实验 8 报告 -202408040228-符航康

一、问题分析

1. 分析

- **处理的对象（数据）**：
  - 珊瑚礁的列数 `n`（整数）。
  - 每列珊瑚的高度 `a_i`（一个整数数组或列表）。
  - 允许使用的最大水量 `x`（长整数，因为水量可能很大）。
  - 水族箱的候选高度 `h`（长整数，因为高度可能与珊瑚高度或水量相关）。
  - 计算得出的给定高度 `h` 时所需的水量 `w`（长整数）。
- **采用的数据结构**：
  - 对于表示各列珊瑚高度的 `a_i`，代码中使用了 `vector<long long> a`（C++）。这是一个动态数组。
  - **原因**：
    - **动态大小**：`vector` 可以根据输入的 `n` 动态调整大小，方便存储未知数量的珊瑚高度。
    - **快速访问**：可以通过索引在 O（1） 时间内访问任意一列珊瑚的高度，这对于计算所需水量非常重要。
    - **顺序存储**：珊瑚的列是按顺序排列的，`vector` 的顺序存储特性与此相符。
    - **类型安全**：`long long` 类型确保了即使珊瑚高度或计算的水量很大时也不会发生溢出。
- **其他数据**：
  - `n`（珊瑚列数）和 `x`（最大水量）使用 `int` 和 `long long` 类型的简单变量存储。
  - `h`（水族箱高度）在二分搜索中作为 `long long` 类型的变量（如 `mid`，`low`，`high`，`ans`）处理。

2. 功能

- **功能 1：检查给定高度的可行性 （`can_build` 函数）**
  - 输入：一个候选的水族箱高度 `h_candidate`，珊瑚列数 `n`，最大可用水量 `max_water`，以及各列珊瑚高度的集合 `corals`。
  - 处理：计算如果水族箱高度为 `h_candidate` 时，需要注入多少水。对于每一列珊瑚，如果其高度 `corals[i]` 小于 `h_candidate`，则需要的水量增加 `h_candidate - corals[i]`。
  - 输出：一个布尔值。如果总需水量不超过 `max_water`，则返回 `true`（可行）；否则返回 `false`（不可行）。

- **功能 2：求解最大可行高度 （`solve` 函数）**
  - 输入：珊瑚列数 `n`，最大可用水量 `x`，以及各列珊瑚高度。
  - 处理：
    i. 确定一个搜索范围，用于寻找可能的最大高度 `h`。最小高度为 1，最大高度可以是一个足够大的值（例如，题目中给定的 `x` 加上最高的珊瑚高度，或者一个固定的上限如 `2*10^9 + 100`，因为 `a_i` 和 `x` 的上限分别是 `10^9`）。
    ii. 使用二分搜索算法在这个范围内查找。
    iii. 对于二分搜索选定的每个中间高度 `mid`，调用 `can_build` 函数检查其可行性。
    iv. 如果 `mid` 可行，说明可能存在更高的高度，因此将 `mid` 记录为当前最优解，并尝试在 `[mid+1, high]` 范围内搜索。
    v. 如果 `mid` 不可行，说明 `mid` 太高了，需要在 `[low, mid-1]` 范围内搜索。
  - 输出：满足条件的最大水族箱高度 `h`。
- **功能 3：处理多个测试用例**
  - 输入：测试用例的数量 `t`。
  - 处理：循环 `t` 次，每次都调用 `solve` 函数解决一个独立的测试用例。

3. 分析并确定处理后的结果如何显示

- 对于每个测试用例，程序将输出一个正整数，该整数表示在不超过可用水量 `x` 的前提下，可以建造的水族箱的最大高度 `h`。
- 每个测试用例的输出结果占一行。

4. 请用题目中样例，详细给出样例求解过程

以第一个测试用例为例：

- `n = 7`, `x = 9`
- 珊瑚高度 `a = [3, 1, 2, 4, 6, 2, 5]`

求解过程 （`solve` 函数）：

1. **初始化搜索范围**：
   - `low = 1`
   - `high` 设为一个足够大的数，例如 `2 * 10^9 + 100` （代码中使用 `2000000000LL + 100LL`）。
   - `ans = 0` （用于存储找到的最大可行高度）
2. **二分搜索**：
   - **迭代 1**：
     - `mid = low + (high - low) / 2` （例如，`mid` 约等于 `10^9`）
     - 调用 `can_build(mid, 7, 9, a)`。显然，如果高度为 `10^9`，所需水量会远超 `9`。
     - `can_build` 返回 `false`。
     - `high = mid - 1`.

- ... (经过多次迭代，搜索范围会迅速缩小) ...
- **假设当前搜索范围缩小到 `low = 1`, `high = 10`**
- **迭代 (例如 `low = 1, high = 10`)：**
  - `mid = 1 + (10 - 1) / 2 = 5`
  - 调用 `can_build(5, 7, 9, a)`：
    - 列 1 (高度 3)：水量 `5 - 3 = 2`
    - 列 2 (高度 1)：水量 `5 - 1 = 4` (总水量 `2+4=6`)
    - 列 3 (高度 2)：水量 `5 - 2 = 3` (总水量 `6+3=9`)
    - 列 4 (高度 4)：水量 `5 - 4 = 1` (总水量 `9+1=10`)
    - 此时 `water_needed = 10 > max_water = 9`。
    - `can_build` 返回 `false`。
  - `high = mid - 1 = 4`.
- **迭代 (`low = 1, high = 4`)：**
  - `mid = 1 + (4 - 1) / 2 = 2`
  - 调用 `can_build(2, 7, 9, a)`：
    - 列 1 (高度 3)：水量 `0` (因为 `3 >= 2`)
    - 列 2 (高度 1)：水量 `2 - 1 = 1`
    - 列 3 (高度 2)：水量 `0` (因为 `2 >= 2`)
    - 列 4 (高度 4)：水量 `0`
    - 列 5 (高度 6)：水量 `0`
    - 列 6 (高度 2)：水量 `0`
    - 列 7 (高度 5)：水量 `0`
    - 总水量 `water_needed = 1`。 `1 <= 9`.
    - `can_build` 返回 `true`。
  - `ans = mid = 2`.
  - `low = mid + 1 = 3`.
- **迭代 (`low = 3, high = 4`)：**
  - `mid = 3 + (4 - 3) / 2 = 3`
  - 调用 `can_build(3, 7, 9, a)`：
    - 列 1 (高度 3)：水量 `0`
    - 列 2 (高度 1)：水量 `3 - 1 = 2`
    - 列 3 (高度 2)：水量 `3 - 2 = 1` (总水量 `2+1=3`)
    - 列 4 (高度 4)：水量 `0`
    - 列 5 (高度 6)：水量 `0`

- 列 6 （高度 2）：水量 `3 - 2 = 1` （总水量 `3+1=4`）
- 列 7 （高度 5）：水量 `0`
- 总水量 `water_needed = 4`。 `4 <= 9`.
- `can_build` 返回 `true`。

- `ans = mid = 3`.
- `low = mid + 1 = 4`.

- **迭代 （`low = 4, high = 4`）：**
  - `mid = 4 + (4 - 4) / 2 = 4`
  - 调用 `can_build(4, 7, 9, a)`：
    - 列 1 （高度 3）：水量 `4 - 3 = 1`
    - 列 2 （高度 1）：水量 `4 - 1 = 3` （总水量 `1+3=4`）
    - 列 3 （高度 2）：水量 `4 - 2 = 2` （总水量 `4+2=6`）
    - 列 4 （高度 4）：水量 `0`
    - 列 5 （高度 6）：水量 `0`
    - 列 6 （高度 2）：水量 `4 - 2 = 2` （总水量 `6+2=8`）
    - 列 7 （高度 5）：水量 `0`
    - 总水量 `water_needed = 8`。 `8 <= 9`.
    - `can_build` 返回 `true`。
  - `ans = mid = 4`.
  - `low = mid + 1 = 5`.
- **迭代 （`low = 5, high = 4`）：**
  - `low > high`，循环终止。

3. **输出结果**：
   - 输出 `ans`，即 `4`。

与样例输出一致。

二、数据结构和算法设计

1. 抽象数据类型设计 （ADT）

可以定义以下抽象数据类型：

代码块

```
ADT AquariumProblemSolver {
    数据对象:
        corals: 珊瑚高度的序列 （例如，整数列表）
        max_water_limit: 可用的最大水量 （例如，长整数）
        num_corals: 珊瑚的列数 （例如，整数）
```

```
    操作:
        // 构造函数/初始化方法
        constructor(heights: 序列, water_limit: 长整数, count: 整数)
        // 检查给定高度是否可行
        // 输入: candidate_height (长整数) - 拟议的水族箱高度
        // 输出: 布尔值 - 如果所需水量不超过 max_water_limit 则为 true, 否则为 false
        // 异常: 如果 candidate_height < 1, 可抛出无效参数异常 (或按题目要求处理, 题目要
        is_height_feasible(candidate_height: 长整数) -> 布尔值
        // 查找最大可行高度
        // 输入: 无 (使用已存储的数据对象)
        // 输出: 长整数 - 最大可行水族箱高度
        // 约束: 搜索范围的下界为1, 上界需合理设定
        find_max_feasible_height() -> 长整数
}
```

2. 物理数据对象设计

- **珊瑚高度 (`a` 或 `corals`)**:
  - 在 C++ 中，使用 `vector<long long> a;`。
  - 这是一个动态数组，元素类型为 `long long`，用于存储每列珊瑚的高度。`long long` 确保高度值较大时不会溢出。
- **珊瑚列数 (`n`)**:
  - 在 C++ 中，使用 `int n;`。
  - 一个整数变量，存储珊瑚礁的列数。
- **最大可用水量 (`x` 或 `max_water`)**:
  - 在 C++ 中，使用 `long long x;`。
  - 一个长整型变量，存储允许使用的最大水量。
- **二分搜索辅助变量**:
  - `low`: `long long low;` 表示当前搜索范围的下界。
  - `high`: `long long high;` 表示当前搜索范围的上界。
  - `mid`: `long long mid;` 表示当前搜索范围的中点。
  - `ans`: `long long ans;` 存储当前找到的最优解（最大可行高度）。

这些物理数据对象直接映射到 C++ 代码中使用的变量。基本操作如访问 `vector` 元素 (`a[i]`)、算术运算、比较等由语言本身提供。

3. 算法思想的设计

核心算法思想是**二分搜索**。

1. **可行性函数的单调性**: 关键在于观察 `can_build(h)` 函数的行为。如果一个高度 `h` 是可行的（即所需水量不超过 `x`），那么任何小于 `h` 的高度 `h'` (且 `h' >= 1`) 也一定是可行的。这是因为降低水族箱高度只会减少或保持所需水量，绝不会增加。 相反，如果一个高度 `h` 是不可行的（所需水量超过 `x`），那么任何大于 `h` 的高度 `h''` 也一定是不可行的，因为增加水族箱高度只会增加或保持所需水量。 这种单调性（可行性从 `true` 变为 `false` 只有一个转折点）是应用二分搜索的前提。

2. **搜索空间**： 我们需要寻找的是最大的可行高度 `h`。 `h` 的最小可能值为 `1`。 `h` 的最大可能值可以估算。一个安全的上界可以是 `x` （如果所有珊瑚高度都为 0，最多能把水面升到 `x`，如果 `n=1` 的话）加上数据中最大的珊瑚高度（因为水面至少要没过最高的珊瑚才可能继续增加水量，但实际上 `2*10^9` 左右是一个更安全的上界，覆盖了 `a_i` 和 `x` 的最大值）。代码中使用的 `2000000000LL + 100LL` 是一个足够大的上界。
3. **二分搜索过程**：
   - 在确定的 `[low, high]` 范围内搜索 `h`。
   - 取中间值 `mid`。
   - 调用 `can_build(mid)` 来判断 `mid` 是否可行。
     - 如果 `mid` 可行：说明 `mid` 是一个潜在的答案，并且可能还存在更大的可行高度。因此，我们将 `ans` 更新为 `mid`，然后将搜索范围调整到 `[mid + 1, high]`，尝试寻找更高的解。
     - 如果 `mid` 不可行：说明 `mid` 太高了，需要的水量超标。因此，最大可行高度必定小于 `mid`。我们将搜索范围调整到 `[low, mid - 1]`。
   - 重复此过程，直到 `low > high`，此时 `ans` 中存储的就是最大的可行高度。

4. 关键功能的算法步骤

A. 检查给定高度的可行性 （`can_build` 逻辑）

1. 初始化 `所需总水量` 为 0。
2. 对于珊瑚礁中的每一列 `i` 从 0 到 `n-1`： a。 获取第 `i` 列珊瑚的高度 `当前珊瑚高度`。 b. 如果 `当前珊瑚高度` 小于 `候选水族箱高度`： i。 计算该列需要的水量： `差额 = 候选水族箱高度 - 当前珊瑚高度`。 ii. 累加到 `所需总水量`： `所需总水量 = 所需总水量 + 差额`。 c. （优化）如果 `所需总水量` 已经大于 `最大允许水量`： i。 立即返回 `不可行` （false）。
3. 如果遍历完所有列后， `所需总水量` 小于或等于 `最大允许水量`： a。 返回 `可行` （true）。
4. 否则（虽然在步骤 2c 中已处理，但作为完整逻辑的一部分）： a。 返回 `不可行` （false）。

B. 求解最大可行高度 （`solve` 逻辑）

1. 读取输入：珊瑚列数 `n`，最大允许水量 `x`，以及各列珊瑚的高度存入列表 `a`。
2. 初始化搜索下界 `low = 1`。
3. 初始化搜索上界 `high` 为一个足够大的值（例如， `2 * 10^9 + 100`）。
4. 初始化当前找到的最大可行高度 `ans = 0` （或 `1`，因为 `h >= 1` 且题目保证有解）。
5. 当 `low` 小于或等于 `high` 时，重复以下步骤： a。 计算中间高度 `mid = low + (high - low) / 2` （这种计算方式可以防止 `low + high` 溢出）。 b. 调用“检查给定高度的可行性”算法（即 `can_build`）判断 `mid` 是否可行，传入参数 `mid`， `n`， `x`， `a`。 c. 如果 `mid` 可行： i。 更新 `ans = mid` （因为 `mid` 是一个可行的解，且我们想找最大的）。 ii. 更新 `low = mid + 1` （尝试寻找更高的可行高度）。 d. 如果 `mid` 不可行： i。 更新 `high = mid - 1` （当前 `mid` 太高，需要降低高度）。
6. 输出 `ans`。

三、算法性能分析

1. `can_build(h_candidate, n, max_water, corals)` **函数**:
   - 该函数需要遍历所有 `n` 列珊瑚。
   - 对于每一列，它执行常数时间的比较和算术运算。
   - 因此，`can_build` 函数的时间复杂度为 **O（N）**，其中 N 是珊瑚的列数。
2. `solve()` **函数**:
   - 该函数的核心是二分搜索。二分搜索的范围是从 `1` 到约 `H_max`（例如 `2 * 10^9`）。
   - 二分搜索的迭代次数约为 `log_2(H_max)`。
   - 在二分搜索的每次迭代中，都会调用一次 `can_build` 函数，其复杂度为 O（N）。
   - 因此，`solve` 函数的总时间复杂度为 **O（N * log H_max）**。
     - 鉴于 `N` 最大为 `2 * 10^5`，`H_max` 约为 `2 * 10^9`，`log_2(2 * 10^9)` 大约是 `log_2(2) + log_2(10^9) = 1 + 9 * log_2(10) ≈ 1 + 9 * 3.32 ≈ 31`。
     - 所以，总操作次数大约在 `(2 * 10^5) * 31` 的数量级，这是可以在典型时限内完成的。
3. **空间复杂度**:
   - 存储珊瑚高度的向量 `a` 需要 O（N） 的空间。
   - 其他变量（如 `n`，`x`，`low`，`high`，`mid`，`ans`）占用 O（1） 的空间。
   - 因此，算法的总体空间复杂度为 **O（N）**。
4. **关于输入读取和多测试用例**:
   - 读取 `n` 个珊瑚高度需要 O（N） 时间。
   - 如果总共有 `t` 个测试用例，则总的时间复杂度将是 `t * O(N * log H_max)`。
   - `ios_base::sync_with_stdio(false); cin.tie(NULL);` 用于加速 C++ 的 I/O 操作，这对于处理大量输入数据是重要的优化。

该算法利用二分搜索有效地在巨大的可能高度范围内找到了最优解，并且其复杂度在给定约束条件下是可接受的。

四、总结与收获

通过完成本次“建造水族馆”问题的实验，收获颇丰。

深刻理解题目要求，并将实际问题抽象为数学模型是解决问题的关键一步。在数据结构的选择上，认识到 vector 对于存储和快速访问动态数量的珊瑚高度非常合适。通过分析问题中“高度可行性”的单调特性，成功应用了二分搜索算法来高效地找到最优解，这极大地降低了时间复杂度。学习了如何确定二分搜索的合理边界，并对算法的时间复杂度和空间复杂度进行了分析，这加深了对算法效率评估的理解。实现过程也锻炼了我的C++编程能力和细节处理能力，例如处理大数据范围时采用 `long long` 类型避免溢出，以及优化I/O操作等。

这次实验不仅巩固了数据结构和算法知识，也提升了分析问题和解决问题的综合能力。