Atcoder Educational DP Contest 题解

张晴川

April 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`

本文发布于 `https://github.com/SamZhangQingChuan/Editorials`。如发现错误或有修改建议，请在 Github 上提 issue。如觉得有帮助，不妨 star 以支持我继续写作。

A Frog 1

题意

有 n 块石头排成一排，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。

数据范围

- $1 \leq n \leq 10^5$

题解

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

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

$O(N)$ 递推一下即可。

核心代码

```
dp[1] = 0
dp[2] = abs(h[2]-h[1]);
for(int i = 3;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 块石头排成一排，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 $i+1, i+2, \ldots, i+k$ 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。

数据范围

- $1 \leq n \leq 10^5$
- $1 \leq k \leq 100$

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：

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

$O(nk)$ 递推一下即可。

核心代码

```
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 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。

数据范围

- $1 \leq n \leq 10^5$

题解

设 $\mathtt{A}[i], \mathtt{B}[i], \mathtt{C}[i]$ 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，

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

$\mathtt{B}[i]$ 和 $\mathtt{C}[i]$ 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 01 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$\texttt{dp}[i][j]$ 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 $\texttt{dp}[i][j]$ 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 $\texttt{dp}[i-1][j-w_i] + v_i$。
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 $\texttt{dp}[i-1][j]$。

所以可以得到转移方程为：

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

递推或记忆化搜索的复杂度均为 $O(nW)$。

核心代码

```cpp
//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 $\texttt{dp}[i][j]$ 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 $\texttt{dp}[i][j]$ 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 $\texttt{dp}[i-1][j-v_i] + w_i$。
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 $\texttt{dp}[i-1][j]$。

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 LCS

题意

给定字符串 s 和 t，求最长公共子序列（不需要连续）。

数据范围

- $1 \le |s|, |t| \le 3000$

题解

设 s, t 的长度分别为 n, m。现在考虑两个串的最后字符，一共有两种情况：

1. $s_n = t_m$，那么答案最末字符显然等于也等于它们俩。
2. $s_n \neq t_m$，那么两者必有其一不在答案中，所以答案为 $\texttt{LCS}(s_{1\dots n}, t_{1\dots m-1})$ 和 $\texttt{LCS}(s_{1\dots n-1}, t_{1\dots m})$ 的较长者。

用 $\texttt{dp}[i][j]$ 表示 s 的前 i 个字符和 t 的前 j 个字符的 LCS 长度。那么可以得到转移方程

$$\texttt{dp}[i][j] = \begin{cases} 0 & i = 0 \text{ or } j = 0 \\ \texttt{dp}[i-1][j-1] + 1 & \texttt{s}[i] = \texttt{t}[j] \\ \max(\texttt{dp}[i-1][j], \texttt{dp}[i][j-1]) & \text{otherwise} \end{cases}$$

答案本身只需要通过 $\texttt{dp}[i][j]$ 的转移来源恢复即可，参考以下代码。

核心代码

```cpp
// 计算最优长度
for(int i = 1; i <= n; i++)
    for(int j = 1; j <= m; j++)
        if(s[i] == t[j])
            dp[i][j] = dp[i - 1][j - 1] + 1;
        else
            dp[i][j] = max(dp[i - 1][j], dp[i][j - 1]);
// 恢复答案
int i = n, j = m;
while(dp[i][j] > 0)
    if(s[i] == t[j]) {
        ans[dp[i][j]] = s[i];
        i--;
        j--;
    } else {
        if(dp[i][j] == dp[i - 1][j])i--;
        else j--;
    }
```

G Longest Path

题意

给定一个 n 个点，m 条边的**有向无环图**，求图中最长路径的长度。定义路径长度为边的数量。

数据范围

- $2 \leq N \leq 10^5$
- $1 \leq M \leq 10^5$

题解

本题是**所有**动态规划模型的**理论**基础。抽象地说，所有动态规划的本质都是在一张（带权）有向无环图上求最长路，之前做的背包，最长公共子序列皆是如此。对应关系如下：

动态规划	有向无环图
状态	节点
转移方程	边
无后效性	图中无环
···	···

这题的转移方程比较显然，即 $\texttt{dp}[i]$ 表示以节点 i 为终点的路径中最长的长度。转移方程为：

$$\texttt{dp}[i] = \max\{\texttt{dp}[j] + 1 \mid \text{存在边 } j \to i\}$$

有向无环图最重要的性质就是没有环，这使得我们可以对节点进行拓扑排序，并按照拓扑序的顺序计算 dp 状态的值。相信你对于宽度优先搜索（BFS）没有什么问题，以下我给出一种用深度优先搜索（DFS）的做法，又称记忆化搜索。

核心代码

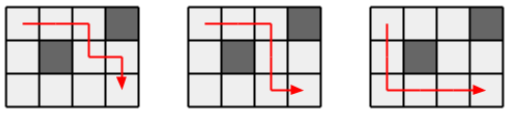
```cpp
void calc(int i) {
    if(vis[i])
        return dp[i];
    else {
        dp[i] = 0;
        vis[i] = true;
        for(auto j:reversed_edges[i]) {
            dp[i] = max(dp[i], calc(j) + 1);
        }
        return dp[i];
    }
}
```

H Grid 1

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)。

网格中有若干个不能经过的障碍，但不会出现在起点和终点。



数据范围

- $2 \le h, w \le 1000$

题解

这题比较简单，用 dp$[i][j]$ 表示走到 (i,j) 的方案数，只需要枚举来源即可。转移方程为：

$$\texttt{dp}[i][j] = \begin{cases} 0 & (i,j) \text{ 是障碍} \\ \texttt{dp}[i-1][j] + \texttt{dp}[i][j-1] & \text{otherwise} \end{cases}$$

顺带一提，如果没有障碍，这个 dp 表格就是 杨辉三角，即 dp$[i][j] = \binom{i+j}{i}$ 。

核心代码

```
dp[1][1] = 1;
for(int i = 1; i <= h; i++) {
    for(int j = 1; j <= w; j++) {
        if(i == 1 and j == 1)continue;
        if(dp[i][j] == '.') //如果可以经过
            dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
    }
}
```

I Coins

题意

有 n 枚硬币排成一排。现在同时抛所有硬币，第 i 枚硬币向上的概率是 p_i，向下的概率是 $1-p_i$，求向上的硬币数量比向下的多的概率。

数据范围

- $1 \le n \le 2999$
- n 是奇数

题解

这题依旧很简单，设 $\texttt{dp}[i][j]$ 表示前 i 个元素有 j 个向上的概率。只需要枚举当前硬币是向上还是向下即可，与背包类似。转移方程为:

$$\texttt{dp}[i][j] = \texttt{dp}[i-1][j-1] \times p_i + \texttt{dp}[i-1][j] \times (1-p_i)$$

计算完毕后，枚举有多少枚硬币向上就做完了。

核心代码

```
prob[0][0] = 1;
for(int i = 1;i<=n;i++){
    dp[i][0] = dp[i-1][0] * (1-p[i]);
    for(int j = 1;j<=i;j++){
        dp[i][j] = dp[i-1][j-1] * p[i] + dp[i-1][j] * (1-p[i]);
    }
}

ans = 0;
for(int j = 0;j<=n;j++){
    if(j > n-j){
        ans += dp[n][j];
    }
}
```

J Sushi

题意

现在有 n 盘寿司在桌上排成一排，第 i 盘寿司里面有 a_i 个寿司。直到吃完所有寿司为止，你都会持续进行下述操作：

- 从 $\{1, 2, \cdots, n\}$ 中**均匀随机**选取一个数 i。如果第 i 盘中仍有寿司，那就吃一块，否则什么也不做。

现在求期望操作次数是多少。

数据范围

- $1 \leq n \leq 300$
- $1 \leq a_i \leq 3$

题解

暴力 DP

首先我们考虑最暴力的做法，用 $\texttt{dp}[c_1][c_2]\cdots[c_n]$ 表示第 i 盘还剩 c_i 个寿司的期望次数。那么枚举随机到的数 i 就可以得到方程：

$$\texttt{dp}[c_1][c_2]\cdots[c_n] = 1 + \sum_{i=1}^{n} \frac{1}{n} \texttt{dp}[c_1][c_2]\cdots[\max(c_i - 1, 0)]\cdots[c_n]$$

合并状态

（注意该方程并不能构成**转移**方程，因为当第 i 盘已经为空的时候，状态不变）

我们现在考虑**合并等价状态**（题外话：动态规划优化的初等方法无非就那么几种，合并状态就是最重要的思想之一）。

注意到由于随机数是均匀选取的，那么盘子的位置是无关紧要的，而只有剩余数量有影响。于是相同数值的不同排列的期望次数必然是相同的。例如 $\texttt{dp}[1][2][3] = \texttt{dp}[3][2][1]$。

所以只需要考虑每种数值出现的次数。因为 $a_i \leq 3$，所以至多有四种不同数值：$\{0, 1, 2, 3\}$。我们重新定义状态 $\texttt{dp}[a][b][c][d]$ 表示当前还剩下 $a/b/c/d$ 盘有 $0/1/2/3$ 个寿司。

有了状态，我们通过枚举当前随机到的盘子里还剩几只寿司得到如下方程：

$$\begin{aligned}\texttt{dp}[a][b][c][d] = 1&+\frac{a}{n}\texttt{dp}[a][b][c][d]\\&+\frac{b}{n}\texttt{dp}[a+1][b-1][c][d]\\&+\frac{c}{n}\texttt{dp}[a][b+1][c-1][d]\\&+\frac{d}{n}\texttt{dp}[a][b][c+1][d-1]\end{aligned}$$

移项整理得到**转移**方程：

$$\begin{aligned}\texttt{dp}[a][b][c][d] =&\frac{n}{b+c+d}\\&+\frac{b}{b+c+d}\texttt{dp}[a+1][b-1][c][d]\\&+\frac{c}{b+c+d}\texttt{dp}[a][b+1][c-1][d]\\&+\frac{d}{b+c+d}\texttt{dp}[a][b][c+1][d-1]\end{aligned}$$

消除无用状态

但由于 $n \le 300$，总状态数高达 $300^4 \approx 8 \times 10^9$ 无法通过。我们需要进一步优化。

注意到任意时刻下，$a+b+c+d$ 的值都应该等于 n：因为不管进行多少次操作，盘子总数总是 n。所以其实只需要知道 b,c,d 就可以反推出 a 的值：$a = n-(b+c+d)$。那么现在只保留后面三维，得到转移方程：

$$\begin{aligned}\texttt{dp}[b][c][d] =&\frac{n}{b+c+d}\\&+\frac{b}{b+c+d}\texttt{dp}[b-1][c][d]\\&+\frac{c}{b+c+d}\texttt{dp}[b+1][c-1][d]\\&+\frac{d}{b+c+d}\texttt{dp}[b][c+1][d-1]\end{aligned}$$

至此，足矣。

核心代码

```
double calc(int b, int c, int d) {
    if(min({b,c,d}) < 0)return 0;
    if(vis[b][c][d])
        return dp[b][c][d];
    else {
        vis[b][c][d] = true;
        double sum = b + c + d;
        dp[b][c][d] = n / sum
                    + b / sum * calc(b - 1, c, d)
                    + c / sum * calc(b + 1, c - 1, d)
                    + d / sum * calc(b, c + 1, d - 1);
        return dp[b][c][d];
    }
}
```

K Stones

题意

给定一个有 n 个正整数的集合 $A = \{a_1, a_2, \ldots, a_n\}$。

现在有 k 块石头和两名玩家。两人轮流进行以下操作，不能进行操作者输：

- 从 A 中指定一个正整数 a_i，并移走恰好 a_i 块石头。

注：每个 a_i 可以重复使用。

数据范围

- $1 \leq n \leq 100$
- $1 \leq k \leq 10^5$
- $1 \leq a_1 < a_2 < \cdots < a_n \leq k$

题解

本题涉及到的知识点是**博弈论**。

注意到以下事实：

1. 剩余 0 块石头时，当前操作者必败。
2. 如果可以使得对手必败，那么当前操作者必胜。
3. 反之，如果不论进行何种操作对手都将必胜，那么当前玩家只能认输。

那么我们设 dp$[i]$ 表示还剩 i 块石头时，当前操作者是否必胜，那么 dp$[i]$ 为**必胜**当且仅当存在 $i \geq a_j$ 且 dp$[i - a_j]$ 为**必败**。

核心代码

```
dp[0] = false;//终局必败
for(int i = 1;i<=k;i++){
    //默认必败
    dp[i] = false;
    for(int j = 0;j<n;j++){
        //如果存在方法使得对手必败，那么当前必胜
        if(i >= a[j] and dp[i-a[j]] == false){
            dp[i] = true;
        }
    }
}
```

L Deque

题意

给定一个双端队列 $a = (a_1, a_2, \ldots, a_n)$。现在有两名玩家轮流进行以下操作，直至队列为空：

- 从队首或队尾取走一个数 c，并且本次收益为 c。

设先/后手的总收益分别为 X 和 Y。先手想要最大化 $X - Y$，后手想要最大化 $Y - X$，即两人都想最大化自己与对手的差值。现两名玩家均以**最优策略**操作，请问最终 $X - Y$ 的值是多少？

数据范围

- $1 \le n \le 3000$
- $1 \le a_i \le 10^9$

题解

其实本题核心就在 **“最优策略”** 这四个字的定义。什么是最优策略？就是选取最坏情况下收益最高的决策。什么是最坏情况呢？这里当然不是指你脑子短路故意选取一个非最优解，而是指假设对方也会以最优策略决策。细心的同学会发现这里的定义是递归的，但这并没有关系，因为在只剩下一个数，或者更平凡的——序列为空的情况下，只存在一种决策供选择。

设 $\texttt{dp}[i][j]$ 表示序列为 $a_i, a_{i+1}, \ldots, a_j$ 时，先手总收益减后手总收益的最大值。那么我们可以得到转移方程：

$$\texttt{dp}[i][j] = \max(a_i - \texttt{dp}[i+1][j], a_j - \texttt{dp}[i][j-1])$$

注意到上式中是减号的原因是，当前的先手在拿完后就变为下一轮的后手，所以当前先手的收益为下一轮中后手的收益。可能比较绕，多思考一下即可。

核心代码

```
void solve(int l,int r){
    if(dp[l][r] != -1)return;
    else{
        if(l == r){
            dp[l][r] = a[l];
        }else{
            solve(l+1, r);
            solve(l, r-1);
            dp[l][r] = max(a[l]-dp[l+1][r],a[r]-dp[l][r-1]);
        }
    }
}
```

M Candies

题意

现在有 n 个小孩和 k 颗糖。第 i 个小孩获得的糖数必须在 $[0, a_i]$ 之内，求一共有多少种把糖发完的方法。

数据范围

- $1 \le n \le 100$
- $0 \le k \le 10^5$
- $0 \le a_i \le k$

题解

有了之前背包的经验，显然我们可以设状态 $\texttt{dp}[i][j]$ 表示前 i 个人一共发了 j 颗糖的方法，并且转移方程就是枚举当前的人获得的糖数 c：

$$\texttt{dp}[i][j] = \sum_{c=0}^{a_i} \texttt{dp}[i-1][j-c]$$

但可达鸭眉头一皱，发现事情并不简单。注意到状态总数为 10^7，而每个状态最多有 $k \le 10^5$ 个转移，总复杂度高达 10^{12}，于是无法通过。

我们仔细观察上面的转移方程，可以发现同一行相邻的两个状态其实差别不大：

$$\texttt{dp}[i][j] = \texttt{dp}[i-1][j-c] + \cdots + \texttt{dp}[i-1][j]$$
$$\texttt{dp}[i][j+1] = \texttt{dp}[i-1][j-c+1] + \cdots + \texttt{dp}[i-1][j+1]$$

通过比对，可以看到其实只首尾两项，于是得到：

$$\texttt{dp}[i][j+1] = \texttt{dp}[i][j] - \texttt{dp}[i-1][j-c] + \texttt{dp}[i-1][j+1]$$

现在计算每个状态只需要 $O(1)$ 次运算，可以通过。

核心代码

```
dp[0][0] = 1;
for(int i = 1; i <= n; i++) {
    dp[i][0] = 1;
    for(int j = 0; j < k; j++) {
        dp[i][j + 1] = dp[i][j] + dp[i - 1][j + 1];
        if(j - a[i] >= 0)
            dp[i][j + 1] -= dp[i - 1][j - a[i]];
    }
}
```

注：本题也可以使用前缀和加速，原理大同小异。

N Slimes

题意

给定序列 $a = [a_1, a_2, \cdots, a_n]$。

每次操作可以把**相邻**的两个元素合并，新元素的值为权值之和。每次合并的成本为新元素的值。

求把所有元素合并成一个的最小成本。

数据范围

- $2 \leq n \leq 400$
- $1 \leq a_i \leq 10^9$

题解

本题是一道典型的**区间型**动态规划。

注意到下列事实：

1. 无论顺序为何，若干元素合并后的权值一定等于初始权值之和。
2. 由于只能合并相邻元素，在**任意**时刻，序列中的每个元素一定对应初始序列的某个连续段：例如 $[1, 2, 3, 4, 5]$ 经过若干次操作后变为 $[(1+2+3), (4+5)]$，则 $(1 + 2 + 3)$ 对应 $[1, 2, 3]$ 。

现在我们来观察将一个序列合并成单独一个元素的过程：$[a_1, a_2, \cdots, a_n] \to s$。不难发现一定是头部若干元素合并后的结果与尾部若干元素合并后的结果的再最终合并的产物。举个例子：

$$[1, 2, 3, 4, 5] \to [(1 + 2 + 3), (4 + 5)] \to 15$$

那么现在设状态 $\texttt{dp}[l][r]$ 表示子序列 $a_l, a_{l+1} \cdots, a_r$ 合并成单一元素的最小成本，通过枚举头尾分界位置 m，可以得到转移方程：

$$\texttt{dp}[l][r] = \min\{\texttt{dp}[l][m] + \texttt{dp}[m + 1][r] \mid l \leq m < r\} + \sum_{i=l}^{r} a_i$$

其中区间和部分可以通过前缀和计算，当然这里由于每个状态有 $O(n)$ 个转移，暴力计算亦可。总复杂度为 $O(n^3)$。

核心代码

```
long long solve(int l, int r) {
    if(l == r) {
        return 0;
    } else {
        if(dp[l][r] != inf) {//已经计算
            return dp[l][r];
        } else {
            for(int m = l; m < r; m++) {
                dp[l][r] = min(dp[l][r], solve(l, m) + solve(m + 1, r));
            }
            for(int i = l; i <= r; i++) {
                dp[l][r] += a[i];
            }
            return dp[l][r];
        }
    }
}
```

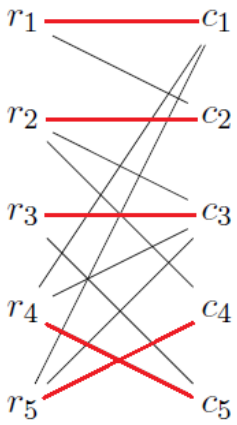
O Matching

题意

给定一个 $n+n$ 个点的二分图，求有多少种完美匹配。

定义：

- 二分图：指顶点可以分成左右两部分，且各自内部顶点间不会有边相连的图。
- 完美匹配：将左右顶点两两配对，但配对的顶点之间必须相邻。



数据范围

- $1 \le n \le 21$

题解

这题是典型的**状态压缩**动态规划，简称状压 DP，一般是指数级的复杂度。尽管复杂度很高，但比最暴力的阶乘级复杂度还是要快不少。

通常我们需要定义一个 `mask` 变量来表示被选取的子集是什么。例如全集是 $\{0,1,2,3,4\}$，而选取的子集是 $\{0,2,3\}$，那么 `mask = 10110`，表达式为 `(1<<0) + (1<<2) + (1<<3)`，其中 `<<` 表示左移。

首先我们考虑暴力做法：依次考虑左边每个点和右边哪个点匹配。假设给定的图是一个二分完全图，即左右之间任意两点均可配对，那么一共有 $n!$ 种匹配方法，在 $n=21$ 的情况下高达 10^{19} 的量级，无法通过。

注意到在考虑顶点 i 的搭档时，我们只关心之前的点已经占用了哪些点，而具体连接方式并不重要。举个例子，假设我们当前考虑顶点 2，之前选择了

$\{(0,3),(1,4)\}$ 还是 $\{(0,4),(1,3)\}$ 对当下的影响是相同的，即不能再使用右边的 3 和 4。

那么我们把等价状态合并：设 dp$[i]$[mask] 表示考虑了节点 $\{0,1,\ldots,i\}$，并且使用的（右侧）点集为 mask。

转移方式很简单：对于状态 dp$[i]$[mask]，只需要枚举节点 i 的邻居 j，如果 mask 包含 j，方案数等于所有 dp$[i-1]$[mask - (1<<j)] 之和。

更进一步地，因为 $\{0,1,\ldots,i\}$ 一共有 $i+1$ 个节点，所以仅当 mask 中 1 的数量等于 $i+1$ 时状态才有意义。于是我们只需要保留 mask 这一维，另一维 i 可以直接设为 __builtin_popcount(mask)-1，其中 __builtin_popcount 表示二进制下 1 的数量。

不难发现，mask 的转移一定是从小的数字转移到大的数字，如 1000 → 1001，所以我们只需要从小到大枚举 mask 即可。

核心代码

```
dp[0] = 1;
for(int mask = 1; mask < (1 << n); mask++) {
    int i = __builtin_popcount(mask) - 1;
    for(int j = 0; j < n; j++) {
        //1. i 和 j 相连
        //2. j 包含在 mask 中
        if(connected[i][j] and (mask & (1 << j)) != 0) {
            dp[mask] += dp[mask - (1 << j)];
        }
    }
}
```

P Independent Set

题意

给一棵 n 个点的树。每个节点可以被染成黑色或白色，但相邻的两个点不能同时染成黑色。求有多少种染色方法。

数据范围

- $1 \leq n \leq 10^5$

题解

本题是树上的动态规划。套路一般都是由儿子的值转移到父亲，自下而上地动态规划。

这道题也不例外，我们设 dp[cur][0/1] 表示以节点 cur 为根的子树，cur 被染成黑色/白色的方案数。

转移方程非常显然：

把根节点染成黑色时，所有儿子被限制为白色，即：

$$\texttt{dp[cur][0]} = \prod \texttt{dp[son][1]}$$

，其中 son 遍历所有 cur 的儿子。

把根节点染成白色时，儿子没有要求，即：

$$\texttt{dp[cur][1]} = \prod (\texttt{dp[son][0]} + \texttt{dp[son][1]})$$

，其中 son 遍历所有 cur 的儿子。

核心代码

```
void dfs(int cur, int fa = 0) {
    dp[cur][0] = dp[cur][1] = 1;
    for(auto son:edge[cur]) {
        if(son != fa) {
            dfs(son, cur);
            dp[cur][0] *= dp[son][1];
            dp[cur][1] *= dp[son][0] + dp[son][1];
        }
    }
}
dfs(1);
ans = dp[1][0] + dp[1][1];
```

Q Flowers

题意

有 n 朵花，从左到右编号为 1 到 n。每朵花有两个属性：高度 h_i 和美丽值 a_i。

我们需要选出一个子序列（可以不连续），使得：

1. 最大化选择的花美丽值的总和
2. 选择的花的高度从左到右严格递增

数据范围

- $1 \le n \le 2 \times 10^5$
- $1 \le h_i \le n$
- h_i 两两不同
- $1 \le a_i \le 10^9$

题解

朴素做法 我们从前向后考虑，设 $\texttt{dp}[i]$ 表示以第 i 朵花结尾的子序列的的最大的美丽值之和。那么一个显然的转移方程是：

$$\texttt{dp}[i] = a_i + \max\{\texttt{dp}[j] \mid j < i \text{ and } h_j < h_i\}$$

可惜复杂度是 $O(n^2)$ 的，但我们可以思考如何优化。

单调性优化 假设我们现在在计算 $\texttt{dp}[i]$ 的值，有 j 和 k 两个决策可以考虑。

如果 j 比 k 更优，即 $\texttt{dp}[j] > \texttt{dp}[k]$ ，并且 $h_j < h_k$，会发生什么呢?

1. 如果可以从 k 转移，因为 $h_j < h_k$，同样也可以从 j 转移。所以不会考虑比 j 劣的 k。
2. 如果无法从 k 转移，那自然不需考虑决策 k。

综上，如果我们只考虑 “有效” 决策，应该尽可能从高度 h 大的决策转移。

现在我们考虑用 `set<pair<long,long>>` 来维护所有有效决策。其中第一维表示高度 h，第二维表示对应的 `dp` 值。在计算 $\texttt{dp}[i]$ 时，使用二分找出比 h_i 小的那些决策中最接近的转移即可。

由于题目保证 a_i 均为正数，不需要考虑 `{h[i],dp[i]}` 本身立刻成为无效决策。只需用 $\texttt{dp}[i]$ 剔除掉比 h_i 大，但更劣的决策即可。详情参考代码。

核心代码

```
set<pair<long, long>> s;
// 存一个初始决策防止无可用决策
s.insert({0, 0});
for(int i = 1; i <= n; i++) {
    //找出前面高度最接近的
    auto optimal = prev(s.lower_bound({h[i], 0}));
    dp[i] = optimal->second + a[i];
    while(true) {
        auto next = s.lower_bound({h[i], 0});
        // 如果存在之后的决策更劣就剔除出去
        if(next != s.end() and next->second <= dp[i])
            s.erase(next);
        else
            break;
    }
    s.insert({h[i], dp[i]});
}
```

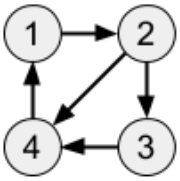
R Walk

题意

给定一个 n 个点的有向图，求图中有多少条长度为 k 的路径。

注：路径可以经过同一顶点和边。

样例



一共有 6 条长度为 2 的路径：

- $1 \to 2 \to 3$
- $1 \to 2 \to 4$
- $2 \to 3 \to 4$
- $2 \to 4 \to 1$
- $3 \to 4 \to 1$
- $4 \to 1 \to 2$

数据范围

- $1 \le n \le 50$
- $1 \le k \le 10^{18}$

题解

我们设 $\texttt{dp}[n][i][j]$ 表示起点和终点分别是 i 和 j，一共走了 n 步的方案数。

Base case 是简单的，$\texttt{dp}[1][i][j]$ 就等于有多少条从 i 到 j 的有向边。

转移方程很显然，我们只需要枚举上一步的位置 k 即可：

$$\texttt{dp}[n+1][i][j] = \sum_{k=1}^{n} \texttt{dp}[n][i][k] \times \texttt{dp}[1][k][j]$$

朴素计算的复杂度是 $O(kn^3)$，在 k 高达 10^{18} 的情况下无法通过。

注意到上述方法本质是：

$$\text{走 n+1 步} = \text{走 n 步} + \text{走 1 步}$$

但我们当然可以拆得更多：

1. 走 2n 步 = 走 n 步 + 走 n 步
2. 走 2n+1 步 = 走 2n 步 + 走 1 步

其实也就是利用快速幂的思想，只不过这里变成了矩阵的幂。由于每走两步，k 至少下降一半，复杂度变为 $O(\log(k)n^3)$，可以通过。具体实现参考代码。

核心代码

```cpp
const int maxn = 50;
typedef array<array<int, maxn>, maxn> State;
long long n, k;

State mul(State a, State b) {
    State res;
    // 初始化
    for(int i = 0; i < n; i++)for(int j = 0; j < n; j++)res[i][j] = 0;

    // i,k,j 的顺序通过缓存优化加快寻址速度
    for(int i = 0; i < n; i++) {
        for(int k = 0; k < n; k++) {
            for(int j = 0; j < n; j++) {
                res[i][j] += 1ll * a[i][k] * b[k][j] % mod;
                res[i][j] %= mod;
            }
        }
    }

    return res;
}

State power(State base, long long k) {
    if(k == 1)return base;
    else if(k % 2 == 0) {
        auto half = power(base, k / 2);
        return mul(half, half);
    } else {
        return mul(power(base, k - 1), base);
    }
}
```

S Digit Sum

题意

求 $\{1, 2, \ldots, k\}$ 中有多少数字的各位之和是 d 的倍数。

数据范围

- $1 \le k < 10^{10000}$
- $1 \le d \le 100$

题解

这题的套路是**数位 DP**。这类题目的基本框架就是求某一个范围 $[l, r]$ 内有多少个数满足 XXX 性质。由于 $[l, r] = [0, r] - [0, l-1]$，我们通常转化为只有上界，而下界为 0 的两个独立 case 分别计算。于是以下只考虑 $[0, r]$ 的计算方法。

假设 $r[0 \ldots n-1]$ 是一个 n 位整数，我们考虑以如下方式暴力枚举所有小于等于 r 的非负整数 x。

```
int x[n],cnt = 0;
for(x[0] = 0;x[0]<10;x[0]++){
    for(x[1] = 0;x[1]<10;x[1]++){
        ...
        for(x[n-1] = 0;x[n-1]<10;x[n-1]++){
            // 1. 如果 x <= r
            // 2. 各位之和模 d 为 0
            if(x <= r and (x[0]+..+x[n-1]) % d == 0){
                cnt++;
            }
        }
    }
}
```

这里我们运用 **01 背包**的思想：在考虑第 i 件物品的时候，只关心之前选择的**总重量**以及**总价值**，而不需要知道具体的方案是什么——即合并等价状态。回到本题，我们枚举第 i 位的时候，只有这些信息是重要的：

1. 之前的数位和。
2. 是否已经小于上界 r，如果是的话，那么可以枚举 `[0-9]`，否则只可以枚举到 $r[i]$ ——因为我们只考虑小于等于 r 的数。

那么我们可以设 `dp[i][j][k = 0/1]` 表示在考虑第 i 位**之前**的数位和 (模 d) 等于 j，以及是否已经小于上界的方案数。

转移也很简单，对于状态 `dp[i][j][k]`，只需要枚举第 i 位的值 `x[i]`，那么转移到的状态为 `dp[i+1][(j+x[i])%d][k or x[i]<r[i]]`。最后一个值的意思是，小于上界的条件是之前已经小于上界或者当前位小于上界对应位。

核心代码

```
dp[0][0][0] = 1;
for(int i = 0; i < n; i++) {
    for(int j = 0; j < d; j++) {
        for(int k = 0; k < 2; k++) {
            for(int cur = 0; cur < 10; cur++) {
                // 如果之前没有已经小于，那么当前位不能超过上界
                if(!k and cur > r[i])break;
                dp[i + 1][(j + cur) % d][k or cur < r[i]] += dp[i][j][k];
            }
        }
    }
}
// 小于上界 + 不小于上界（即等于上界） - 0 的情况（因为题目里不包含 0）
ans = dp[n][0][0] + dp[n][0][1] - 1;
```

T Permutation

题意

给定一个由 '<' 和 '>' 组成的长度为 $n-1$ 的字符串 s。

求有多少 $1-n$ 的排列 p 与 s 的描述相匹配，即如果 $s_i =$ '<'，则 $p_i < p_{i+1}$；如果 $s_i =$ '>'，则 $p_i > p_{i+1}$。

数据范围

- $1 \le n \le 3000$

题解

首先考虑暴力枚举加剪枝的做法。从最后一位开始确定，假设 $p_n = x$ ，那么我们想求出有多少种 $\{1, \dots n\} \setminus \{x\}$ 的排列满足 $s_{1 \dots n-2}$ 的要求，且最末元素小于（或大于）x。

但注意到，集合 $\{1, \dots n\} \setminus \{x\}$ 和集合 $\{1, \dots n-1\}$ 都包含 $n-1$ 个互不相同的数，只考虑顺序的话，在结构上并无不同。$\{1, \dots n\} \setminus \{x\}$ 的排列以第 i 个数结尾的方案数等于 $\{1, \dots n-1\}$ 的排列以第 i 个数结尾的方案数。

现在设 $\texttt{dp}[l][i]$ 表示 $\{1, \dots l\}$ 有多少种排列满足 $s_{1 \dots l-1}$ 的要求且以第 i 个数结尾，其中 $\texttt{dp}[1][1] = 1$。

可以得到转移方程：

$$\texttt{dp}[l][x] = \begin{cases} \sum_{i=1}^{x-1} \texttt{dp}[l-1][i] & s_{l-1} = \texttt{'<'} \\ \sum_{i=x}^{l-1} \texttt{dp}[l-1][i] & s_{l-1} = \texttt{'>'} \end{cases}$$

注：在 '>' 的情况下，下标从 x 开始是因为 $\{1, \dots l\} \setminus \{x\}$ 的第 x 个元素是 $x+1$ 比 x 大。

上述写法是 $O(n^3)$ 的，无法通过，但注意到前缀和形式可以用递推优化到 $O(n^2)$：

$$\texttt{dp}[l][x] = \begin{cases} \texttt{dp}[l][x-1] + \texttt{dp}[l-1][x-1] & s_{l-1} = \texttt{'<'} \\ \texttt{dp}[l][x+1] + \texttt{dp}[l-1][x] & s_{l-1} = \texttt{'>'} \end{cases}$$

核心代码

```
dp[1][1] = 1;
for(int l = 2; l <= n; l++) {
    if(s[l - 1] == '<') {
        for(int x = 1; x <= l; x++) {
            dp[l][x] = dp[l][x - 1] + dp[l - 1][x - 1];
        }
    } else if(s[l - 1] == '>') {
        for(int x = l; x >= 1; x--) {
            dp[l][x] = dp[l][x + 1] + dp[l - 1][x];
        }
    }
}
long long ans = 0;
for(int i = 1; i <= n; i++)
    ans += dp[n][i];
```

U Grouping

题意

有 n 只兔子，现在把兔子分为若干组，如果兔子 i 和兔子 j 在同一组，收益就加上 $a_{i,j}$。求最大收益。

数据范围

- $1 \le n \le 16$

题解

首先设 cost[mask] 表示 mask 对应的子集划分在一组的收益。这一步计算非常简单，用暴力的 $O(2^n \cdot n^2)$ 方法即可：

```
for(int mask = 0; mask < (1 << n); mask++) {
    for(int i = 0; i < n; i++) {
        for(int j = i + 1; j < n; j++) {
            if(((1 << i) & mask) and ((1 << j) & mask)) {
                cost[mask] += a[i][j];
```

接下来设 dp[mask] 表示将 mask 分成若干组的最大收益。计算方法也很简单，对于 mask，只需要枚举子集 submask 递归计算：

$$\texttt{dp[mask]} = \max_{\texttt{submask} \subseteq \texttt{mask}} (\texttt{cost[submask]} + \texttt{dp[mask} - \texttt{submask]})$$

但朴素的方法会导致复杂度高达 $O(2^n \cdot 2^n) = O(4^n)$。

我们考虑用如下方式枚举子集的子集：

```
for(int mask = 0; mask < (1 << n); mask++) {
    for(int submask = mask; submask > 0; submask = (submask - 1) & mask) {
        dp[mask] = max(dp[mask], dp[mask - submask] + cost[submask]);
    }
}
```

不难发现，submask = (submask - 1) & mask 的本质是把当前 submask 的最后一个 1 去掉，并恢复之前被去掉的 1。举个例子：

$$\begin{aligned} \texttt{mask} &= 10101 \\ \texttt{submask} &= 10100 \\ \texttt{(submask - 1) \& mask} &= 10001 \end{aligned}$$

，其中第 3 位被去掉，第 5 位被加上。

因为这种枚举方法每进行一步一定会生成一个新的子集，效率达到最优。我们现在只需要计算复杂度是否可以接受即可。

注意到每个大小为 i 的子集都包含 2^i 个子集，而一共有 $\binom{n}{i}$ 个大小为 i 的子集。所以根据二项式定理，总运算次数应等于：

$$\sum_{i=0}^{n}\binom{n}{i}2^i=\sum_{i=0}^{n}\binom{n}{i}2^i1^{n-i}=(2+1)^n=3^n$$

于是可以通过。

V Subtree

题意

给一棵 n 个点的树，对于每个点，求这棵树（作为图）有多少联通子图包含它。

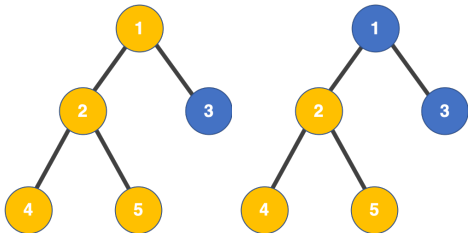
数据范围

- $1 \le n \le 10^5$

题解

首先我们把这棵树看成以 1 为根的有根树。用 $\texttt{son}[i]$ 表示 i 的儿子（们）以及 $\texttt{fa}[i]$ 表示 i 的父亲。

首先注意到树的联通子图必然还是树。对于一个点，我们把所有包含它的子图分成两类，一类以它为根，而另一类的根是它的祖先。假设现在考虑点 2。在下面右图中，子图以 2 为根，但在左图中，以 2 的祖先 1 为根。



首先我们考虑如何计算对于每个点 i，以它为根的方案数 $\texttt{dp}[i]$。由于根已经确定，只需要考虑有多少种向下的延伸方式。由于是树，子树间没有联系，所以向各棵子树的延伸方式是独立的，我们只需要把每棵子树的延伸方案数乘起来即可。

对于一棵根为 s 的子树，要么完全不进入，要么每种延伸方案恰好和以 s 为根的方案一一对应（把根节点 i 去掉就得到了子树的一个方案）。所以可以得到公式：

$$\texttt{dp}[i] = \prod_{s \in \texttt{son}[i]} (1 + \texttt{dp}[s])$$

以上只考虑了每个点作为子图的根的情况，这对于全局根节点 1 是足够的，但其他节点总会有向上延伸的情况。由于 $\texttt{dp}[1]$ 恰好等于各个儿子贡献的乘积，不难可以想到我们可以把父亲看作一个特殊的儿子，相当于把父亲旋转成儿子。而父亲方向的贡献应该等于将 i 这个儿子去掉后所剩的方案数，即 $\frac{\texttt{ans}[f]}{dp[i]}$。由于需要在模意义下计算，我们通过前缀积和后缀积的方法避免除法，具体参见代码。

核心代码

```cpp
long long ans[maxn], dp[maxn];

// 第一次 dfs 只计算向下的方案数
void dfs1(ll cur, ll fa) {
    dp[cur] = 1;
    for(auto s:son[cur]) {
        if(s != fa) {
            dfs1(s, cur);
            (dp[cur] *= dp[s] + 1) %= mod;
        }
    }
}

// 第二次 dfs 计算每个点的答案，其中 faContribution 表示来自父亲的贡献
// 由于根节点没有父亲，向上仅有一种方案，于是默认值为 1
void dfs2(ll cur, ll fa, ll faContribution = 1) {
    ans[cur] = dp[cur] * faContribution % mod;
    vector<long long> prefix_prod, suffix_prod;
    for(auto s:son[cur]) {
        if(s != fa) {
            prefix_prod.push_back(dp[s] + 1);
        }
    }

    suffix_prod = prefix_prod;
    for(int i = 1; i < prefix_prod.size(); i++)
        (prefix_prod[i] *= prefix_prod[i - 1]) %= mod;
    for(int i = (int) suffix_prod.size() - 2; i >= 0; i--)
        (suffix_prod[i] *= suffix_prod[i + 1]) %= mod;

    int cnt = 0;
    for(auto s:son[cur]) {
        if(s != fa) {
            long long contribution = faContribution;
            if(cnt > 0)
                (contribution *= prefix_prod[cnt - 1]) %= mod;
            if(cnt + 1 < suffix_prod.size())
                (contribution *= suffix_prod[cnt + 1]) %= mod;
            dfs2(s, cur, contribution + 1);
            cnt++;
        }
    }
}
```

W Intervals

题意

有 m 个区间 $[l_i, r_i]$，第 i 个区间的权值为 a_i。现在要用 **0** 和 **1** 填一个长度为 n 的数组（下标从 1 开始）。

对于第 i 个区间，如果 $[l_i, r_i]$ 内存在任何一个位置是 **1**，就加上它的权值。求最大权值和。

数据范围

- $1 \le n \le 2 \times 10^5$
- $1 \le m \le 2 \times 10^5$
- $1 \le l_i \le r_i \le n$
- $|a_i| \le 10^9$

题解

这题需要你会使用线段树进行区间更新和区间求最大值。

我们设 $\texttt{dp}[i]$ 表示最后一个 **1** 的位置是 i 的最大权值和，$\texttt{sum}[i]$ 表示所有包含 i 的区间的权值之和。$\texttt{sum}[i]$ 可以用差分、线段树或者其他数据结构求出。

计算 $\texttt{dp}[i]$ 的时候，我们考虑枚举倒数第二个 **1** 所在的位置 j（为了方便实现假设 0 处为 **1**）。如果没有区间同时覆盖 i 和 j 的话，可以得到转移方程:

$$\texttt{dp}[i] = \max_{0 \le j < i} (\texttt{sum}[i] + \texttt{dp}[j])$$

但如果有某个区间 $[l, r]$ 同时覆盖 i 和 j 的话，会导致重复计算贡献。解决方法也很简单，我们考虑用线段树维护 dp 数组。在进入 $[l, r]$ 的时候，给 $[l, r]$ 每个元素 $-a$；在离开 $[l, r]$ 的时候，给 $[l, r]$ 每个元素 $+a$，这一步用区间更新即可。具体参考代码。

核心代码

```
/*
    sum 维护 区间 sum
    dp  维护 区间 max
    update - 区间 add
    query  - 区间 max
*/
for(int i = 1; i <= m; i++) {
    int l, r, a;
    cin >> l >> r >> a;
    sum.update(l, r, 1, a);
    /*
        在 l   加入线段
        在 r+1 移走线段
    */
    start[l].push_back({{l, r}, a});
    finish[r + 1].push_back({{l, r}, a});
}
long long ans = 0;
for(int i = 1; i <= n; i++) {
    // 加入
    for(auto interval:start[i]) {
        int l, r, a = interval.second;
        tie(l, r) = interval.first;
        dp.update(l, r, 1, -a);
    }
    // 移走
    for(auto interval:finish[i]) {
        int l, r, a = interval.second;
        tie(l, r) = interval.first;
        dp.update(l, r, 1, a);
    }
    // 求 sum[i]
    long long s = LLONG_MIN;
    sum.query(i, i, 1, s);
    // 求 dp[0..i-1] 的最优值
    long long max = LLONG_MIN;
    dp.query(0, i - 1, 1, max);
    // 更新 dp[i]
    dp.update(i, i, 1, max + s);
    if(max + s > ans)
        ans = max + s;
}
```

X Tower

题意

给 n 块砖，第 i 块砖的重量为 w_i，坚硬度为 s_i，权值为 v_i。

现在需要选取若干块砖自下而上垒成一座塔，每层恰好有一块砖。对于每一块砖，它上面所有砖的重量之和不能超过他的坚硬度。

求最大化的选取的砖的权值之和。

数据范围

- $1 \le n \le 10^3$
- $1 \le w_i, s_i \le 10^4$
- $1 \le v_i \le 10^9$

题解

这题的难点我们既要选取子集还要规定一个顺序，所以我们先思考如何在给定顺序下选取最优子集。

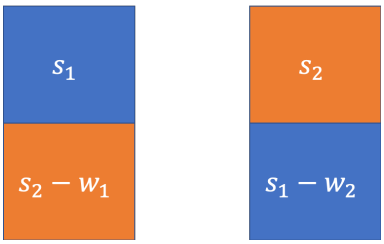
首先注意到这样的事实：在最上层垒一块砖相当于把之前的每一块砖的坚硬度同时减去当前砖的重量，所以根据木桶原理，我们只需要关心最脆弱的砖，而不需要关心每块砖各自的承重。

假设之前最大承重为 j，当前砖的重量为 w，坚硬度为 s，那么在把当前砖垒上去之后新的最大承重为 $\min(j-w, s)$。

现在规定排在后面的砖只能在垒在前面的上方，我们设 $\texttt{dp}[i][j]$ 表示前 i 块砖最多还能承受的重量是 j 的最大权值。和背包的思想类似，容易得到转移：

$$\begin{aligned} \texttt{dp}[i+1][j] &\longleftarrow \texttt{dp}[i][j] && \text{不选} \\ \texttt{dp}[i+1][\min(j-w_{i+1}, s_{i+1})] &\longleftarrow \texttt{dp}[i][j] + v_{i+1} && \text{选} \end{aligned}$$

初始条件为 $\texttt{dp}[0][\infty] = 0$。



确定顺序　现在我们考虑如何确定添加的顺序。

假设最优解中有相邻的两块砖 1 和 2，坚硬度为 s_1, s_2，重量为 w_1, w_2。左图中 1 在上方，右图中 2 在上方。

两种方式得到的最大承重分别是 $\min(s_1, s_2 - w_1)$ 和 $\min(s_2, s_1 - w_2)$。由于两种方式对于权值的贡献都是 $v_1 + v_2$，我们只需要选择能最大化承重的那种即可。

假设 $s_2 - w_1 \geq s_1 - w_2$。由于 s_1 显然大于 $s_1 - w_2$，所以可以得到：

$$\min(s_1, s_2 - w_1) \geq s_1 - w_2 \geq \min(s_2, s_1 - w_2)$$

所以在 $s_2 - w_1 \geq s_1 - w_2$ 的情况下，把 2 放在下面肯定是更好的选择（至少不会更差）。注意到这个条件等价于 $s_2 + w_2 \geq s_1 + w_1$，所以我们可以把所有砖按照 $s_i + w_i$ 从大到小排序，以这个顺序垒起来的砖块一定是最优的。

核心代码

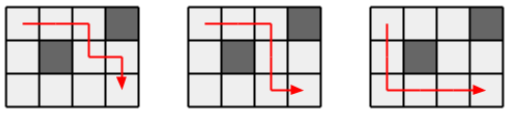
```cpp
struct Brick {
    int w, s, v;
    bool operator<(const Brick &other) const {
        return w + s > other.w + other.s;
    }
};
int main() {
    int n;cin >> n;
    vector<Brick> bricks(n);
    for(auto &brick:bricks) cin >> brick.w >> brick.s >> brick.v;
    sort(begin(bricks), end(bricks));
    const long long maxs = 10010;
    // 无穷大设为两倍最大承重即可
    vector<long long> dp(2 * maxs + 10, 0);
    // 类似于完全背包的写法，因为最大承重只会降低，所以从小的开始枚举
    for(auto brick:bricks)
        for(int j = brick.w; j < dp.size(); j++)
            if(dp[min(j - brick.w, brick.s)] < dp[j] + brick.v)
                dp[min(j - brick.w, brick.s)] = dp[j] + brick.v;
    cout << *max_element(begin(dp), end(dp)) << endl;
}
```

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

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

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~~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:

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

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，无法同时经过。
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 $\texttt{dp}[i][0/1]$ 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：

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

初始条件为：

$$\texttt{dp[(1,1)][1]} = 1$$

答案为：

$$\texttt{dp}[(h,w)][0] - \texttt{dp}[(h,w)][1]$$

总复杂度是 $O(n^2)$。

核心代码

```
dp[0][1] = 1;
for(int i=1;i<n+2;i++) {
    for(int j = 0;j<i;j++){
        if(x[j]<=x[i] and y[j]<=y[i]){
            for(int parity = 0;parity<2;parity++){
                dp[i][0] += dp[j][1] * binomial(x[i]-x[j]+y[i]-y[j],x[i]-x[j])
            }
        }
    }
}
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 $h_1 < h_2 < \cdots < h_N$。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 $\texttt{dp}[i]$ 为跳到第 i 块石头上所需的最小体力值。其次，用 $\texttt{cost}(i,k)$ 表示从 i 转移到 k 得到的值，表达式为 $\texttt{dp}[i]+(h_k-h_i)^2+C$。显然有 $\texttt{dp}[k] = \min\{\texttt{cost}(i,k) \mid 1 \le i < k\}$。在计算 $\texttt{dp}[k]$ 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 $\texttt{dp}[k]$ 的时候，决策 j 要优于决策 i，即 $\texttt{cost}(i,k) > \texttt{cost}(j,k)$，我们看看可以为计算 $\texttt{dp}[k+1]$ 提供什么便利。根据 $\texttt{cost}(i,k)$ 的定义，我们可以得到决策 i 的成本增幅为：

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

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 [2..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
```