

Atcoder Educational DP Contest 题解

张晴川

March 13, 2020

Contents

A Frog 1	4
B Frog 2	5
C Vacation	6
D Knapsack 1	7
E Knapsack 2	8
F	9
G	10
H	11
I	12
J	13
K	14
L	15
M	16
N	17
O	18
P	19
Q	20

R	21
S	22
T	23
U	24
V	25
W	26
X	27
Y Grid 2	28
Z Frog 3	31

前言

本文为 **Atcoder** 上的 *Educational DP Contest* 的简要题解。该场比赛共有 26 题，从易到难，是练习动态规划的优秀之选。

提交地址为: <https://atcoder.jp/contests/dp>

如发现错误或有修改建议，请联系我 (qzha536@aucklanduni.ac.nz)。

本文会在我的 github 上 持续更新。

A Frog 1

题意

有 $n(1 \leq n \leq 10^5)$ 块石头 (编号为 1 到 n) 排成一排, 第 i 块石头的高度为 h_i 。青蛙现在站在第 1 块石头上。它可以从第 i 块石头跳到第 $i+1$ 或者 $i+2$ 块石头上。从第 i 块石头跳到第 j 块需要消耗 $|h_j - h_i|$ 点体力。请求出跳到第 n 块石头所需的最小体力值。

题解

设 $dp[i]$ 为跳到第 i 块石头上所需的最小体力值, 则 $dp[n]$ 为题之所求。根据题意, 不难发现有:

$$dp[i] = \begin{cases} 0 & i = 1 \\ |h_2 - h_1| & i = 2 \\ \min(dp[i-1] + |h_i - h_{i-1}|, dp[i-2] + |h_i - h_{i-2}|) & i \geq 3 \end{cases}$$

$O(N)$ 递推一下即可。

核心代码

```
1 dp[1] = 0
2 dp[2] = abs(h[2]-h[1]);
3 for(int i = 2; i<=n; i++){
4     dp[i] = min(dp[i-1]+abs(h[i]-h[i-1]), dp[i-2]+abs(h[i]-h[i-2]));
5 }
```

B Frog 2

题意

有 $n(1 \leq n \leq 10^5)$ 块石头 (编号为 1 到 n) 排成一排, 第 i 块石头的高度为 h_i 。青蛙现在站在第 1 块石头上。它可以从第 i 块石头跳到第 $i+1, i+2, \dots, i+k(1 \leq k \leq 100)$ 块石头上。从第 i 块石头跳到第 j 块需要消耗 $|h_j - h_i|$ 点体力。请求出跳到第 n 块石头所需的最小体力值。

题解

设 $dp[i]$ 为跳到第 i 块石头上所需的最小体力值, 则 $dp[n]$ 为题之所求。根据题意, 不难发现有:

$$dp[i] = \begin{cases} 0 & i = 1 \\ \min\{dp[j] + |h_i - h_j| \mid i - k \leq j < i\} & i \geq 2 \end{cases}$$

$O(nk)$ 递推一下即可。

核心代码

```
1 dp[1] = 0
2 for(int i = 2; i <= n; i++){
3     for(int j = max(i-k, 1); j < i; j++){
4         dp[i] = min(dp[i], dp[j] + abs(h[i] - h[j]));
5     }
6 }
```

C Vacation

题意

太郎的暑假有 $n(1 \leq n \leq 10^5)$ 天，第 i 天他可以选择以下三种活动之一：

- 游泳，获得 a_i 点幸福值。
- 捉虫，获得 b_i 点幸福值。
- 写作业，获得 c_i 点幸福值。

但他不能连续两天进行同一种活动，请求出最多可以获得多少幸福值。

题解

设 $A[i], B[i], C[i]$ 分别表示在第 i 天进行三种活动的前提下，前 i 天的最大幸福值。那么可以得到，

$$A[i] = \begin{cases} a_1 & i = 1 \\ \max(B[i-1], C[i-1]) + a_i & i \geq 2 \end{cases}$$

$B[i]$ 和 $C[i]$ 的求法类似，故不赘述。实现时可以开一个大小为 $n \times 3$ 的二维数组。

核心代码

```
1 //v[d][i] 表示第 d 天进行第 i 项活动获得的幸福值
2 for(int i = 0; i < 3; i++){
3     dp[1][i] = v[1][i];
4 }
5 for(int day = 2; day <= n; day++){
6     for(int cur = 0; cur < 3; cur++){
7         dp[day][cur] = 0;
8         for(int last = 0; last < 3; last++){
9             if(cur != last){
10                 dp[day][cur] = max(dp[day-1][last] + v[day][cur]);
11             }
12         }
13     }
14 }
```

D Knapsack 1

题意

经典背包问题。有 n 个物品，每个物品有重量 w_i 和价值 v_i 。背包的容量为 W ，换句话说，可以放进背包的物品的重量总和不能超过 W 。请最大化被选取物品的价值总和。

数据范围

- $1 \leq n \leq 100$
- $1 \leq W \leq 10^5$
- $1 \leq v_i \leq 10^9$

题解

我们设 $dp[i][j]$ 表示“只考虑前 i 个物品的情况下，背包容量是 j 所能凑出的最大价值之和”。那么在计算 $dp[i][j]$ 的时候，所有情况可以分成两类考虑，

1. **拿第 i 件物品**，那么现在背包容量只剩下 $j - w_i$ ，而由于每样物品只能拿一件，所以我们只需要考虑前 $i - 1$ 件物品的最优选取方式，即最终价值为 $dp[i - 1][j - w_i] + v_i$ 。
2. **不拿第 i 件物品**，那么背包容量仍然是 j 。由于我们选择不拿第 i 件物品，现在只需要考虑前 $i - 1$ 件物品的最优选取方式，即最终价值为 $dp[i - 1][j]$ 。

所以可以得到转移方程为：

$$dp[i][j] = \begin{cases} 0 & i = 0 \\ \max(dp[i - 1][j - w_i] + v_i, dp[i - 1][j]) & i \geq 1 \end{cases}$$

递推或记忆化搜索的复杂度均为 $O(nW)$ 。

核心代码

```
1 //注意要开 long long
2 for(int i = 1; i <= n; i++)
3     for(int j = 0; j <= W; j++)
4         if(j < w[i])
5             //容量不够
6             dp[i][j] = dp[i - 1][j];
7         else
8             //容量足够
9             dp[i][j] = max(dp[i - 1][j - w[i]] + v[i], dp[i - 1][j]);
```

E Knapsack 2

题意

同上题，仅数据范围不同：

数据范围

- $1 \leq n \leq 100$
- $1 \leq W \leq 10^9$
- $1 \leq v_i \leq 10^3$

题解

由于 W 达到了 10^9 的量级，之前的 $O(nW)$ 算法无法通过，但由于每样物品的价值上限仅为 10^3 ，我们可以另辟蹊径。设 $dp[i][j]$ 表示“只考虑前 i 个物品的情况下，总价值是 j 所需的最小容量”。那么在计算 $dp[i][j]$ 的时候，所有情况依然可以分成两类考虑：

1. 拿第 i 件物品，那么别的物品的总价值需要凑出 $j - v_i$ ，而由于每样物品只能拿一件，所以我们只需要考虑前 $i - 1$ 件物品的最优选取方式，即最终重量为 $dp[i - 1][j - v_i] + w_i$ 。
2. 不拿第 i 件物品，那么别的物品需要凑出 j 的价值。由于我们选择不拿第 i 件物品，现在只需要考虑前 $i - 1$ 件物品的最优选取方式，即最小重量为 $dp[i - 1][j]$ 。

计算完所有状态的值后，只需要选取满足重量上限的最大价值即可。

核心代码

```
1  int bound = n*1000; //最大价值
2  for(int v = 0; v <= bound; v++)
3      if(v == 0) dp[0][v] = 0;
4      else dp[0][v] = inf;
5  for(int i = 1; i <= n; i++)
6      for(int tot = 0; tot <= bound; tot++)
7          if(tot < v[i])
8              dp[i][tot] = dp[i-1][tot]
9          else
10             dp[i][tot] = min(dp[i-1][tot], dp[i-1][tot-v[i]] + w[i]);
11  answer = 0
12  for(int tot = 0; tot <= bound; tot++)
13      if(dp[n][tot] <= W)
14          answer = tot
```


F

题意

题解

核心代码

G

题意

题解

核心代码

H

题意

题解

核心代码

I

题意

题解

核心代码

J

题意

题解

核心代码

K

题意

题解

核心代码

1 K

L

题意

题解

核心代码

1 L

M

题意

题解

核心代码

1 M

N

题意

题解

核心代码

1 N

O

题意

题解

核心代码

1 0

P

题意

题解

核心代码

1 P

Q

题意

题解

核心代码

1 Q

R

题意

题解

核心代码

1 R

S

题意

题解

核心代码

1 S

T

题意

题解

核心代码

1 T

U

题意

题解

核心代码

1 U

V

题意

题解

核心代码

1 V

W

题意

题解

核心代码

1 W

X

题意

题解

核心代码

1 X

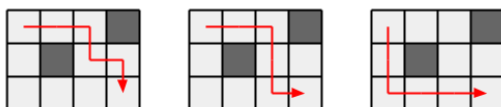
[normalem]ulem

Y Grid 2

题意

给定一个 h 行 w 列的网格，现在位于 $(1,1)$ ，每次可以向下或右走一格，求有多少种方式（对 $10^9 + 7$ 取模）走到 (h,w) 。

网格中有 n 个不能经过的障碍，但不会出现在起点和终点。



数据范围

- $2 \leq h, w \leq 10^5$
- $1 \leq n \leq 3000$

题解

（注：由于非此题重点，本题解不包含计算组合数取模的教程，请自行上网搜板子）本题是 *Grid 1* 的加强版。由于 $h \times w$ 高达 10^{10} ，无法使用之前的方法。接下来我们来使用常见于小学奥数数的**容斥原理**解决这一题。

没有障碍 那么答案就是组合数 $\binom{(h-1)+(w-1)}{h-1}$ —— 因为一共要走 $(h-1)+(w-1)$ 步，选择其中的 $h-1$ 步向下。

一个障碍 设障碍格为 (a,b) ，那么答案应该是总方案数减去经过障碍格的不合法方案数，即：

$$\underbrace{\binom{(h-1)+(w-1)}{h-1}}_{\text{总方案数}} - \underbrace{\binom{(a-1)+(b-1)}{a-1} \binom{(h-a)+(w-b)}{h-a}}_{\text{经过 (a,b) 的非方案数}}$$

两个障碍 设障碍格为 $A:(a,b)$ 和 $B:(c,d)$ ，那么答案等于总方案数减去经过 A 的不合法方案数减去经过 B 的不合法方案数最后加回被计算了两次的同时经过 A 和 B 的不合法方案数。关于最后一部分，这里有两种情况。

1. 不存在同时经过的路径，如上图所示。那么数量就是 0，即答案为

$$\begin{aligned}
& \binom{(h-1)+(w-1)}{h-1} \\
& - \binom{(a-1)+(b-1)}{a-1} \binom{(h-a)+(w-b)}{h-a} \\
& - \binom{(c-1)+(d-1)}{a-1} \binom{(h-c)+(w-d)}{h-c}
\end{aligned}$$

2. 存在同时经过的路径。假设先经过 A 然后经过 B，那么方案数为

$$\begin{aligned}
& \binom{(h-1)+(w-1)}{h-1} \\
& - \binom{(a-1)+(b-1)}{a-1} \binom{(h-a)+(w-b)}{h-a} \\
& - \binom{(c-1)+(d-1)}{a-1} \binom{(h-c)+(w-d)}{h-c} \\
& + \binom{(a-1)+(b-1)}{a-1} \binom{(c-a)+(d-b)}{c-a} \binom{(h-c)+(w-d)}{h-c}
\end{aligned}$$

多个障碍 总的来说，容斥原理告诉我们，对于障碍的每个子集，如果有奇数个就减去，有偶数个就加上。但 n 个障碍就意味着有 2^n 个子集需要计算，这是无法承受的，我们需要加速。

注意到以下事实：

1. 如果一个子集中存在一个格子在另一个格子的**严格**右上方，那么可以直接忽略，因为我们只能向右下方走。
2. 设障碍的一个子集**从左上到右下**依次为 $\{(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)\}$ ，那么这个子集的贡献为：

$$(-1)^m \binom{(x_1-1)+(y_1-1)}{x_1-1} \prod_{i=2}^m \binom{(x_i-x_{i-1})+(y_i-y_{i-1})}{x_i-x_{i-1}} \binom{(h-x_m)+(w-y_m)}{h-x_m}$$

3. 如果把起点和终点也看作的障碍格，那么形式更加简洁：

$$(-1)^m \prod_{i=2}^m \binom{(x_i-x_{i-1})+(y_i-y_{i-1})}{x_i-x_{i-1}}$$

经过以上铺垫，我们大致可以窥到完整解法的轮廓了。首先把所有障碍（以及起点和终点）按左上到右下的顺序排序。那么我们设 $dp[i][0/1]$ 表示走到第 i 个

格子一共经过了奇数/偶数个障碍。那么转移方程为：

$$dp[i][0/1] = \sum_{j \text{ 在 } i \text{ 的左上方}} dp[j][1/0] \binom{(x_i - x_j) + (y_i - y_j)}{x_i - x_j}$$

初始条件为：

$$dp[(1, 1)][1] = 1$$

答案为：

$$dp[(h, w)][0] - dp[(h, w)][1]$$

总复杂度是 $O(n^2)$ 。

核心代码

```
1 dp[0][1] = 1;
2 for(int i=1; i<n+2; i++) {
3     for(int j = 0; j<i; j++){
4         if(x[j]<=x[i] and y[j]<=y[i]){
5             for(int parity = 0; parity<2; parity++){
6                 dp[i][0] += dp[j][1] * binomial(x[i]-x[j]+y[i]-y[j], x[i]-x[j])
7             }
8         }
9     }
10 }
11 ans = dp[n-1][0] - dp[n-1][1]
```

Z Frog 3

题意

有 n 块石头（编号为 1 到 n ）排成一排，第 i 块石头的高度为 h_i ，并满足 $h_1 < h_2 < \dots < h_N$ 。青蛙现在站在第 1 块石头上。它可以从第 i 块石头跳到第 $i+1, i+2, \dots, N$ 块石头上。从第 i 块石头跳到第 j 块需要消耗 $(h_j - h_i)^2 + C$ 点体力。请求出跳到第 n 块石头所需的最小体力值。

数据范围

- $2 \leq N \leq 2 \times 10^5$
- $1 \leq C \leq 10^{12}$
- $1 \leq h_1 < h_2 < \dots < h_N \leq 10^6$

题解

这题用到的技巧被称为**决策单调性**。这是一个貌似高端，但本质简单的知识点。下面我来分别介绍一下什么是**决策**和**单调性**。

什么是决策 首先，我们依旧设 $dp[i]$ 为跳到第 i 块石头上所需的最小体力值。其次，用 $cost(i, k)$ 表示从 i 转移到 k 得到的值，表达式为 $dp[i] + (h_k - h_i)^2 + C$ 。显然有 $dp[k] = \min\{cost(i, k) \mid 1 \leq i < k\}$ 。在计算 $dp[k]$ 的时候，每一个转移源头 i 被称为一个**决策**。

枚举所有的决策的总复杂度是 $O(n^2)$ 。这显然是不可承受的，于是我们需要探索更多性质。



什么是单调性 假设在考虑 $dp[k]$ 的时候，决策 j 要优于决策 i ，即 $cost(i, k) > cost(j, k)$ ，我们看看可以为计算 $dp[k+1]$ 提供什么便利。根据 $cost(i, k)$ 的定义，我们可以得到决策 i 的成本增幅为：

$$\begin{aligned}\Delta_i &= cost(i, k+1) - cost(i, k) \\ &= \underbrace{(dp[i] + (h_{k+1} - h_i)^2 + C)}_{cost(i, k+1)} - \underbrace{(dp[i] + (h_k - h_i)^2 + C)}_{cost(i, k)} \\ &= (h_{k+1} - h_i)^2 - (h_k - h_i)^2 \\ &= (h_{k+1} + h_k - 2h_i)(h_{k+1} - h_k) \quad (\text{平方差公式})\end{aligned}$$

同理可得 $\Delta_j = (h_{k+1} + h_k - 2h_j)(h_{k+1} - h_k)$

注意到 $h_i < h_j < h_k < h_{k+1}$ 是**单调递增**的，不难计算 $\Delta_i - \Delta_j = (2h_j - 2h_i)(h_{k+1} - h_k) > 0$ ，即决策 i 的成本上升得比决策 j 更多。由于决策 j 本来

就是更优的决策，它将一直保持优势下去。于是我们可以得出结论：一旦一个决策被后来者打败，那么再也不需要被考虑。

在计算一个状态的值的时候，我们定义成本最小的决策为最优决策，如有多种取最左的。举个例子：

i	1	2	3	4	5
最优决策	N/A	1	1	2	3

由于之前的结论，最优决策从左到右一定是单调不下降的：只有可能出现 111222333，而不会有 111333222。因为一旦 2 被 3 打败，就永远不会比 3 更优了。这就是所谓的单调性。

由于决策具有单调性，我们只需要二分出每个决策作为最优决策的起点即可。我们这里用 `pair<int,int>` 的数组实现，每个元素 `(i,pos)` 表示决策 `i` 从 `pos` 开始成为最优策略。例如我们用 `[(1,1),(2,4),(3,7)]` 表示 111222333。

鉴于文字描述会过于冗长，直接观察以下伪代码：

核心代码

```

1  dp = [0,inf,inf,...] # 下标从 0 开始
2  optimal_strategy = [(1,2)] # 一开始 0 是所有状态的最优策略
3  for j in [2..n]:
4      # 首先二分出当前位置的最优策略
5          l = 1, r = optimal_strategy.length()
6          while l < r:
7              mid = (l + r) / 2 + 1
8              if optimal_strategy[mid][0] <= j:
9                  l = mid
10             else:
11                 r = mid-1
12         dp[j] = cost(optimal_strategy[l][0],j)
13     # 然后我们计算成为最优决策的起点
14     # 首先去除完全劣于当前策略的策略
15     i,pos = optimal_strategy.back()
16     while cost(i,pos) > cost(j,pos):
17         optimal_strategy.pop()
18         i,pos = optimal_strategy.back()
19     # 开始二分起点
20     l = pos, r = n
21     while l < r:
22         mid = (l + r) / 2
23         if cost(i,mid) <= cost(j,mid):
24             l = mid + 1
25         else:
26             r = mid
27     optimal_strategy.push((j,l)); # 二分后 l 就是想求的起点

```