Atcoder Educational DP Contest 题解

张晴川

March 13, 2020

Contents

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 $n(1 \le n \le 10^5)$ 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：

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

$O(N)$ 递推一下即可。

核心代码

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

B Frog 2

题意

有 $n(1 \le n \le 10^5)$ 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 $i+1, i+2, \ldots, i+k(1 \le k \le 100)$ 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:

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

$O(nk)$ 递推一下即可。

核心代码

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

C　Vacation

题意

太郎的暑假有 $n(1 \le n \le 10^5)$ 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，

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

$B[i]$ 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

```
//v[d][i] 表示第 d 天进行第 i 项活动获得的幸福值
for(int i = 0;i<3;i++){
    dp[1][i] = v[1][i];
}
for(int day = 2;day<=n;day++){
    for(int cur = 0;cur<3;cur++){
        dp[day][cur] = 0;
        for(int last = 0;last<3;last++){
            if(cur != last){
                dp[day][cur] = max(dp[day][cur],dp[day-1][last] + v[day][cur]);
            }
        }
    }
}
```

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

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

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:

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

递推或记忆化搜索的复杂度均为 $O(nW)$。

核心代码

```
//注意要开 long long
for(int i = 1;i<=n;i++)
    for(int j = 0;j<=W;j++)
        if(j < w[i])
        //容量不够
            dp[i][j] = dp[i-1][j];
        else
        //容量足够
            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

- $1 \le n \le 100$
- $1 \le W \le \mathbf{10^9}$
- $1 \le v_i \le \mathbf{10^3}$

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

```
int bound = n*1000;//最大价值
for(int v = 0;v<=bound;v++)
    if(v == 0) dp[0][v] = 0;
    else dp[0][v] = inf;
for(int i = 1;i<=n;i++)
    for(int tot = 0;tot<=bound;tot++)
        if(tot < v[i])
            dp[i][tot] = dp[i-1][tot]
        else
            dp[i][tot] = min(dp[i-1][tot],dp[i-1][tot-v[i]] + w[i]);
answer = 0
for(int tot = 0;tot<=bound;tot++)
    if(dp[n][tot] <= W)
        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

```
K
```

L

题意

题解

核心代码

```
L
```

M

题意

题解

核心代码

```
M
```

N

题意

题解

核心代码

```
N
```

O

题意

题解

核心代码

```
0
```

P

题意

题解

核心代码

```
P
```

Q

题意

题解

核心代码

```
Q
```

R

题意

题解

核心代码

```
R
```

S

题意

题解

核心代码

```
S
```

T

题意

题解

核心代码

```
T
```

U

题意

题解

核心代码

```
U
```

V

题意

题解

核心代码

```
V
```

W

题意

题解

核心代码

```
W
```

X

题意

题解

核心代码

```
X
```

Y

题意

题解

核心代码

```
Y
```

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 $h_1 < h_2 < \cdots < h_N$。青蛙现在站在第 1 块石头上。它可以从第 i 块石头跳到第 $i+1, i+2, \ldots, N$ 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

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

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 $dp[k] = \min\{cost(i,k) \mid 1 \le i < k\}$。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：

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

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

```
dp = [0,inf,inf,...] # 下标从 0 开始
optimal_strategy = [(1,2)] # 一开始 0 是所有状态的最优策略
for j in range(1,n):
# 首先二分出当前位置的最优策略
    l = 1, r = optimal_strategy.length()
    while l < r:
        mid = (l + r) / 2 + 1
        if optimal_strategy[mid][0] <= j:
            l = mid
        else:
            r = mid-1
    dp[j] = cost(optimal_strategy[l][0],j)
# 然后我们计算成为最优决策的起点
    # 首先去除完全劣于当前策略的策略
    i,pos = optimal_strategy.back()
    while cost(i,pos) > cost(j,pos):
        optimal_strategy.pop()
        i,pos = optimal_strategy.back()
    # 开始二分起点
    l = pos, r = n
    while l < r:
        mid = (l + r) / 2
        if cost(i,mid) <= cost(j,mid):
            l = mid + 1
        else:
            r = mid
    optimal_strategy.push((j,l)); # 二分后 l 就是想求的起点
```