Addition question solution:

### 题目分析

这道题的核心是通过贪心算法找到一种调度方式，使所有参赛者完成三项全能项目的总时间最短。每个参赛者有预计的游泳时间、骑行时间和跑步时间。参赛者必须依次出发，一个接一个地进入泳池。

### 解题思路

1. \*\*定义问题\*\*：

- 每个参赛者都有三个时间：游泳时间 \( S \)、骑行时间 \( C \)、跑步时间 \( R \)。

- 参赛者必须按顺序出发，一个完成游泳后，另一个开始。

2. \*\*目标\*\*：

- 使所有参赛者完成比赛的总时间最短。

3. \*\*贪心策略\*\*：

- 关键在于决定谁先开始，谁后开始。直觉上，先让游泳时间长的参赛者开始，这样可以减少他们对后续参赛者的等待时间。

### 贪心算法描述

1. \*\*计算每个参赛者的总非游泳时间\*\*：即骑行时间 \( C \) + 跑步时间 \( R \)。

2. \*\*排序\*\*：按照总非游泳时间从大到小对参赛者进行排序。

3. \*\*按顺序安排参赛者\*\*：按照排序后的顺序安排参赛者进入泳池。

### 算法实现

```python

def triathlon\_schedule(contestants):

# contestants 是一个列表，元素是 (S, C, R)

# 计算总非游泳时间，并排序

contestants.sort(key=lambda x: -(x[1] + x[2]))

current\_time = 0

max\_completion\_time = 0

for S, C, R in contestants:

current\_time += S

completion\_time = current\_time + C + R

if completion\_time > max\_completion\_time:

max\_completion\_time = completion\_time

return max\_completion\_time

# 示例输入

contestants = [

(30, 60, 20),

(20, 50, 30),

(50, 20, 40)

]

# 计算最小完成时间

print(triathlon\_schedule(contestants))

```

### 正确性证明

1. \*\*贪心选择性质\*\*：

- 我们的选择是先安排总非游泳时间长的参赛者，假设我们有两个参赛者 \( A \) 和 \( B \)，如果 \( A \) 的总非游泳时间比 \( B \) 的总非游泳时间长，那么先安排 \( A \) 可以减少 \( B \) 的等待时间，从而减少总完成时间。

2. \*\*归纳证明\*\*：

- 基本情况：只有一个参赛者时，显然选择顺序无关紧要。

- 归纳假设：对于 \( k \) 个参赛者，贪心选择是最优的。

- 归纳步骤：对于 \( k+1 \) 个参赛者，假设前 \( k \) 个已经按贪心选择排列，第 \( k+1 \) 个按照总非游泳时间从大到小排序，确保最小化总完成时间。

### 时间复杂度分析

- \*\*排序\*\*：排序的时间复杂度为 \( O(n \log n) \)。

- \*\*遍历计算\*\*：遍历所有参赛者计算完成时间的时间复杂度为 \( O(n) \)。

- \*\*总时间复杂度\*\*：总时间复杂度为 \( O(n \log n) \)。

通过以上步骤，我们能够有效地安排参赛者的顺序，使得比赛总完成时间最短。

2.

### 题目分析

我们需要为每个灯塔计算它能通过纸飞机到达的最左边的灯塔。我们需要找到一个算法，使得算法的时间复杂度为 \(O(n)\)。

### 解题思路

我们可以使用单调栈（Monotonic Stack）来解决这个问题。单调栈是一种非常适合处理类似于 "Next Greater Element" 或者 "Previous Greater Element" 问题的数据结构。

1. \*\*单调栈的定义\*\*：

- 单调递减栈：栈中的元素从栈底到栈顶是递减的（在这里，考虑灯塔的高度）。

2. \*\*算法描述\*\*：

- 使用单调递减栈来存储灯塔的索引，保持栈中的高度是递减的。

- 遍历灯塔数组，对于每一个灯塔，检查栈顶的灯塔高度：

- 如果当前灯塔高度小于栈顶灯塔高度，弹出栈顶元素，继续检查新的栈顶。

- 如果当前灯塔高度不小于栈顶灯塔高度，将当前灯塔的索引入栈。

- 这样，当我们处理完一个灯塔后，栈顶的灯塔就是最左边的一个可以到达的灯塔。

### 算法实现

```python

def find\_leftmost\_reachable\_lighthouses(heights):

n = len(heights)

result = [-1] \* n # 初始化结果数组

stack = []

for i in range(n):

# 检查栈顶的灯塔高度是否大于等于当前灯塔高度

while stack and heights[stack[-1]] <= heights[i]:

stack.pop()

# 栈顶的灯塔就是最左边的一个可以到达的灯塔

if stack:

result[i] = stack[-1]

# 将当前灯塔索引入栈

stack.append(i)

return result

# 示例输入

heights = [4, 2, 7, 1, 3, 5]

# 计算每个灯塔可以到达的最左边的灯塔

print(find\_leftmost\_reachable\_lighthouses(heights))

```

### 证明其正确性

1. \*\*单调性维护\*\*：

- 栈中的灯塔高度是单调递减的，这保证了我们总是能找到当前灯塔能到达的最左边的灯塔。

- 当我们遇到一个新的灯塔高度时，如果栈顶的灯塔高度小于当前灯塔高度，这意味着栈顶的灯塔不能再被当前灯塔"遮挡"，所以可以安全地移除。

2. \*\*覆盖所有情况\*\*：

- 对于每一个灯塔 \(i\)，我们都检查了它之前所有的灯塔，通过单调栈结构保证了我们找到的是最左边的灯塔。

### 时间复杂度分析

- 每个灯塔最多只会被压栈和出栈一次，所以时间复杂度为 \(O(n)\)。

- 空间复杂度也是 \(O(n)\)，用于存储栈和结果数组。

因此，该算法满足题目的时间复杂度要求，能够在 \(O(n)\) 时间内解决问题。



### 题目分析

我们需要将一个 \( n \times n \) 的棋盘（其中 \( n \) 是 2 的幂）使用 L 形状的瓷砖铺满，其中缺少一个任意选择的方格。每个 L 形瓷砖由 3 个方格组成。我们需要设计一个算法来完成铺设，并且时间复杂度为 \( O(n^2) \)。

### 解决思路

1. \*\*分治法\*\*：我们可以利用递归来解决这个问题。将棋盘划分为 4 个 \( \frac{n}{2} \times \frac{n}{2} \) 的子棋盘。然后根据缺失的方格所在的子棋盘，将剩余的 3 个子棋盘的中心放置一个 L 形瓷砖，使得它们看起来像是缺失了一格（即一个递归子问题）。

2. \*\*归纳证明\*\*：通过归纳证明，可以证明这种铺法是可行的。

### 算法描述

1. \*\*基础情况\*\*：如果 \( n = 2 \)，那么直接用一个 L 形瓷砖覆盖 3 个格子，留出缺失的那个格子即可。

2. \*\*递归步骤\*\*：

- 将棋盘划分为 4 个 \( \frac{n}{2} \times \frac{n}{2} \) 的子棋盘。

- 找到缺失的那个格子在哪个子棋盘中。

- 在剩余的 3 个子棋盘的中心放置一个 L 形瓷砖。

- 递归地处理每个子棋盘。

### 算法实现

以下是算法的 Python 实现：

```python

def place\_tromino(board, n, r, c, tr, tc):

"""

使用 L 形瓷砖铺设 n x n 棋盘, board 是棋盘，r 和 c 是缺失的方格的位置

tr 和 tc 是当前子棋盘的左上角坐标

"""

if n == 2:

# 直接铺设 2x2 的棋盘

tile = 1

for i in range(2):

for j in range(2):

if not (tr + i == r and tc + j == c):

board[tr + i][tc + j] = tile

return

# 当前 L 形瓷砖的编号

global tile

tile += 1

t = tile

# 棋盘的中心

mid = n // 2

# 识别缺失方格在的子棋盘

quadrant = (r >= tr + mid) \* 2 + (c >= tc + mid)

# 放置中心的 L 形瓷砖

for i in range(2):

for j in range(2):

if (i \* 2 + j) != quadrant:

board[tr + mid - 1 + i][tc + mid - 1 + j] = t

# 递归处理 4 个子棋盘

place\_tromino(board, mid, r if quadrant == 0 else tr + mid - 1, c if quadrant == 0 else tc + mid - 1, tr, tc)

place\_tromino(board, mid, r if quadrant == 1 else tr + mid - 1, c if quadrant == 1 else tc + mid, tr, tc + mid)

place\_tromino(board, mid, r if quadrant == 2 else tr + mid, c if quadrant == 2 else tc + mid - 1, tr + mid, tc)

place\_tromino(board, mid, r if quadrant == 3 else tr + mid, c if quadrant == 3 else tc + mid, tr + mid, tc + mid)

def solve\_tiling\_problem(n, missing\_r, missing\_c):

board = [[0 for \_ in range(n)] for \_ in range(n)]

place\_tromino(board, n, missing\_r, missing\_c, 0, 0)

return board

# 初始化瓷砖编号

tile = 0

# 示例

n = 4

missing\_r, missing\_c = 1, 1

board = solve\_tiling\_problem(n, missing\_r, missing\_c)

# 打印棋盘

for row in board:

print(row)

```

### 证明其正确性

通过递归拆分棋盘并用 L 形瓷砖覆盖中心位置的策略，可以保证每次递归调用中总能正确地覆盖非缺失方格部分。每次递归调用都会处理一个更小的棋盘，最终归结到基础情况 \( n = 2 \)，这个情况显然是可以用一个 L 形瓷砖覆盖的。

### 时间复杂度分析

每次递归调用会处理 4 个大小为 \( \frac{n}{2} \times \frac{n}{2} \) 的子棋盘，递归的深度是 \( \log n \)。每层递归中，我们对棋盘进行常数次的操作，所以总时间复杂度是 \( O(n^2) \)。



### 题目解析

给定一个算法 `POWER(a, n)` 计算 \( a^n \)，需要完成以下任务：

(a) 证明该算法的正确性。

(b) 提供递推关系。

(c) 使用主定理（Master Theorem）给出时间复杂度。

(d) 使用展开法（unrolling）给出时间复杂度。

### (a) 证明算法的正确性

算法的基本思路是通过递归和分治法来计算 \( a^n \)：

1. 如果 \( n = 1 \)，直接返回 \( a \)，这显然是正确的。

2. 如果 \( n \) 是偶数，则 \( a^n = (a^{n/2})^2 \)。这一步递归计算 \( a^{n/2} \)，然后返回其平方。

3. 如果 \( n \) 是奇数，则 \( a^n = a \times a^{(n-1)/2} \times a^{(n-1)/2} \)。这一步递归计算 \( a^{(n-1)/2} \)，然后返回 \( a \) 乘以该值的平方。

通过归纳法证明：

- \*\*基础情况\*\*：当 \( n = 1 \) 时，算法返回 \( a \)，这显然是正确的。

- \*\*归纳假设\*\*：假设算法对 \( n \leq k \) 都是正确的。

- \*\*归纳步骤\*\*：当 \( n = k + 1 \) 时，如果 \( n \) 是偶数，则 \( a^n = (a^{n/2})^2 \)，根据归纳假设，递归调用 `POWER(a, n/2)` 是正确的，因此 \( a^n \) 的结果也正确。如果 \( n \) 是奇数，则 \( a^n = a \times a^{(n-1)/2} \times a^{(n-1)/2} \)，根据归纳假设，递归调用 `POWER(a, (n-1)/2)` 是正确的，因此 \( a^n \) 的结果也正确。

综上所述，算法是正确的。

### (b) 提供递推关系

从代码可以看出，每次递归调用将问题规模减半，因此递推关系为：

\[ T(n) = T\left(\frac{n}{2}\right) + O(1) \]

### (c) 使用主定理（Master Theorem）给出时间复杂度

根据主定理，递推关系 \( T(n) = T\left(\frac{n}{2}\right) + O(1) \) 中，符合 \( a = 1, b = 2, f(n) = O(1) \) 的形式：

- \( a = 1 \)

- \( b = 2 \)

- \( f(n) = O(1) \)

根据主定理的三种情况，我们需要比较 \( f(n) \) 与 \( n^{\log\_b a} \)：

- \( \log\_b a = \log\_2 1 = 0 \)

- \( f(n) = O(1) \)

主定理的第一种情况适用，即如果 \( f(n) = O(n^c) \) 且 \( c < \log\_b a \)，则 \( T(n) = O(n^{\log\_b a}) \)。这里 \( c = 0 \) 且 \( \log\_b a = 0 \)，因此 \( T(n) = O(1) \)。

所以，递归的时间复杂度是 \( O(\log n) \)。

### (d) 使用展开法（unrolling）给出时间复杂度

展开递归过程：

\[ T(n) = T\left(\frac{n}{2}\right) + O(1) \]

\[ T\left(\frac{n}{2}\right) = T\left(\frac{n}{4}\right) + O(1) \]

\[ T\left(\frac{n}{4}\right) = T\left(\frac{n}{8}\right) + O(1) \]

...

一直展开到 \( T(1) \)：

\[ T(n) = T\left(\frac{n}{2^k}\right) + k \cdot O(1) \]

当 \( \frac{n}{2^k} = 1 \) 时， \( k = \log n \)，则

\[ T(n) = T(1) + \log n \cdot O(1) = O(1) + O(\log n) = O(\log n) \]

综上所述，使用展开法（unrolling）得到的时间复杂度也是 \( O(\log n) \)。

所以，无论使用哪种方法分析，该算法的时间复杂度都是 \( O(\log n) \)。



### 题目解析

给定一个简单多边形 \( P \)，要求提供一个 \( O(n) \) 时间复杂度的算法来计算其凸包 \( S \)。凸包是所有内角小于180度的形状。输入是一个按逆时针方向排列的边列表，每个边都有一个起点和终点。

### 解题思路

可以使用 Graham 扫描法来解决这个问题。以下是具体步骤：

1. \*\*找到最低点\*\*：在所有点中找到 y 坐标最小的点（如果有多个点具有相同的 y 坐标，则选择 x 坐标最小的那个）。这一步在 \( O(n) \) 时间内完成。

2. \*\*按极角排序\*\*：将所有点按相对于最低点的极角排序。可以通过计算极角来进行排序，这一步可以在 \( O(n \log n) \) 时间内完成，但由于题目提示我们可以在 \( O(1) \) 时间内计算两条线段之间的角度，我们可以在 \( O(n) \) 时间内完成。

3. \*\*构建凸包\*\*：使用一个栈来保存构建凸包的点。对于每个点，检查栈顶的两个点是否与当前点形成一个“左转”，如果是则将当前点入栈，否则将栈顶点出栈。这一步在 \( O(n) \) 时间内完成。

### 算法描述

```python

def compute\_convex\_hull(points):

# Step 1: Find the point with the lowest y-coordinate (and leftmost if tie)

lowest\_point = min(points, key=lambda p: (p[1], p[0]))

# Step 2: Sort the points based on the angle with the lowest point

def angle\_with\_lowest(point):

return (point[1] - lowest\_point[1]) / (point[0] - lowest\_point[0] + 1e-9)

sorted\_points = sorted(points, key=angle\_with\_lowest)

# Step 3: Construct the convex hull using a stack

hull = []

for point in sorted\_points:

while len(hull) > 1 and not is\_left\_turn(hull[-2], hull[-1], point):

hull.pop()

hull.append(point)

return hull

def is\_left\_turn(p1, p2, p3):

# Calculate cross product to determine the turn direction

return (p2[0] - p1[0]) \* (p3[1] - p2[1]) - (p2[1] - p1[1]) \* (p3[0] - p2[0]) > 0

```

### 正确性证明

1. \*\*找到最低点\*\*：这是所有凸包算法的标准步骤，确保从最低点开始构建凸包可以使得算法更简单。

2. \*\*按极角排序\*\*：按极角排序保证了所有点都按逆时针顺序排列，这对于构建凸包是必要的。

3. \*\*构建凸包\*\*：使用栈来检查是否形成左转，如果不是则回退，这确保了构建的多边形的每个内角都小于180度，因此是凸包。

### 时间复杂度分析

1. \*\*找到最低点\*\*：\( O(n) \)

2. \*\*按极角排序\*\*：由于题目提示可以在 \( O(1) \) 时间内计算角度，可以在 \( O(n) \) 时间内完成排序。

3. \*\*构建凸包\*\*：每个点最多入栈一次，出栈一次，因此 \( O(n) \)。

综上所述，该算法的总时间复杂度为 \( O(n) \)。