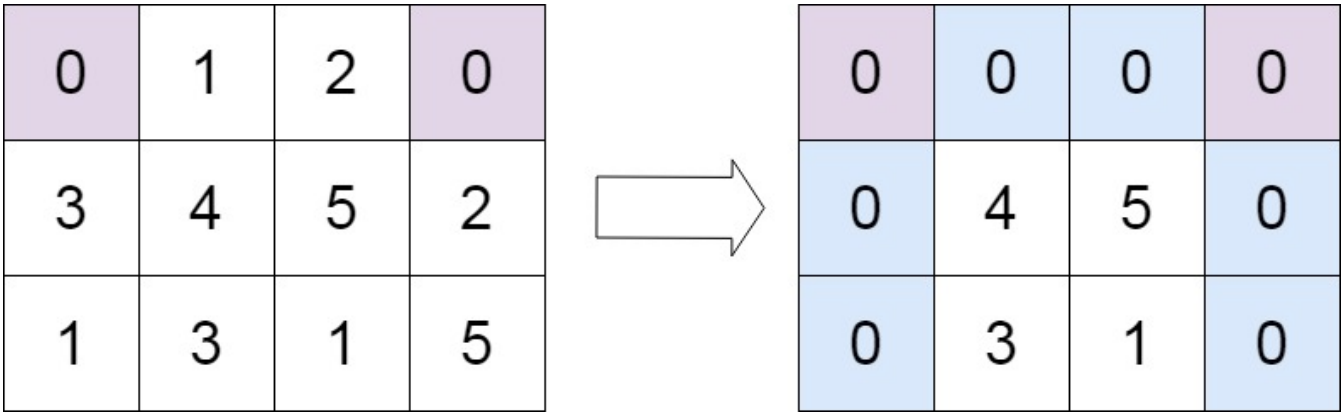
给定一个 `*m* x *n*` 的矩阵，如果一个元素为 **0** ，则将其所在行和列的所有元素都设为 **0** 。请使用 [原地](#) 算法。

示例 1：




```
输入：matrix = [[1,1,1],[1,0,1],[1,1,1]]
输出：[[1,0,1],[0,0,0],[1,0,1]]
```

示例 2：



```
输入：matrix = [[0,1,2,0],[3,4,5,2],[1,3,1,5]]
输出：[[0,0,0,0],[0,4,5,0],[0,3,1,0]]
```

提示：

- `m == matrix.length`
- `n == matrix[0].length`
- `1 <= m, n <= 200`
- `-231 <= matrix[i][j] <= 231 - 1`

进阶：

- 一个直观的解决方案是使用 `O(*m**n*)` 的额外空间，但这并不是一个好的解决方案。
- 一个简单的改进方案是使用 `O(*m* + *n*)` 的额外空间，但这仍然不是最好的解决方案。
- 你能想出一个仅使用常量空间的解决方案吗?

```python
class Solution:
    def setZeroes(self, matrix: List[List[int]]) -> None:
        """
        Do not return anything, modify matrix in-place instead.
        """
        m = len(matrix); n = len(matrix[0])
        visited = set()
        for i in range(m):
            for j in range(n):
                if (i,j) not in visited and matrix[i][j] == 0:
                    for r in range(m):
                        if matrix[r][j] == 0:
                            continue
                        matrix[r][j] = 0
                        visited.add((r,j))
                    for c in range(n):
                        if matrix[i][c] == 0:
                            continue
                        matrix[i][c] = 0
                        visited.add((i,c))
                    visited.add((i,j))

        return matrix

```

74.搜索二维矩阵

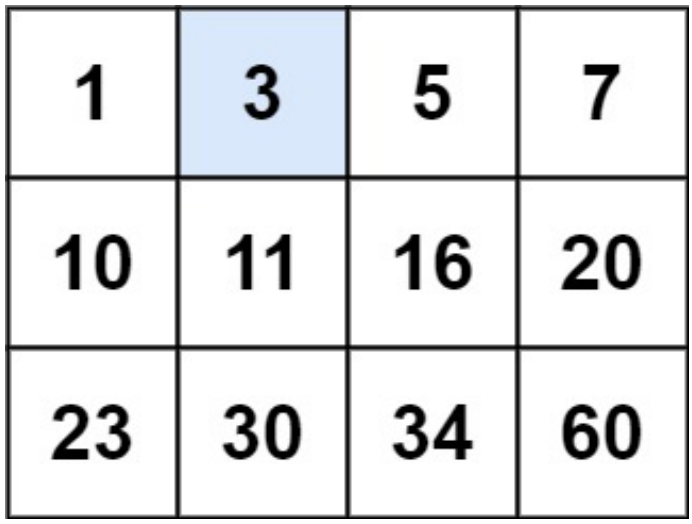
binary search, https://leetcode.cn/problems/search-a-2d-matrix/

给你一个满足下述两条属性的 `m x n` 整数矩阵：

- 每行中的整数从左到右按非严格递增顺序排列。
- 每行的第一个整数大于前一行的最后一个整数。

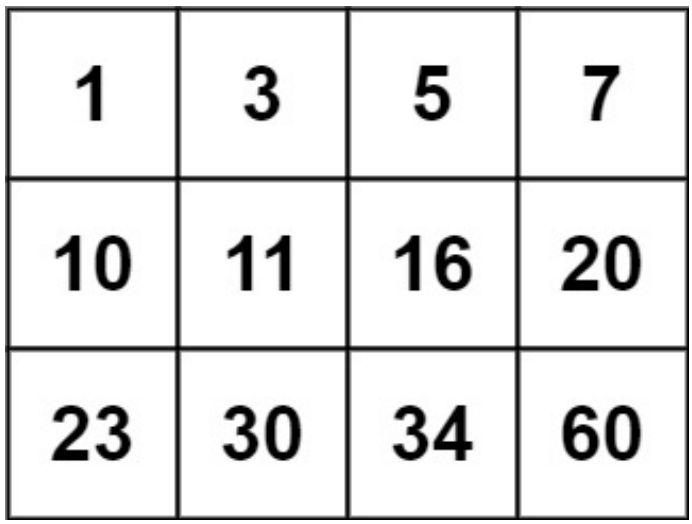
给你一个整数 `target` ，如果 `target` 在矩阵中，返回 `true` ；否则，返回 `false` 。

示例 1：




```
输入：matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 3
输出：true
```

示例 2：



```
输入：matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 13
输出：false
```

提示：

- `m == matrix.length`
- `n == matrix[i].length`
- `1 <= m, n <= 100`
- `-104 <= matrix[i][j], target <= 104`

```python
class Solution:
    def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:
        a = []
        for row in matrix:
            a.extend(row)

        if target == a[0]:
            return True

        flag = False
        lo = 0; hi = len(a)
        while lo < hi:
            mid = (lo + hi) // 2
            if a[mid] < target:
                lo = mid + 1
            elif a[mid] > target:
                hi = mid
            else:
                flag = True
                break

        return flag
```

03468: 电池的寿命

greedy, http://cs101.openjudge.cn/practice/03468/

小S新买了一个掌上游戏机，这个游戏机由两节5号电池供电。为了保证能够长时间玩游戏，他买了很多5号电池，这些电池的生产商不同，质量也有差异，因而使用寿命也有所不同，有的能使用5个小时，有的可能就只能使用3个小时。显然如果他只有两个电池一个能用5小时一个能用3小时，那么他只能玩3个小时的游戏，有一个电池剩下的电量无法使用，但是如果他有更多的电池，就可以更加充分地利用它们，比如他有三个电池分别能用3、3、5小时，他可以先使用两节能用3个小时的电池，使用半个小时后再把其中一个换成能使用5个小时的电池，两个半小时后再把剩下的一节电池换成刚才换下的电池（那个电池还能用2.5个小时），这样总共就可以使用5.5个小时，没有一点浪费。

现在已知电池的数量和电池能够使用的时间，请你找一种方案使得使用时间尽可能的长。

输入

输入包含多组数据。每组数据包括两行，第一行是一个整数N ($2 \le N \le 1000$)，表示电池的数目，接下来一行是N个正整数表示电池能使用的时间。

输出

对每组数据输出一行，表示电池能使用的时间，保留到小数点后1位。

样例输入

```
2
3 5
3
3 3 5
```

样例输出

```
3.0
5.5
```

通过分析可以发现，理想的最优结果是所有电池总电量的一半。一个很明显达不到理想结果的例子是当寿命最长的电池寿命高于其他所有电池寿命之和时，此时最优结果是其他所有电池寿命之和。猜测其他情况下，理想的最优结果是成立的，即:
(1)情况一：寿命最长的电池寿命高于其他所有电池寿命之和，此时答案是其他所有电池寿命之和;
(2)情况二：寿命最长的电池寿命小于或等于其他所有电池寿命之和，此时猜测答案就是总的电量/2。

```python
while True:
    try:
        n=int(input())
        l=list(map(int,input().split()))
        l.sort()
        maxi=l.pop()
        sumi=sum(l)
        if sumi<maxi:
            num=sumi
        else:
            num=(sumi+maxi)/2
        print(f"{num:.1f}")
```

```
    except:
        break
```

698.划分为k个相等的子集

dfs, https://leetcode.cn/problems/partition-to-k-equal-sum-subsets/

给定一个整数数组 `nums` 和一个正整数 `k`，找出是否有可能把这个数组分成 `k` 个非空子集，其总和都相等。

示例 1：

```
输入： nums = [4, 3, 2, 3, 5, 2, 1], k = 4
输出： True
说明： 有可能将其分成 4 个子集（5），（1,4），（2,3），（2,3）等于总和。
```

示例 2:

```
输入: nums = [1,2,3,4], k = 3
输出: false
```

提示：

- `1 <= k <= len(nums) <= 16`
- `0 < nums[i] < 10000`
- 每个元素的频率在 `[1,4]` 范围内

```python
from functools import lru_cache
from typing import List

class Solution:
    def canPartitionKSubsets(self, nums: List[int], k: int) -> bool:
        total = sum(nums)
        if total % k != 0:
            return False

        target = total // k
        nums.sort(reverse=True)

        # Early exit if the largest number is greater than the target
```

```
        if nums[0] > target:
            return False

        # The 'su' array keeps track of the sum of each subset
        su = [0] * k

        @lru_cache(maxsize=None)
        def recursion(nums_tuple, su_tuple):
            nums = list(nums_tuple)
            su = list(su_tuple)

            if not nums:
                return all(s == target for s in su)

            current_num = nums[0]
            for i in range(k):
                if su[i] + current_num <= target:
                    su[i] += current_num
                    if recursion(tuple(nums[1:]), tuple(su)):
                        return True
                    su[i] -= current_num

                # If the current subset is empty and adding the number doesn't
work, break
                if su[i] == 0:
                    break

            return False

        return recursion(tuple(nums), tuple(su))
```

Explanation of Changes:

1. **Tuple for `nums` and `su`**: We maintain the inputs to the recursive function as tuples, which are hashable and can be used in `lru_cache`. This ensures that the recursion doesn't recompute already visited states.
2. **Early exit**: The condition `if nums[0] > target` ensures that if the largest number is greater than the target sum, we immediately return `False` (this avoids unnecessary computation).
3. **Optimized backtracking**: We exit the loop early if `su[i] == 0` to avoid repeating attempts to place the same number in already empty positions in `su`.

41.缺失的第一个正数

标记，置换，https://leetcode.cn/problems/first-missing-positive/

给你一个未排序的整数数组 `nums` ，请你找出其中没有出现的最小的正整数。

请你实现时间复杂度为 `O(n)` 并且只使用常数级别额外空间的解决方案。

示例 1：

```
输入：nums = [1,2,0]
输出：3
解释：范围 [1,2] 中的数字都在数组中。
```

示例 2：

```
输入：nums = [3,4,-1,1]
输出：2
解释：1 在数组中，但 2 没有。
```

示例 3：

```
输入：nums = [7,8,9,11,12]
输出：1
解释：最小的正数 1 没有出现。
```

提示：

- `1 <= nums.length <= 105`
- `-231 <= nums[i] <= 231 - 1`

这个解法，使用了集合，不满足题面“使用常数级别额外空间”。

```python
class Solution:
    def firstMissingPositive(self, nums: List[int]) -> int:
        n = len(nums)
        dp = [-1]*(n+1)
        minv = 1
        visited = set()
        for i in nums:
            if i < 1:
                continue
            visited.add(i)
            if minv == i:
                tmp = minv + 1
                while tmp in visited:
                    tmp += 1
                minv = tmp
```

```
        return minv
```

LeetCode题解：

https://leetcode.cn/problems/first-missing-positive/solutions/304743/que-shi-de-di-yi-ge-zheng-shu-by-leetcode-solution/

实际上，对于一个长度为 N 的数组，其中没有出现的最小正整数只能在 [1,N+1] 中。这是因为如果 [1,N] 都出现了，那么答案是 N+1，否则答案是 [1,N] 中没有出现的最小正整数。这样一来，我们将所有在 [1,N] 范围内的数放入哈希表，也可以得到最终的答案。而给定的数组恰好长度为 N，这让我们有了一种将数组设计成哈希表的思路：

我们对数组进行遍历，对于遍历到的数 x，如果它在 [1,N] 的范围内，那么就将数组中的第 x−1 个位置（注意：数组下标从 0 开始）打上「标记」。在遍历结束之后，如果所有的位置都被打上了标记，那么答案是 N+1，否则答案是最小的没有打上标记的位置加 1。

那么如何设计这个「标记」呢？由于数组中的数没有任何限制，因此这并不是一件容易的事情。但我们可以继续利用上面的提到的性质：由于我们只在意 [1,N] 中的数，因此我们可以先对数组进行遍历，把不在 [1,N] 范围内的数修改成任意一个大于 N 的数（例如 N+1）。这样一来，数组中的所有数就都是正数了，因此我们就可以将「标记」表示为「负号」。算法的流程如下：

我们将数组中所有小于等于 0 的数修改为 N+1；

我们遍历数组中的每一个数 x，它可能已经被打了标记，因此原本对应的数为 $|x|$，其中 $|$ 为绝对值符号。如果 $|x|\in[1,N]$，那么我们给数组中的第 $|x|-1$ 个位置的数添加一个负号。注意如果它已经有负号，不需要重复添加；

在遍历完成之后，如果数组中的每一个数都是负数，那么答案是 N+1，否则答案是第一个正数的位置加 1。

```python
class Solution:
    def firstMissingPositive(self, nums: List[int]) -> int:
        n = len(nums)
        for i in range(n):
            if nums[i] <= 0:
                nums[i] = n + 1

        for i in range(n):
            num = abs(nums[i])
            if num <= n:
                nums[num - 1] = -abs(nums[num - 1])

        for i in range(n):
            if nums[i] > 0:
                return i + 1

        return n + 1
```

复杂度分析

时间复杂度：O(N)，其中 N 是数组的长度。

空间复杂度：O(1)。

方法二：置换

除了打标记以外，我们还可以使用置换的方法，将给定的数组「恢复」成下面的形式：

如果数组中包含 $x \in [1, N]$，那么恢复后，数组的第 x−1 个元素为 x。

在恢复后，数组应当有 [1, 2, ..., N] 的形式，但其中有若干个位置上的数是错误的，每一个错误的位置就代表了一个缺失的正数。以题目中的示例二 [3, 4, -1, 1] 为例，恢复后的数组应当为 [1, -1, 3, 4]，我们就可以知道缺失的数为 2。

那么我们如何将数组进行恢复呢？我们可以对数组进行一次遍历，对于遍历到的数 x=nums[i]，如果 x∈[1,N]，我们就知道 x 应当出现在数组中的 x−1 的位置，因此交换 nums[i] 和 nums[x−1]，这样 x 就出现在了正确的位置。在完成交换后，新的 nums[i] 可能还在 [1,N] 的范围内，我们需要继续进行交换操作，直到 $x \notin [1, N]$。

注意到上面的方法可能会陷入死循环。如果 nums[i] 恰好与 nums[x−1] 相等，那么就会无限交换下去。此时我们有 nums[i]=x=nums[x−1]，说明 x 已经出现在了正确的位置。因此我们可以跳出循环，开始遍历下一个数。

由于每次的交换操作都会使得某一个数交换到正确的位置，因此交换的次数最多为 N，整个方法的时间复杂度为 O(N)。

```python
class Solution:
    def firstMissingPositive(self, nums: List[int]) -> int:
        n = len(nums)
        for i in range(n):
            while 1 <= nums[i] <= n and nums[nums[i] - 1] != nums[i]:
                nums[nums[i] - 1], nums[i] = nums[i], nums[nums[i] - 1]
        for i in range(n):
            if nums[i] != i + 1:
                return i + 1
        return n + 1
```

复杂度分析

- 时间复杂度：*O*(*N*)，其中 *N* 是数组的长度。
- 空间复杂度：*O*(1)。

994.腐烂的橘子

bfs, https://leetcode.cn/problems/rotting-oranges/

在给定的 `m x n` 网格 `grid` 中，每个单元格可以有以下三个值之一：

- 值 `0` 代表空单元格；
- 值 `1` 代表新鲜橘子；
- 值 `2` 代表腐烂的橘子。

每分钟，腐烂的橘子 **周围 4 个方向上相邻** 的新鲜橘子都会腐烂。

返回 *直到单元格中没有新鲜橘子为止所必须经过的最小分钟数。如果不可能，返回* `-1` 。

示例 1：

```
输入：grid = [[2,1,1],[1,1,0],[0,1,1]]
输出：4
```

示例 2：

```
输入：grid = [[2,1,1],[0,1,1],[1,0,1]]
输出：-1
解释：左下角的橘子（第 2 行， 第 0 列）永远不会腐烂，因为腐烂只会发生在 4 个方向上。
```

示例 3：

```
输入：grid = [[0,2]]
输出：0
解释：因为 0 分钟时已经没有新鲜橘子了，所以答案就是 0 。
```

提示：

- `m == grid.length`
- `n == grid[i].length`
- `1 <= m, n <= 10`
- `grid[i][j]` 仅为 `0`、`1` 或 `2`

```python
from typing import List
from collections import deque

class Solution:
    def orangesRotting(self, grid: List[List[int]]) -> int:
        rows, cols = len(grid), len(grid[0])
        queue = deque()
        fresh_oranges = 0

```

```
        # 初始化队列和统计新鲜橘子
        for r in range(rows):
            for c in range(cols):
                if grid[r][c] == 2:
                    queue.append((r, c, 0))  # (row, col, minutes)
                elif grid[r][c] == 1:
                    fresh_oranges += 1

        # 定义4个方向
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
        minutes = 0

        # 开始BFS
        while queue:
            r, c, minutes = queue.popleft()
            for dr, dc in directions:
                nr, nc = r + dr, c + dc
                if 0 <= nr < rows and 0 <= nc < cols and grid[nr][nc] == 1:
                    grid[nr][nc] = 2  # 腐烂新鲜橘子
                    fresh_oranges -= 1
                    queue.append((nr, nc, minutes + 1))

        # 如果还有新鲜橘子，返回-1；否则返回分钟数
        return -1 if fresh_oranges > 0 else minutes
```

37.解数独

backtracking, https://leetcode.cn/problems/sudoku-solver/

编写一个程序，通过填充空格来解决数独问题。

数独的解法需 **遵循如下规则**：

1. 数字 `1-9` 在每一行只能出现一次。
2. 数字 `1-9` 在每一列只能出现一次。
3. 数字 `1-9` 在每一个以粗实线分隔的 `3x3` 宫内只能出现一次。（请参考示例图）

数独部分空格内已填入了数字，空白格用 `'.'` 表示。

示例 1：

```
输入：board = [["5","3",".",".","7",".",".",".","."],
["6",".",".","1","9","5",".",".","."],[".","9","8",".",".",".",".",".","6","."],
["8",".",".",".","6",".",".",".","3"],["4",".",".","8",".","3",".",".","1"],
["7",".",".",".","2",".",".",".","6"],[".","6",".",".",".",".","2","8","."],
[".",".",".","4","1","9",".",".","5"],[".",".",".",".","8",".",".","7","9"]]
输出：[["5","3","4","6","7","8","9","1","2"],["6","7","2","1","9","5","3","4","8"],
["1","9","8","3","4","2","5","6","7"],["8","5","9","7","6","1","4","2","3"],
["4","2","6","8","5","3","7","9","1"],["7","1","3","9","2","4","8","5","6"],
["9","6","1","5","3","7","2","8","4"],["2","8","7","4","1","9","6","3","5"],
["3","4","5","2","8","6","1","7","9"]]
解释：输入的数独如上图所示，唯一有效的解决方案如下所示：
```

提示：

- `board.length == 9`
- `board[i].length == 9`
- `board[i][j]` 是一位数字或者 `'.'`
- 题目数据 **保证** 输入数独仅有一个解

这个代码超时了，如何优化

```python
class Solution(object):
    def solveSudoku(self, board):
        """
        :type board: List[List[str]]
        :rtype: void Do not return anything, modify board in-place instead.
        """
        def is_valid(row, col, val, board):
            for i in range(9):
                if board[row][i] == val or board[i][col] == val:
                    return False
            start_row, start_col = row - row % 3, col - col % 3
            for i in range(3):
                for j in range(3):
                    if board[start_row + i][start_col + j] == val:
                        return False
            return True

        def backtrack(board):
            for i in range(9):
                for j in range(9):
                    if board[i][j] == '.':
                        for k in range(1, 10):
                            val = str(k)
                            if is_valid(i, j, val, board):
                                board[i][j] = val
                                if backtrack(board):
```

```
                                    return True
                                board[i][j] = '.'
                        return False
            return True

        backtrack(board)
```

你提供的代码是一个典型的数独解法，使用了回溯法。对于超时问题，可以从以下几个方向进行优化：

1. 提前检查列、行和3x3宫内的合法性

每次调用 `is_valid` 时，都会遍历整个行、列和3x3宫，检查是否合法。这是导致超时的主要原因之一。我们可以通过缓存这些信息来减少冗余计算。

2. 使用集合来存储已使用的数字

我们可以使用三个集合来分别跟踪每行、每列和每个3x3宫内的已用数字，从而避免每次都进行完全的遍历。

3. 选择最优的空格进行回溯

在回溯过程中，可以尝试优先填充空格数量最少的地方（即空格周围已填数字最少的地方）。这将减少回溯的次数。

代码优化：

```python
class Solution(object):
    def solveSudoku(self, board):
        """
        :type board: List[List[str]]
        :rtype: void Do not return anything, modify board in-place instead.
        """

        # 记录行、列、3x3宫中已经使用的数字
        rows = [set() for _ in range(9)]
        cols = [set() for _ in range(9)]
        boxes = [set() for _ in range(9)]

        # 初始化已经填入的数字
        for i in range(9):
            for j in range(9):
                if board[i][j] != '.':
                    val = board[i][j]
                    rows[i].add(val)
                    cols[j].add(val)
                    boxes[(i // 3) * 3 + (j // 3)].add(val)

        # 回溯函数
        def backtrack(i, j):
            if i == 9:  # 如果行遍历完成，说明已解答完毕
                return True
```

```
            if j == 9:  # 如果列遍历完成，跳到下一行
                return backtrack(i + 1, 0)
            if board[i][j] != '.':  # 已经填充过的格子，跳过
                return backtrack(i, j + 1)

            # 尝试填充每个数字 1-9
            for num in range(1, 10):
                num_str = str(num)
                box_idx = (i // 3) * 3 + (j // 3)
                if num_str not in rows[i] and num_str not in cols[j] and num_str
not in boxes[box_idx]:
                    # 填充数字
                    board[i][j] = num_str
                    rows[i].add(num_str)
                    cols[j].add(num_str)
                    boxes[box_idx].add(num_str)

                    # 继续回溯
                    if backtrack(i, j + 1):
                        return True

                    # 回溯时撤销选择
                    board[i][j] = '.'
                    rows[i].remove(num_str)
                    cols[j].remove(num_str)
                    boxes[box_idx].remove(num_str)

            return False

        # 从(0,0)开始回溯
        backtrack(0, 0)
```

关键优化点：

1. **缓存行、列和宫内的数字**： 使用三个 `set` 列表（`rows`, `cols`, `boxes`）来缓存每个行、列和宫中已经填入的数字，避免在每次回溯时重新遍历。
2. **减少 `is_valid` 的调用**： 通过使用 `set` 来检查数字是否已经在当前行、列或宫内，这样比遍历整个行、列和宫要高效得多。
3. **回溯的条件优化**：
    - 每次处理一个格子时，只有在格子为空 (`.`) 时才会进行数字填充和回溯。
    - 用行、列和宫的限制条件来提前剪枝，避免不必要的回溯。

时间复杂度：

通过使用 `set` 来进行快速查询，减少了每次合法性检查的时间复杂度。整体的时间复杂度接近 `O(9^(N^2))`，但由于回溯过程中有效的剪枝和预处理，实际表现会更快。

Kadane's algorithm

Kadane's 算法是一种用于解决最大子数组问题（Maximum Subarray Problem）的高效算法，即在一个一维整数数组中找到具有最大和的连续子数组。该算法由 Jay Kadane 于 1984 年提出，并以其线性时间复杂度 O(n) 而闻名，其中 n 是输入数组的长度。

算法原理

Kadane's 算法的核心思想是遍历整个数组，并在每一步都计算到当前位置为止的最大子数组和。它通过维护两个变量来实现这一点：

- **`max_current`**：表示以当前元素结尾的最大子数组和。
- **`max_global`**：表示迄今为止找到的最大子数组和。

随着遍历的进行，`max_current` 会根据当前元素更新，如果 `max_current` 变得比 `max_global` 大，则更新 `max_global`。

算法步骤

1. **初始化**：
   - 将 `max_current` 和 `max_global` 初始化为数组的第一个元素。
2. **遍历数组**：
   - 从第二个元素开始遍历数组。
   - 对于每个元素，更新 `max_current` 为 `max_current + 当前元素` 和 `当前元素` 中较大的那个值。这相当于决定是否将当前元素加入到现有的子数组中，还是重新开始一个新的子数组。
   - 如果 `max_current` 大于 `max_global`，则更新 `max_global` 为 `max_current` 的值。
3. **返回结果**：
   - 最终，`max_global` 包含的就是最大子数组的和。

示例代码 (Python)

```python
def max_subarray_sum(arr):
    if not arr:
        return 0

    max_current = max_global = arr[0]

    for num in arr[1:]:
        max_current = max(num, max_current + num)
        if max_current > max_global:
            max_global = max_current

    return max_global

# 测试用例
arr = [-2, 1, -3, 4, -1, 2, 1, -5, 4]
print("最大子数组和为:", max_subarray_sum(arr))  # 输出：最大子数组和为：6
```

处理负数数组

对于全部元素都是负数的情况，上述算法仍然适用。在这种情况下，`max_global` 将等于数组中的最大单个元素，因为任何包含多个元素的子数组的和都会更小。

扩展应用

Kadane's 算法不仅可以用于寻找最大子数组和，还可以稍加修改来解决其他相关问题，例如：

- **最小子数组和**：只需改变比较逻辑，使 `max` 操作变为 `min` 操作即可。
- **带约束条件的最大子数组和**：例如要求子数组至少包含 k 个元素，可以通过调整算法逻辑或使用滑动窗口等方法来满足特定需求。

总之，Kadane's 算法是一个非常优雅且高效的解决方案，适用于各种涉及连续子数组和的问题。如果你有更多关于此算法的具体问题或需要进一步的帮助，请随时提问！

M20744: 土豪购物，这是 Kadane's Algorithm 变形啊。

E22548: 机智的股民老张，感觉也是变形。

示例02766: 最大子矩阵

dp, http://cs101.openjudge.cn/practice/02766/

已知矩阵的大小定义为矩阵中所有元素的和。给定一个矩阵，你的任务是找到最大的非空(大小至少是1 * 1)子矩阵。

比如，如下4 * 4的矩阵

0 -2 -7 0
9 2 -6 2
-4 1 -4 1
-1 8 0 -2

的最大子矩阵是

9 2
-4 1
-1 8

这个子矩阵的大小是15。

输入

输入是一个N * N的矩阵。输入的第一行给出N (0 < N <= 100)。再后面的若干行中，依次（首先从左到右给出第一行的N个整数，再从左到右给出第二行的N个整数……）给出矩阵中的N2个整数，整数之间由空白字符分隔（空格或者空行）。已知矩阵中整数的范围都在[-127, 127]。

输出

输出最大子矩阵的大小。

样例输入

```
4
0 -2 -7 0 9 2 -6 2
-4 1 -4  1 -1

8  0 -2
```

样例输出

```
15
```

来源：翻译自 Greater New York 2001 的试题

```python
'''
为了找到最大的非空子矩阵，可以使用动态规划中的Kadane算法进行扩展来处理二维矩阵。
基本思路是将二维问题转化为一维问题：可以计算出从第i行到第j行的列的累计和，
这样就得到了一个一维数组。然后对这个一维数组应用Kadane算法，找到最大的子数组和。
通过遍历所有可能的行组合，我们可以找到最大的子矩阵。
'''
def max_submatrix(matrix):
    def kadane(arr):
        # max_ending_here 用于追踪到当前元素为止包含当前元素的最大子数组和。
        # max_so_far 用于存储迄今为止遇到的最大子数组和。
        max_end_here = max_so_far = arr[0]
        for x in arr[1:]:
            # 对于每个新元素，我们决定是开始一个新的子数组（仅包含当前元素 x），
            # 还是将当前元素添加到现有的子数组中。这一步是 Kadane 算法的核心。
            max_end_here = max(x, max_end_here + x)
            max_so_far = max(max_so_far, max_end_here)
        return max_so_far

    rows = len(matrix)
    cols = len(matrix[0])
    max_sum = float('-inf')

    for left in range(cols):
        temp = [0] * rows
        for right in range(left, cols):
            for row in range(rows):
                temp[row] += matrix[row][right]
            max_sum = max(max_sum, kadane(temp))
    return max_sum

n = int(input())
nums = []

while len(nums) < n * n:
    nums.extend(input().split())
matrix = [list(map(int, nums[i * n:(i + 1) * n])) for i in range(n)]
```

```
37
38 max_sum = max_submatrix(matrix)
39 print(max_sum)
```