计算机学院《算法设计与分析》第二次作业

20373363 李子涵

October 22, 2022

1 小跳蛙问题

给定 n 块石头，依次编号为 1 到 n，第 i 块石头的高度是 h_i，青蛙最远跳跃距离 k。
现有一只小跳蛙在第 1 块石头上，它重复以下操作，直到它到达第 n 块石头：
若它当前在第 i 块石头上，则可跳到第 $j(i+1 \leq j \leq min(i+k, n))$ 块石头上，耗费的体力为 $|h_i - h_j|$。
试设计算法求它最少耗费多少体力可以到达第 n 块石头，写出伪代码并分析算法的时间复杂度。

1.1 状态设计

$s[i]$ 表示跳到第 i 块石头上耗费的最小体力。

1.2 状态转移

青蛙当前所在第 i 块石头，青蛙的最远跳跃距离 k，则青蛙可以从第 $i-j$ 块石头跳过 $j = 1, 2, .., min\{k, i-1\}$ 块石头到达第 i 块石头。此时耗费的体力为 $s[i-j] + |h_i - h_{i-j}|$。

故转移方程为：$s[i] = \min\limits_{j \in [1..min(k,i-1)]} \{s[i-j] + |h_i - h_{i-j}|\}$

1.3 边界条件

小跳蛙在第一块石头，初始状态为 $s[1] = 0$。

1.4 目标状态

到达第 n 块石头的最小代价 $s[n]$。

1.5 伪代码

Algorithm 1: 到达第 n 块石头耗费的最小体力

```
Input: 代表每块石头高度的数组 h[1..n]，青蛙的最大跳跃距离 k
Output: 到达第 n 块石头耗费的最小体力
function minStrength(h[1..n], k):
    // 新建状态数组s[1..n]
    let s[1..n] be new array;
    // 设置初始状态
    s[1] ← 0;
    for i ← 2 to n do
        // 赋初值为从前一块石头转移过来的体力
        s[i] ← s[i−1] + abs( h[i] − h[i−1] );
        // 记录跳到第 i 块石头时的最小体力
        for j ← 2 to min(k, i−1) do
            s[i] ← min{s[i], s[j] + abs( h[i] − h[i−j] )};
        end
    end
    // 最小体力
    minStrength ← s[n];
    return minStrength;
end
```

1.6 复杂度分析

总状态是 $O(n)$ 级别的，每个状态的转移是 $O(k)$ 的，所以总时间复杂度为 $T(n) = O(nk)$。

2 二进制串变换问题

给定两个长度均为 n 的仅由 0 和 1 组成的字符串 a 和 b，你可以对串 a 进行如下操作:

1. 对任意 $i, j(1 \le i, j \le n)$，交换 a_i 和 a_j，操作代价为 $|i-j|$;
2. 对任意 $i(1 \le i \le n)$，取反 a_i，操作代价为 1;

请你设计算法计算将串 a 变为串 b 所需的最小代价（只能对串 a 进行操作)，写出伪代码并分析算法的时间复杂度。

2.1 状态设计

$c[i]$ 表示把串 a 的前 i 位变成串 b 的前 i 位的最小代价。

2.2 状态转移

由操作代价知，只有**连续两个**数字需要取反，且连续的两个数字**不相等**时，选择**交换**这两个数字的代价**严格小于**对两个数字分别取反。

故有如下递推式:

$$c[i] = \begin{cases} c[i-1]+1 & \textbf{if } (a[i] \neq b[i]) \textbf{ and } (\ (a[i-1] == b[i-1]) \textbf{ or } (a[i-1] == a[i])\ ) \\ c[i-2]+1 & \textbf{if } (a[i] \neq b[i]) \textbf{ and } (a[i-1] \neq b[i-1]) \textbf{ and } (a[i-1] \neq a[i]) \\ c[i-1] & \textbf{if } a[i] == b[i] \end{cases}$$

2.3 边界条件

$i=0$ 时，串 a 和串 b 都是空串，不需要进行操作。$c[0]=0$。

$a[1]$ 为串 a 的第一个数字，不会与前序数字进行交换操作，只需考虑是否需要进行取反操作。如果 $a[1]==b[1]$，第一个数字不需要取反，$c[1]=0$；否则 $a[1]\neq b[1]$，第一个数字需要取反，$c[1]=1$。

2.4 目标状态

将串 $a[1..n]$ 变为串 $b[1..n]$ 所需的最小代价 $c[n]$。

2.5 伪代码

Algorithm 2: 将串 a 变为串 b 所需的最小代价

Input: 代表串 a 的数组 $a[1..n]$，代表串 b 的数组 $b[1..n]$

Output: 将串 a 变为串 b 所需的最小代价

```
function minCost(a[1..n], b[1..n]):
    // 新建状态数组c[0..n]
    let c[0..n] be new array;
    // 设置初始状态
    c[0] = 0;
    c[1] = (a[1] == b[1]) ? 0 : 1;
    for i ← 2 to n do
        if (a[i] == b[i]) then
            c[i] ← c[i − 1];
        end
        else if (a[i − 1] == b[i − 1]) or (a[i − 1] == a[i]) then
            c[i] ← c[i − 1] + 1;
        end
        else
            c[i] ← c[i − 2] + 1;
        end
    end
    minCost ← c[i];
    return minCost;
end
```

2.6 复杂度分析

总状态是 $O(n)$ 级别的，每个状态的转移是 $O(1)$ 的，所以总时间复杂度为 $T(n)=O(n)$。

3 球队组建问题

有 $2n$ 个学生分为两排，每排有 n 个人，由左至右分别编号为 $1,2,..,n$，如图所示。现在请你在这两排学生中挑选出一些学生组成一支球队，挑选出的学生编号必须是严格递增的（编号相同的两名学生最多只能取其中一个）。此外，为避免球队中的队员都来自同一排，不能同时选择同一排相邻的两名学生（例如，若选择第一排的 5 号同学，就不能再选择第一排的 4 号和 6 号同学）。组建队伍的总人数没有限制。

给出同学们的身高数据 $h_{i,j}, h_{1,k}(1\leq k\leq n)$ 表示第一排同学的身高，$h_{2,k}(1\leq k\leq n)$ 表示第二排同学的身高。请你设计算法使组建成的球队中队员的身高之和最大，写出伪代码并分析算法的时间复杂度。

3.1 状态设计

设按序号从小到大选择组建成的球队的队员。$sum[i][j]$ 表示选择第 i 排编号为 j 的的同学加入球队或者不选择编号为 j 的同学加入球队后，此时队伍的最大总身高。

注:

1. $i=1$ 或 2 时，分别表示选择**第一排或第二排**编号 j 的同学加入球队。
2. $i=0$ 时，表示**不选择**同学加入球队作为编号 j 的队员。

3.2 状态转移

考虑到**不能**同时选择同一排**相邻**的两名学生:

1. 如果选择第一排编号为 j 的队员, 则前一个编号为 $j-1$ 的队员可以**不选**或者选择**第二排**编号为 $j-1$ 的同学。此时可能获得的最大总身高为 $h_{1,j}+max\ \{\ sum[0][j-1],\ sum[2][j-1]\ \}$。

2. 如果选择第二排编号为 j 的队员, 则前一个编号为 $j-1$ 的队员可以**不选**或者选择**第一排**编号为 $j-1$ 的同学。此时可能获得的最大总身高为 $h_{2,j}+max\ \{\ sum[0][j-1],\ sum[1][j-1]\ \}$。

3. 如果不选择编号 j 的同学加入球队作为编号 j 的队员, 则前一个编号为 $j-1$ 的队员可以选择**第一排或者第二排**编号为 $j-1$ 的同学。此时可能获得的最大总身高为 $max\ \{\ sum[1][j-1],\ sum[2][j-1]\ \}$。

故有如下递推式:

$$sum[i][j]=\begin{cases} h_{1,j}+max\ \{\ sum[0][j-1],\ sum[2][j-1]\ \} & \textbf{if } i=1 \\ h_{2,j}+max\ \{\ sum[0][j-1],\ sum[1][j-1]\ \} & \textbf{if } i=2 \\ max\ \{\ sum[1][j-1],\ sum[2][j-1]\ \} & \textbf{if } i=0 \end{cases}$$

3.3 记录决策方案

记录前序队员所在排数：数组 $row[i][j]$，表示第 j 个队员来自第 i 排时，第 $j-1$ 个队员来自于第 $row[i][j]$ 排，即 $sum[i][j]$ 状态是由 $sum[\ row[i][j]\ ][\ j-1\ ]$ 状态转移而来的。从而利用数组 row 迭代得到编号为 $j, j-1, .., 1$ 的队员来自第几排。

记录最大身高和对应的决策方案：数组 $choose[1..choose.length]$，其中 $choose[j]$ 记录每个编号为 j 的队员来自第几排 ($choose[j]=1$ 或 2) 或者不选择编号为 j 的队员 ($choose[j]=0$)。

状态转移结束后，找到 $sum[k][n] (k\in[1..3])$ 中最大身高和对应的下标 k，记录 $choose[n]=k$, 就可以利用记录前序队员所在排数的 $row[i][j]$ 数组，得到前序队员所在排数 $row[\ choose[n]\ ][\ n-1\ ]$ 得到第 $n-1$ 前序队员所在的排数，进而迭代得到所有前序队员所在的排数，记入数组 $choose[1..choose.length]$，代表最大身高和对应的决策方案。

3.4 边界条件

编号为 $j=1$ 的队员没有选择的前置约束,故对应的最大总身高 $sum[0][1]=0$, $sum[1][1]=h_{1,1}$, $sum[2][1]=h_{2,1}$，且第一个队员前没有队员，故 $row[0][1]$, $row[1][1]$, $row[2][1]$ 无意义。

3.5 目标状态

则最大总身高对应的状态为 $sum[k][n]$, $s.t.\ sum[k][n]=max\{\ sum[0][n],\ sum[1][n],\ sum[2][n]\ \}$。对应的队员选择方案为数组 $choose[1..choose.length]$，其中 $choose[j]$ 记录每个编号为 j 的队员来自第几排 ($choose[j]=1$ 或 2) 或者不选择编号为 j 的队员 ($choose[j]=0$)。。

3.6 伪代码

Algorithm 3: 组建球队使得队员的身高之和最大，及队员的选择方案

Input: 代表队员身高的二维数组 $h[1..2][1..n]$，其中 $h[i][j]$ 代表第 i 排编号为 j 的队员身高 $h_{i,j}$

Output: 组建的最多 n 个队员的球队的最大身高之和

```
function maxSumHeight(h[1..2][1..n]):
    // 新建状态数组和记录前序队员的数组sum[0..2][1..n],row[0..2][1..n]
    let sum[0..2][1..n],row[0..2][1..n] be new arrays;
    // 记录初始状态
    sum[0][1] ← 0, sum[1][1] ← h[1][1], sum[2][1] ← h[2][1];
    // 状态转移
    for j ← 2 to n do
        // 第j个球员从第一排选择
        sum[1][j] ← h[1][j] + max{ sum[0][j − 1], sum[2][j − 1] };
        row[1][j] ← (sum[0][j − 1] > sum[2][j − 1]) ? 0 : 2;
        // 第j个球员从第一排选择
        sum[2][j] ← h[2][j] + max{ sum[0][j − 1], sum[1][j − 1] };
        row[2][j] ← (sum[0][j − 1] > sum[1][j − 1]) ? 0 : 1;
        // 不选择第j个球员
        sum[0][j] ← max{ sum[1][j − 1], sum[2][j − 1] };
        row[0][j] ← (sum[1][j − 1] > sum[2][j − 1]) ? 1 : 2;
    end
    // 记录最大身高和与其对应的第n个球员的排数
    maxSumHeight ← 0, lastRow ← 0;
    for k ← 0 to 2 do
        if sum[k][n] > maxSumHeight then
            maxSumHeight ← sum[k][n], lastRow ← k;
        end
    end
    // 新建记录决策方案数组choose[1..n]
    let choose[1..n] be new array;
    choose[n] ← lastRow;
    for j ← n to 2 do
        choose[j − 1] ← row[choose[j]][j];
    end
end
```

3.7 复杂度分析

总状态是 $O(n)$ 级别的，每个状态的转移是 $O(1)$ 的，所以总时间复杂度为 $T(n) = O(n)$。

4 括号匹配问题

定义合法的括号串如下:

1. 空串是合法的括号串;

2. 若串 s 是合法的，则 (s) 和 $[s]$ 也是合法的;

3. 若串 a, b 均是合法的，则 ab 也是合法的。

现在给定由 $'[','\,]'$ 和 $'(','\,)'$ 构成的字符串，请你设计算法计算该串中合法的子序列的最大长度，写出伪代码并分析算法的时间复杂度。例如字串 ([(])])，最长的合法子序列 ([()]) 长度为 6。

4.1 状态设计

$len[i][j](i \leq j)$ 表示给定序列中从 i 到 j 的子串中最长的合法子序列。

4.2 状态转移

分别考虑合法字符串的两种构造方式:

对于构造方式 1: 若串 s 是合法的，则 (s) 和 $[s]$ 也是合法的;

对于子串 $s[i..j]$，如果 $s[i]$ 和 $s[j]$ 可以构成 ( ) 或 [ ]，则由构造方式一推出的合法子序列的长度为 $len[i][j] = 2 + len[i+1][j-1]$。

对于构造方式 2: 若串 a, b 均是合法的，则 ab 也是合法的。

对于子串 $s[i..j]$，存在 $i \leq t < j, (t \in [i, j-1])$，使得子串 $s[i..j]$ 由子串 $s[i..t]$ 与子串 $s[(t+1)..j]$ 拼接而成。构造方式二共有 $(j-i)$ 种构造方式，对于每一个 $t \in [i, j-1]$，推出的合法子序列的长度分别为 $len[i][j] = len[i][t] + len[t+1][j]$。

故有如下递推式:

$$len[i][j] = max \begin{cases} 2 + len[i+1][j-1] & \textbf{if } (s[i] + s[j]) == \ '(\,)' \textbf{ or } '[\,]' \\ len[i][t] + len[t+1][j] & t \in [i, j-1] \end{cases}$$

4.3 边界条件

对于单个字符 $s[i](i \in [1..n])$，都不能构成合法字符串，故初始状态为 $len[i][i] = 0$。

对于每个子串 $s[i][i+1](i \in [1..n-1])$，初始状态为

$$len[i][i+1] = \begin{cases} 2 & \textbf{if } s[i, i+1] == \ '(\,)' \textbf{ or } s[i, i+1] == \ '[\,]' \\ 0 & \textbf{if } s[i, i+1] \neq \ '(\,)' \textbf{ and } s[i, i+1] \neq \ '[\,]' \end{cases}$$

4.4 目标状态

串 $s[1..n]$ 中合法的子序列的最大长度 $len[1][n]$。

4.5 伪代码

Algorithm 4: 串 s 中合法的子序列的最大长度

```
Input: 代表串 s 的字符串数组 s[1..n]
Output: 串 s 中合法的子序列的最大长度
function maxIllegalLength(s[1..n]):
    // 新建状态数组len[0..n][0..n]
    let len[0..n][0..n] be new array;
    // 设置初始状态
    for i ← 1 to n do
        len[i][i] ← 0;
    end
    for i ← 1 to n − 1 do
        if s[i, i + 1] == '( )' or '[ ]' then
            len[i][i + 1] ← 2;
        end
        else
            len[i][i + 1] ← 0;
        end
    end
    // 状态转移
    for k ← 2 to n − 1 do
        for i ← 1  to  n − k do
            j ← i + k;
            // k 为子串长度，i 为子串第一个字符下标，j为子串最后一个字符下标
            // 合法字符串构造方式一
            if (s[i] + s[j]) == '( )' or (s[i] + s[j]) == '[ ]' then
                len[i][j] ← 2 + len[i + 1][j − 1];
            end
            else
                len[i][j] ← 0;
            end
            // 合法字符串构造方式二
            for t ← i to j − 1 do
                len[i][j] → max{ len[i][j], len[i][t] + len[t + 1][j] };
            end
        end
    end
    maxIllegalLength ← len[1][n];
    return maxIllegalLength;
end
```

4.6 复杂度分析

总状态是 $O(n^2)$ 级别的，每个状态的转移是 $O(n)$ 的，所以总时间复杂度为 $T(n) = O(n^3)$。

5 箱子问题

给定 n 种箱子 $a_1,..,a_n$，第 i 种箱子 a_i 可表示为 $h_i\times w_i\times d_i$ 的长方体。请用这些箱子搭建一个尽可能高的塔：如果一个箱子 A 要水平的放在另一个箱子 B 上，那么要求箱子 A 底面的长和宽都严格小于箱子 B。可以任意旋转箱子，每种箱子可以用任意次。

设计一个算法求出一个建塔方案使得该塔的高度最高，写出伪代码并分析算法的时间复杂度。

5.1 状态设计

（一）分析题意：

1. 对于一个 $h_i\times w_i\times d_i$ 的箱子，最多两两组合成三种不同底面的箱子。
2. 要求下侧箱子底面的长和宽都**严格小于**上侧箱子，所以可知对于 $h_i\times w_i\times d_i$ 的箱子，假设 $h_i\le w_i\le d_i$，则三种不同底面的箱子**最多有其中两种**箱子可以叠在一个塔中。
3. 为了便于比较两个箱子长和宽，对于 $h_i\le w_i\le d_i$ 的箱子，定义它的三种底面的**长 × 宽**为元组 $(a\times b)$ 分别为 $\{(h_i,w_i),(w_i,d_i),(h_i,d_i)|i\in[1..n],h_i\le w_i\le d_i\}$。则**分别比较** $a_i<a_j,b_i<b_j$，即可得知底面 (a_i,b_i) 是否**严格小于** (a_j,b_j)。

（二）由题意设计数据结构：

记录 3n 个底面及高：由上述分析，新建一个数组 $bottom[1..3n]$，数组元素为箱子的**长宽高**元组 (a,b,c)，对应 $(bottom[i][0],bottom[i][1],bottom[i][2])$。其中元组的前两维作为箱子底面的长和宽，第三维作为箱子的高度。每个箱子分别用三个不同的面作为底面，底面的长和宽与其对应的高作为元组存进 $bottom[1..3n]$ 数组，包括：$\{(h_i,w_i,d_i),(w_i,d_i,h_i),(h_i,d_i,w_i)|i\in[1..n],h_i\le w_i\le d_i\}$。对 $bottom$ 以第一维作第一排序条件，第二维作第二排序条件，由小到大归并排序。

记录状态：$sum[i]$，表示 $bottom[i]$ 的中保存的底面作为**最下方**箱子的底面时，塔的**最大总高度**。

记录决策方案：$upBox[i]$，当 $bottom[i]$ 的面作为**最下方**的箱子的底面时，记其**上一个**箱子序号为 $upBox[i]$，其底面为 $bottom[upBox[i]]$ 中保存的底面。

5.2 状态转移

考虑每个箱子底面 $bottom[i]$ 上方的最大塔高时，只需要考虑底面严格小于 (a_i,b_i) 的底面对应的最大塔高，再加上此底面对应的高 c_i 即可。由上述箱子的数据结构知，$bottom[j](j\in[1..i-1])$ 总满足 $a_i\ge a_j$，$bottom[q](q\in[i+1..3n])$ 总满足 $a_i\le a_q$。故只需在 $bottom[j](j\in[1..i-1])$ 中遍历，对于底面**严格小于** (a_i,b_i) 的底面，找出其对应的**最大塔高**即可。

每个箱子 $bottom[i]$ 为 (a_i,b_i,c_i)，长为 a 和宽为 b 的面作为最下方的箱子的底面时，塔的最大总高度为箱子高度 c_i 与前序箱子 $bottom[1..i-1]$ 中底面严格小于 (a_i,b_i) 的底面 $sum[j](j\in[1..i-1])$ 对应的最大塔高之和 c_i+sum。

故有如下递推式：

$sum[i]=bottom[i][2]+max\{sum[k]\}$,

其中 $k\in[1,i-1]$ **and** $bottom[k][0]<bottom[i][0]$ **and** $bottom[k][1]<bottom[i][1]$

5.3 记录决策方案

记最大塔高对应的决策方案 $boxes[boxes.length]$ 代表**自下而上的箱子序号**，对应箱子为 $bottom[boxes[i]]$。

记 $upBox[i]$ 表示箱子最大高度 $sum[i]$ 是由第 $upBox[i]$ 个箱子的最大塔高 $sum[upBox[i]]$ 加上当前的箱子高度 $bottom[i][2]$ 转移而来的。

为了避免获取最大塔高时需要遍历查找最大值，记录当前最大塔高 $maxHeight$ 和其最底部箱子序号 $downBox$。

可以利用记录上侧箱子序号的 $upBox[1..3n]$ 数组，与最大塔高对应的最底部箱子序号 $boxes[1] = downBox$，得到倒数第二大的箱子序号为 $boxes[2] = upBox[boxes[1]]$，倒数第三大的箱子序号为 $upBox[boxes[2]], ...$ 以此类推直到 $upBox[boxes[boxes.length]]$ 为零，说明 $boxes[boxes.length]$ 即为最顶部的箱子序号，从而迭代得到最底部箱子的上侧所有箱子序号，记为数组 $boxes[1..boxes.length]$。

通过记录最大塔高对应的决策方案的所有**箱子序号**的数组 $boxes[i](i \in [1..boxes.length])$，即可找到所有箱子对应的**底面和高度** $bottom[boxes[i]](i \in [1..boxes.length])$，代表最大塔高对应的决策方案。

5.4 边界条件

对于箱子 $bottom[i]$，如果不存在前序箱子 $bottom[1..i-1]$ 中底面严格小于 (a, b) 的底面，则箱子 $bottom[i]$ 的最大塔高 $sum[i]$ 为箱子高度 $bottom[i][2]$。

此箱子上方没有箱子，故上方箱子序号 $upBox[i]$ 记为 0。

5.5 目标状态

塔的最高高度 $max\{sum[i]\}(i \in [1..3n])$。

5.6 复杂度分析

总状态是 $O(n)$ 级别的，每个状态的转移是 $O(n)$ 的，所以总时间复杂度为 $T(n) = O(n^2)$。

此种方法复杂度为 $O(n^2)$ 的伪代码见下页 Algorithm 5

5.7 方法二 使用树状数组优化时间复杂度为 $O(log(n))$

(一) 数据结构的设计：

1. 记录 3n 个底面及高，与上述方法相同，保证长小于等于宽：新建一个数组 $bottom[1..3n]$，数组元素为箱子的长宽高元组 (a, b, c)，对应 $(bottom[i][0], bottom[i][1], bottom[i][2])$。其中元组的前两维作为箱子底面的长和宽，第三维作为箱子的高度。每个箱子分别用三个不同的面作为底面，底面的长和宽与其对应的高作为元组存进 $bottom[1..3n]$ 数组，包括：$\{(h_i, w_i, d_i), (w_i, d_i, h_i), (h_i, d_i, w_i) | i \in [1..n], h_i \le w_i \le d_i\}$。时间复杂度 $O(n)$。

2. 对于箱子数组 $bottom[1..3n]$，按照先使长递增再使宽递减归并排序，并去除相等的底面，得到 $bottomSortByLength[1..m]$：对 $bottom$ 以元组的第一维（长）作第一排序条件**升序**，长相等时按元组的第二维（宽）作第二排序条件**降序**，归并排序。时间复杂度 $O(nlogn)$。

遍历此数组，如果存在**长宽均相等**的数组元素，只保留此底面与其**最大的高**。记**去重**后数组元素个数为 m。时间复杂度 $O(n)$。

此时得到长递增且长相等时宽递减的有序数组，简称其为**长递增宽递减**数组 $bottomSortByLength[1..m]$。此数组有下列性质：对于任意底面 $bottomSortByLength[k]$，长度严格小于此底面的箱子一定在长递增宽递减数组 $bottomSortByLength[1..k-1]$ 中。

3. 对于长递增宽递减 $bottomSortByLength[1..m]$ 数组与其对应下标 index，按照先使宽递增再使长递减归并排序，得到另一个数组 $bottomSortByWidth[1..m]$：

数组 $bottomSortByWidth[1..m]$ 的元素为四维元组 $(a, b, c, index)$。归并排序时间复杂度 $O(nlogn)$。

简称其为**宽递增长递减**数组 $bottomSortByWidth[1..m]$。此数组有下列性质：对于任意底面宽递增长递减数组 $bottomSortByWidth[k]$，宽度严格小于此底面的箱子一定在 $bottomSortByWidth[1..k-1]$ 中。

4. 记录状态：$sum[i]$，表示 $bottomSortByLength[1..m]$ 的中保存的底面作为**最下方**箱子的底面时，塔的最大总高度。

5. 记录决策方案：$upBox[i]$，当 $bottomSortByLength[1..m]$ 的面作为最下方的箱子的底面时，记其**上一个**箱子序号为 $upBox[i]$，底面为 $bottomSortByLength[\ upBox[i]\ ]$ 中保存的底面。

6. 建立树状数组 $tree[1..m]$：使得对于任意下标 $index$，可以以时间复杂度 $O(logn)$ 查询并更新 $sum[1..index]$ 中的最大值。

（二）状态转移：

遍历宽递增长递减数组 $bottomSortByWidth[1..m]$，获取数组元素 $bottomSortByWidth[k]$ 中存储的下标 $index$，其中 $index$ 对应的箱子底面为 $bottomSortByLength[k]$，更新 $sum[index]$ 的状态。

更新 $sum[index]$ 的状态时，只需根据树状数组 $tree[1..m]$，查询并更新 $sum[1..k]$ 中的最大值。

遍历每个状态的时间复杂度 $O(n)$，每个状态转移时，对于树状数组的查询并更新的时间复杂度为 $O(nlogn)$。故状态转移的总复杂度 $O(nlogn)$。

（三）可以证明此种状态转移方式一定正确且最优：

由（一）数据结构的 2 和 3 知：

1. 遍历到宽递增长递减数组的 $bottomSortByWidth[k]$ 时，只有 $bottomSortByWidth[1..k-1]$ 发生了更新。对于底面 (a_k, b_k)，由宽递增长递减的排序条件可知，$bottomSortByWidth[1..k-1]$ 的宽一定小于或等于 b_k；

2. 查询 $sum[1..index]$ 中的最大值，相当于查询 $bottomSortByLength[1..index]$ 中的最大塔高。对于下标 $p \in [1..index]$，考虑底面 (a_p, b_p) 与底面 (a_k, b_k) 的关系：由数组 $bottomSortByLength$ 长递增宽递减与预处理的去重，必有 $a_p < a_k$，或者 $a_p == a_k \ and \ b_p > b_k$。

- 如果 $a_p == a_k \ and \ b_p > b_k$：由数组 $bottomSortByWidth$ 按宽递增遍历，更新 $sum[k]$ 时，其 $sum[p]$ 尚未被更新，$sum[p] = 0$ 不影响最大值的查询。

- 如果 $a_p < a_k$，且宽 b_p 大于 b_k，则其 $sum[p]$ 尚未被更新 $sum[p] = 0$ 不影响最大值的查询

- 如果 $a_p < a_k$，则宽 b_p 等于 b_k，由数组 $bottomSortByWidth$ 按宽相等时长递减顺序遍历，更新 $sum[k]$ 时，其 $sum[p]$ 尚未被更新，$sum[p] = 0$ 不影响最大值的查询。

- 如果 $a_p < a_k$，且宽 b_p 小于 b_k，则 $sum[p]$ 作为更新 $sum[k]$ 的值之一。

故由上述分析知，遍历到宽递增长递减数组的 $bottomSortByWidth[k]$ 时，如果查询到长递增宽递减数组 $bottomSortByLength[1..index]$ 中的最大非零塔高，则其底面**一定符合严格小于**的性质。

3. 对于未被遍历更新到的 $bottomSortByWidth[k+1..n]$，宽比 b_k 大，不需要用其更新 $sum[k]$。对于 $bottomSortByLength[k+1..n]$，长比 a_k 大，不需要用其更新 $sum[k]$。

4. 故得证。

（四）状态转移方程及决策方案记录

$$sum[index] = \max_{k \in [1..index-1]} sum[k] + bottomSortByLength[index][2]$$

$\max\limits_{k \in [1..index-1]} sum[k]$ 通过树状数组 $tree[1..m]$ 查询并维护。树状数组 $tree[1..m][0]$ 记录最大值对应的 $value$。$tree[1..m][1]$ 记录树状数组作为父节点被更新时，其**最底侧**箱子的序号，此值用于更新每个箱子得到最大塔高 $sum[i]$ 时紧上面的箱子下标 $upBox[i]$。

维护一个**全局最大**塔高 $maxHeight$ 和对应的**最底部**箱子序号 $downBox$。再根据 $upBox[i]$ 记录最大值对应的**前序下标**，与**方法一**中迭代寻找最优决策方案的方法相同。

（五）时间复杂度分析

预处理与归并排序的时间复杂度为 $O(3nlog3n)$。

最多 $3n$ 种状态，每个状态转移时采用树状数组，查询与维护最大塔高都是 $O(log3n)$。故总时间复杂度为 $T(n) = O(3nlog3n) = O(nlogn)$。

此种方法复杂度为 $O(nlogn)$ 的伪代码见下页 Algorithm 6-7

5.8 方法一：塔的最高高度 $O(n^2)$ 的伪代码

Algorithm 5: 方法一：塔的最高高度 $O(n^2)$

Input: 代表每个箱子长宽高的三个数组 $h[1..n], w[1..n], d[1..n]$

Output: 塔的最高高度

function $maxHeight(h[1..n], w[1..n], d[1..n])$**:**

  // 记录每个箱子的三个底面与其对应的高

  let $bottom[1..3n]$ **be new array with value** $\{(h_i, w_i, d_i), (w_i, d_i, h_i), (h_i, d_i, w_i) | i \in [1..n], h_i \leq w_i \leq d_i\}$;

  对 $bottom[1..3n]$ 进行按第一维从小到大、第一维相等按第二维从小到大归并排序;

  // 新建状态数组和记录每个箱子的紧上方箱子序号的数组$sum[1..n], upBox[1..n]$

  let $sum[1..n], upBox[1..n]$ **be new arrays with value 0**;

  // $maxHeight, downBox$记录当前最大塔高和对应底面的箱子序号

  $maxHeight \leftarrow 0, downBox \leftarrow 0$;

  for $i \leftarrow 1$ *to* n **do**

    for $j \leftarrow 1$ *to* $i - 1$ **do**

      if $(bottom[i][0] > bottom[j][0])$ **and** $(bottom[i][1] > bottom[j][1])$ **and** $(sum[i] < sum[j])$ **then**

        $sum[i] \leftarrow sum[j],\ upBox[i] \leftarrow j$;

      end

    end

    $sum[i] \leftarrow sum[i] + bottom[i][2]$;

    if $sum[i] > maxHeight$ **then**

      $maxHeight \leftarrow sum[i], downBox \leftarrow i$;

    end

  end

  // 最大塔高对应的决策方案$boxes[boxes.length]$代表自下而上的箱子序号

  // 序号$boxes[i]$对应的箱子为$bottom[boxes[i]]$

  let $boxes[1..3n]$ **be new array**;

  $boxes.length \leftarrow 1$;

  $boxes[boxes.length] \leftarrow downBox$;

  // 底侧箱子序号$curBox$

  $curBox \leftarrow boxes[boxes.length]$;

  // 底侧箱子的上一个箱子的序号$lastBox$

  $lastBox \leftarrow upBox[curBox]$;

  while $lastBox[curBox] \neq 0$ **do**

    $boxes.length+ = 1$;

    $boxes[boxes.length] = lastBox$;

    $curBox \leftarrow boxes[boxes.length]$;

    $lastBox \leftarrow upBox[curBox]$;

  end

end

5.9 方法二：塔的最高高度 $O(nlogn)$ 的伪代码

Algorithm 6: 树状数组

```
Input: 代表每个箱子长宽高的三个数组 h[1..n], w[1..n], d[1..n]
Output: 塔的最高高度
function searchMaxAndUpdate(h, i):
    max → 0, thisUpBox → 0, find_i → i, update_i → i;
    // find_i指向向下寻找最大值时经历的子节点，update_i指向向上更新最大值时经历的父节点
    while find_i > 0 do
        if max < tree[find_i][0] then
            max ← tree[find_i][0];
            thisUpBox ← tree[find_i][1];
            // 使用树状数组父节点作为最大塔高时，thisUpBox记录此最大塔高的最底层箱子序号
        end
        find_i ← find_i − lowbit(find_i);
    end
    max ← max + h;
    upBox[i] ← thisUpBox;
    // 记录此被更新箱子的紧上方箱子序号
    while update_i ≤ tree.length do
        if max > tree[update_i][0] then
            tree[update_i][0] ← max;
            tree[update_i][1] ← i;
            // tree[update_i][1]记录：树状数组父节点被更新时，其最下方箱子的序号
            // 用于后续使用此树状数组父节点作为最大塔高更新其他箱子时，作为其紧上方箱子序号
        end
        update_i ← update_i + lowbit(update_i);
    end
end
```

接下页

Algorithm 7: 方法二：塔的最高高度 $O(nlogn)$

Input: 代表每个箱子长宽高的三个数组 $h[1..n], w[1..n], d[1..n]$

Output: 塔的最高高度

function $maxHeight(h[1..n], w[1..n], d[1..n])$**:**

// 数组，记录每个箱子的三个底面与其对应的高

let $bottom[1..3n]$ **be new array value** $\{(h_i, w_i, d_i), (w_i, d_i, h_i), (h_i, d_i, w_i) | i \in [1..n], h_i \leq w_i \leq d_i\}$;

对 $bottom[1..3n]$ 进行按第一维从小到大、第一维相等按第二维从大到小归并排序;

对 $bottom[1..3n]$ 进行去重：遍历此数组，如果存在长宽均相等的数组元素，只保留此数组元素的第一维和第二维 (底面) 与其最大的高，记为数组 $bottomSortByLength[1..m]$;

let $bottomSortByWidth[1..m]$ **be new array value copied from array** $bottomSortByLength[1..m]$;

令数组 $bottomSortByWidth[1..m]$ 存储的元组增加一维;

值为其第一次归并排序后的对应下标 $index$;

对数组 $bottomSortByWidth[1..m]$ 进行按第二维从小到大、第一维相等按第二维从大到小归并排序;

// 新建状态数组、树状数组 $sum[1..m], tree[1..m][1..2]$

// 记录每个箱子的紧上方箱子序号的数组 $upBox[1..m]$

let $sum[1..m], tree[1..m][1..2], upBox[1..m]$ **be new arrays with value 0**;

// $maxHeight, downBox$记录当前最大塔高和对应底面的箱子序号

$maxHeight \leftarrow 0, downBox \leftarrow 0$;

for $j \leftarrow 1$ *to* n **do**

    // 从$bottomSortByLength[j][4]$获取当前元素在数组$bottomSortByLength$中的下标

    $i \leftarrow bottomSortByLength[j][4]$;

    // $searchMaxAndUpdate()$函数更新$sum[i]$，$upBox[i]$，$tree[1..m][1..2]$

    $searchMaxAndUpdate(bottomSortByLength[i][2], i)$;

    if $sum[i] > maxHeight$ **then**

        $maxHeight \leftarrow sum[i], downBox \leftarrow i$;

    end

end

// 最大塔高对应的决策方案$boxes[boxes.length]$代表自下而上的箱子序号

// 序号$boxes[i]$对应的箱子为$bottom[boxes[i]]$

let $boxes[1..3n]$ **be new array**;

$boxes.length \leftarrow 1$;

$boxes[boxes.length] \leftarrow downBox$;

// 底侧箱子序号$curBox$

$curBox \leftarrow boxes[boxes.length]$;

// 底侧箱子的上一个箱子的序号$lastBox$

$lastBox \leftarrow upBox[curBox]$;

while $lastBox[curBox] \neq 0$ **do**

    $boxes.length+ = 1$;

    $boxes[boxes.length] = lastBox$;

    $curBox \leftarrow boxes[boxes.length]$;

    $lastBox \leftarrow upBox[curBox]$;

end

end