综合动态规划

wrpwrp
洛谷网校

课前提示

- 上课的时候专心听讲解，**不要跟着老师抄代码**，下课后独立完成。
- 不使用 AI 做题，AI 会做不等于自己会。
- 不抄袭题解（含对照题解抄一遍），抄对不等于会做。

  看完题解后，关闭题解独立练习。

  练习中途遇到问题，应当分析题目及自己的思路，而非回忆题解或再次参考题解。

前言

今天的主要内容：

进行各种各样的 DP 串讲， 一维/二维DP， 各种背包，区间DP，前缀和优化DP。

除此以外， 也会适当展示 DP 的不同实现方法， 如记忆化搜索。

动态规划的核心要素

状态定义

用一个 变量/数组 记录子问题的最优解

状态转移方程

子问题之间的关系（如何由小问题推到大问题）

初始条件（边界）

最简单的子问题的答案

线性 DP

线性 DP

这类问题主要特征就是在序列上，设计 DP 状态的时候往往倾向于直接设计一个状态 $f[i]$ 表示前 i 个位置的最优解，然后利用线性的结构进行转移。

最长上升子序列

在一个给定的序列中，找到一个最长的子序列，使得这个子序 列元素的数值依次递增。

子序列：子序列就是在原来序列中找出一部分组成的序列（子 序列不一定连续）

最长上升子序列

设 $f[i]$ 表示 以 a$[i]$结尾的最长上升子序列长度。

$$f\ [i] = \max_{1 \le j < i, a[j] < a[i]} f[j] + 1$$

时间复杂度 $O(n^2)$

扩展：$O(n \log n)$ 求最长上升子序列

在一个给定的序列中，找到一个最长的子序列，使得这个子序列元素的数值依次递增。

维护一个辅助序列 num，$num[i]$ 表示在所有长度为 i 的上升子序列中，末尾元素的最小值。设 num 序列当前有 len 个元素。

从前往后考虑每个数，如果当前数比 $num[len]$ 更大，直接将其 插入序列尾部，序列长度 $len++$。

否则用它替换掉序列中，第一个大于等于它的数（保持序列单调性）

最后得到的序列长度即为最长上升子序列长度

扩展：$O(n \log n)$ 求最长上升子序列

```
int len = 1;
num[1] = a[1];
for (int i = 2; i <= n; i ++) {
  if (a[i] > num[len]) {
    ++ len;
    num[len] = a[i];
  }
  else {
    pos = lower_bound(num + 1, num + len + 1, a[i]) - num;
    num[pos] = a[i];
  }
}
```

P1028 [NOIP 2001 普及组] 数的计算

给出正整数 n，要求按如下方式构造数列：

1. 只有一个数 n 的数列是一个合法的数列。
2. 在一个合法的数列的末尾加入一个正整数，但是这个正整数不能超过该数列最后一项的一半，可以得到一个新的合法数列。

请你求出，一共有多少个合法的数列。两个合法数列 a, b 不同当且仅当两数列长度不同或存在一个正整数 $i \leq |a|$，使得 $a_i \neq b_i$。

对于全部的测试点，保证 $1 \leq n \leq 10^3$。

P1028 [NOIP 2001 普及组] 数的计算

设 $f[i]$ 表示以 i 结尾合法序列个数。

$$f[i] = 1 + \sum_{j=0}^{\frac{i}{2}} f[j]$$

至此得到 $O(n^2)$ 的做法，能否得到 $O(n)$ 复杂度的做法呢?

P1028 [NOIP 2001 普及组] 数的计算

考虑前缀和优化。

设 $g[i]$ 表示 $f[i]$ 的前缀和。

$$f[i] = g\left[\frac{n}{2}\right] + 1$$

这样就得到了 O(n) 复杂度的做法。

P3842 [TJOI2007] 线段

在一个 $n \times n$ 的平面上，在每一行中有一条线段，第 i 行的线段的左端点是 (i, L_i)，右端点是 (i, R_i)。

你从 $(1, 1)$ 点出发，要求沿途走过所有的线段，最终到达 (n, n) 点，且所走的路程长度要尽量短。

更具体一些说，你在任何时候只能选择向下走一步（行数增加 1）、向左走一步（列数减少 1）或是向右走一步（列数增加 1）。当然，由于你不能向上行走，因此在从任何一行向下走到另一行的时候，你必须保证已经走完本行的那条线段。

对于 100% 的数据，$1 \leq n \leq 2 \times 10^4$，$1 \leq L_i \leq R_i \leq n$。

P3842 [TJOI2007] 线段

可以发现，如果认为左上角是 $(1, 1)$，右下角是 (n, n)，我们总是大体上从上往下走，所以我们可以顺着 Y 轴考虑，设某个状态 f[i]表示走完前 i 行的状态。

但是直接这样设是无法表达我们处在第 i 行的哪个位置，怎么表达呢？

P3842 [TJOI2007] 线段

我们进一步观察可以发现，一次转移结束，我们要么处于这一行对应线段的左端点，要么处于右端点。

设 $f[i][0], f[i][1]$ 分别表示从起点出发走到第 i 行， 站在当前对应线段的左边或者右边的最短路程， 即可简单转移。

P3842 [TJOI2007] 线段

```
int dis(int i, int isright, int lastpos) {
  if (! isright) {
    return abs (lastpos - r[i]) + r[i] - l[i] + 1;
  }
  else {
    return abs (lastpos - l[i]) + r[i] - l[i] + 1;
  }
}
```

```
f[1][0] = r[1] - 1 + r[1] - l[1];
f[1][1] = r[1] - 1;
for (int i = 2; i <= n; i ++) {
  f[i][0] = min(f[i - 1][0] + dis(i, 0, l[i - 1]), f[i - 1][1] +dis(i, 0, r[i - 1]));
  f[i][1] = min(f[i - 1][0] + dis(i, 1, l[i - 1]), f[i - 1][1] +dis(i, 1, r[i - 1]));
}
long long ans = min (f[n][0] + n - l[n], f[n][1] + n - r[n]);
cout << ans << endl;
```

二维/高维 DP

二维/高维 DP

二维/高维 DP 这个名词可以其实既可以理解成解决高维问题的DP，又可以理解成 dp 数组有多个维度的 DP。

P1004 [NOIP 2000 提高组] 方格取数

设有 $N \times N$ 的方格图 $(N \leq 200)$，每个方 格上有一个自然数。某人从图的左上角 的 A 点出发，可以向下行走，也可以向 右走，直到到达右下角的 B 点。在走过 的路上，他可以取走方格中的数（取走后的方格中将变为数字 0）。

此人从 A 点到 B 点共走两次，试找出 2 条这样的路径，使得取得的数之和最大。

A 0	0	0	0	0	0	0	0
0	0	13	0	0	6	0	0
0	0	0	0	7	0	0	0
0	0	0	14	0	0	0	0
0	21	0	0	0	4	0	0
0	0	15	0	0	0	0	0
0	14	0	0	0	0	0	0
0	0	0	0	0	0	0	0 B

P1004 [NOIP 2000 提高组] 方格取数

这是一个维度比较高的问题， 我们也把思路放开来想。

考虑两条路径同时走，很自然地想到用 $f[x_1][y_1][x_2][y_2]$表示两点位置及此时最大值。

但是这是 n^4 的!

P1004 [NOIP 2000 提高组] 方格取数

缩减冗余维度是高维问题下常见的缩减状态的方法。

由于同一时刻 $x_1 + y_1 = x_2 + y_2 = T + 1$ 状态可以转化为 $f[T][x_1][x_2]$ 。

复杂度优化为 $O(n^3)$

P1004 [NOIP 2000 提高组] 方格取数

```
for (int T = 1; T <= n + n; T ++)
  for (int x1 = 1; x1 <= n; x1 ++)
    for (int x2 = 1; x2 <= n; x2 ++) {
      int y1 = T + 1 - x1;
      int y2 = T + 1 - x2;
      f[T][x1][x2] = max({
        f[T - 1][x1 - 1][x2 - 1],
        f[T - 1][x1][x2 - 1],
        f[T - 1][x1][x2],
        f[T - 1][x1 - 1][x2]
      });
      f[T][x1][x2] += a[x1][y1] + a[x2][y2];
      if (x1 == x2 && y1 == y2)
        f[T][x1][x2] -= a[x1][y1];
    }
cout << f[n + n][n][n] << endl;
```

背包问题

背包问题

背包问题几乎是最常见的一类 DP 问题， 其有几个常见的变种，比如01背包、完全背包、多重背包等， 我们将依次介绍。

01背包

有 N 件物品和一个容量为 V 的背包。每件物品只有一件。放入第 i 件物品耗费的空间是 C_i，得到的价值是 W_i。求解将哪些物品装入 背包可使价值总和最大。

01背包

转移方程：

$$F[i,v]=\max\{F[i-1,v],F[i-1,v-C_i]+W_i\}$$

滚动数组（倒序循环，v 从容量 m 向 0 循环）：

$$F[v]=\max\{F[v],F[v-C_i]+W_i\}$$

完全背包

有 N 件物品和一个容量为 V 的背包。每件物品有无数件。放入第 i 件物品耗费的空间是 C_i，得到的价值是 W_i。求解将哪些物品装入 背包可使价值总和最大。

完全背包

滚动数组（正序循环，v 从小往大循环）：

$$F[v] = \max\{F[v], F[v - C_i] + W_i\}$$

多重背包

有 N 件物品和一个容量为 V 的背包。每件物品有若干件。放入第 i 件物品耗费的空间是 C_i，得到的价值是 W_i。求解将哪些物品装入 背包可使价值总和最大。

多重背包

同一类物品有多个怎么办?

首先这个东西有一个单调队列优化背包的做法，可以让多重背包做到和01背包一致的复杂度，我们会在提高组阶段学习。

我们这里讲个简单的小技巧。

多重背包

我们把物品进行二进制拆分。具体来说， 加入有 c 个 ， 做如下拆分：

$$c = 2^0 + 2^1 + 2^2 + \cdots + r$$

例如， $13 = 1 + 2 + 4 + 6$

对拆分后的物品进行 01 背包即可。

二进制拆分

```
for (int i = 1; i <= n; i ++) {
  int v, w, cnt;
  cin >> v >> w >> cnt;
  for (int j = 0; (1 << j) <= cnt; j ++) {
    V[++ tot] = (1 << j) * v;
    W[tot] = (1 << j) * w;
    cnt -= (1 << j);
  }
  V[++ tot] = cnt * v;
  W[tot] = cnt * w;
}
```

背包问题

背包问题主要的难点在于变式多而奇怪， 我们快速来看看一些背包问题的例子。

P1776 宝物筛选

终于，破解了千年的难题。小 FF 找到了王室的宝物室，里面堆满了无数价值连城的宝物。

这下小 FF 可发财了，嘎嘎。但是这里的宝物实在是太多了，小 FF 的采集车似乎装不下那么多宝物。看来小 FF 只能含泪舍弃其中的一部分宝物了。

小 FF 对洞穴里的宝物进行了整理，他发现每样宝物都有一件或者多件。他粗略估算了下每样宝物的价值，之后开始了宝物筛选工作：小 FF 有一个最大载重为 W 的采集车，洞穴里总共有 n 种宝物，每种宝物的价值为 v_i，重量为 w_i，每种宝物有 m_i 件。小 FF 希望在采集车不超载的前提下，选择一些宝物装进采集车，使得它们的价值和最大。

对于 30% 的数据，$n \leq \sum m_i \leq 10^4$，$0 \leq W \leq 10^3$。

对于 100% 的数据，$n \leq \sum m_i \leq 10^5$，$0 \leq W \leq 4 \times 10^4$，$1 \leq n \leq 100$。

P1776 宝物筛选

标准的多重背包问题， 套用刚刚讲的多重背包的解决方法即可。

[NOIP 2006 提高组] 金明的预算方案

金明有一笔预算 n 元，要从 m 件物品中挑选若干件购买。物品分为主件和附件：主件可以单独购买，而附件必须依附在对应的主件上才能购买，一个主件最多有两个附件。每件物品有价格 v 和重要度 p，其价值定义为 $v \times p$ 。在不超过预算的前提下，要求选择物品使得总价值最大。

$$n \leq 3.2 \times 10^4, m \leq 60, v \leq 10^4$$

[NOIP 2006 提高组] 金明的预算方案

这是一个带依赖的背包问题。

简单来说，就是买一个物品之前，必须先买别的物品。

一般来说，这样的问题需要引入树形dp解决，但是这里有一个特殊的条件：一个主件最多有两个附件

这使得我们可以对物品分组，每组的情况直接枚举出来，问题转化为 01 背包。

[NOIP 2006 提高组] 金明的预算方案

对于每个主件以及依附于它的附件，构成一个物品组，对于这个物品组， 我们有五种购买方式：

不购买、买且只买主件 、买主件，并且买附件1 、买主件，并且买附件2 、买这个主件，并且买附件1和附件2

最外层遍历物品组编号，第二层遍历钱数，转移时取以上五种情况中的 最大值即可。

[NOIP 2006 提高组] 金明的预算方案

```
for (int i = 1; i <= m; i++) {
  for (int j = 1; j <= n; j++) {
    if (a[i].q > 0)
      dp[i][j] = dp[i - 1][j];
    else {
      int f1 = a[i].f1, f2 = a[i].f2;
      int val = a[i].v * a[i].w;
      int val1 = a[f1].v * a[f1].w;
      int val2 = a[f2].v * a[f2].w;
      int t = dp[i - 1][j];
      if (j >= a[i].v)
        t = max (t, dp[i - 1][j - a[i].v] + val);
      if (f1 > 0 && j >= a[i].v + a[f1].v)
        t = max (t, dp[i - 1][j - a[i].v - a[f1].v] + val + val1);
      if (f2 > 0 && j >= a[i].v + a[f2].v)
        t = max (t, dp[i - 1][j - a[i].v - a[f2].v] + val + val2);
      if (f1 > 0 && f2 > 0 && j >= a[i].v + a[f1].v + a[f2].v)
        t = max(t, dp[i-1][j-a[i].v-a[f1].v - a[f2].v] + val + val1 + val2);
      dp[i][j] = t;
    }
  }
}
```

P1941 [NOIP 2014 提高组] 飞扬的小鸟

游戏界面是一个长为 n，高为 m 的二维平面，其中有 k 个管道（忽略管道的宽度）。

小鸟始终在游戏界面内移动。小鸟从游戏界面最左边任意整数高度位置出发，到达游戏界面最右边时，游戏完成。

小鸟每个单位时间沿横坐标方向右移的距离为 1，竖直移动的距离由玩家控制。如果点击屏幕，小鸟就会上升一定高度 x，每个单位时间可以点击多次，效果叠加；如果不点击屏幕，小鸟就会下降一定高度 y。小鸟位于横坐标方向不同位置时，上升的高度 x 和下降的高度 y 可能互不相同。

小鸟高度等于 0 或者小鸟碰到管道时，游戏失败。小鸟高度为 m 时，无法再上升。

现在,请你判断是否可以完成游戏。如果可以，输出最少点击屏幕数；否则，输出小鸟最多可以通过多少个管道缝隙。

对于 100% 的数据：$5 \leq n \leq 10000$，$5 \leq m \leq 1000$，$0 \leq k < n$，$0 < x, y < m$，$0 < p < n$，$0 \leq l < h \leq m$，$l + 1 < h$。

P1941 [NOIP 2014 提高组] 飞扬的小鸟

P1941 [NOIP 2014 提高组] 飞扬的小鸟

这也是一种背包问题在实际考试中的出现形式，看起来跟背包啥关系都没有，需要对问题进行细致的分析。

P1941 [NOIP 2014 提高组] 飞扬的小鸟

观察到 n, m 范围不大，这启发我们可以考虑设计 f[i][j] 表示走到 (i, j) 这个位置的最小操作次数。

我们从前往后考虑每列的操作，其实我们的选项就是，要么点击若干次屏幕往上飞，要么不点屏幕往下掉。

P1941 [NOIP 2014 提高组] 飞扬的小鸟

这其实可以看成“往上飞”就是一个完全背包，“往下掉”就是一个 01 背包。

往上飞：

```
for (int j = x[i] + 1; j <= m; j ++)
  f[i][j] = min ({f[i][j], f[i - 1][j - x[i]] + 1, f[i][j - x[i]] + 1});
for (int j = m - x[i]; j <= m; j ++)
  f[i][m] = min ({f[i][m], f[i - 1][j] + 1, f[i][j] + 1});
```

往下掉：

```
for (int j = 1; j <= m - y[i]; j ++)
f[i][j] = min (f[i][j], f[i - 1][j + y[i]]);
```

P1941 [NOIP 2014 提高组] 飞扬的小鸟

总复杂度是 $O(NM)$ 的

区间 DP

区间 DP

区间动态规划问题的基本思想是，对于每个区间，他们的最优 值都是由几段更小区间的最优值得到，于是可以分而治之。

一般状态直接设计为 $f[l][r]$ 表示区间 $[l,r]$ 最优解，然后枚举中间点转移。

区间 dp 往往形式固定，但是转移较为繁杂，需要仔细思考。

P1775 石子合并（弱化版）

设有 N $(N \leq 300)$ 堆石子排成一排，每堆石子有一定的质量 m_i $(m_i \leq 1000)$。

现在要将这 N 堆石子合并成为一堆。每次只能合并相邻的两堆，合并的代价为这两堆石子的质量之和。

试找出一种合理的方法，使总的代价最小，并输出最小代价。

P1775 石子合并（弱化版）

设 $f[i][j]$ 表示从区间 i 合并到 j 产生的最小值。

通过枚举分界点转移。

$$f[l][r] = \min\{f[l][r], f[l][k] + f[k+1][r] + value\}$$

P1775 石子合并（弱化版）

```
for (int len = 2; len <= n; len ++) {
  for (int i = 1; i + len - 1 <= n; i ++) {
    int j = i + len - 1;
    for (int k = i; k < j; k ++) {
      f[i][j] =
      min(f[i][j], f[i][k] + f[k + 1][j] + sum[j] - sum[i - 1]);
    }
  }
}
```

P1880 [NOI1995] 石子合并

在一个圆形操场的四周摆放 N 堆石子，现要将石子有次序地合并成一堆，规定每次只能选相邻的 2 堆合并成新的一堆，并将新的一堆的石子数，记为该次合并的得分。

试设计出一个算法,计算出将 N 堆石子合并成 1 堆的最小得分和最大得分。

$1 \leq N \leq 100$，$0 \leq a_i \leq 20$。

P1880 [NOI1995] 石子合并

断环为链!

P1880 [NOI1995] 石子合并

具体来说，对于环形问题， 我们常常使用断环为链的技巧。

我们可以将原数组复制一份接在数组最后， 这样每连续 n 个元素对应断环为链的其中一种情况，然后我们一次考虑可能的 n 种情况即可。

P1880 [NOI1995] 石子合并

```
for (int len = 2; len <= n; len ++)
  for (int l = 1; l + len - 1 <= n * 2; l ++) {
    int r = l + len - 1;
    int maxx = -1e9, minn = 1e9;
    for (int k = l; k < r; k ++) {
      maxx = max(maxx, f[l][k] + f[k + 1][r] + sum[r] - sum[l - 1]);
      minn = min(minn, g[l][k] + g[k + 1][r] + sum[r] - sum[l - 1]);
    }
  f[l][r] = maxx;
  g[l][r] = minn;
}
int maxx = -1e9, minn = 1e9;
for (int i = 1; i <= n; i ++) {
  maxx = max(maxx, f[i][i + n - 1]);
  minn = min(minn, g[i][i + n - 1]);
}
```

P4342 Polygon

多边形是一个玩家在一个有 n 个顶点的多边形上的游戏。

第一步，删除其中一条边。

随后每一步：选择一条边连接的两个顶点 V1 和 V2，用边上的运算符计算 V1 和 V2 得到的结果来替换这两个顶点。

游戏结束时，只有一个顶点，没有多余的边。

计算最高可能的分数，并输出初始删掉哪条边得到最高分数。

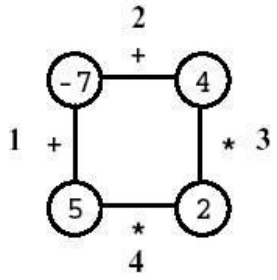

Figure 1. Graphical representation of a polygon

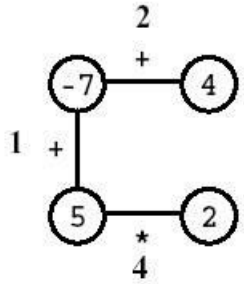

Figure 2. Removing edge **3**

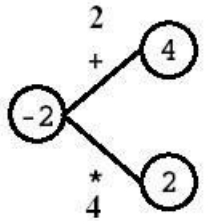

Figure 3. Picking edge **1**

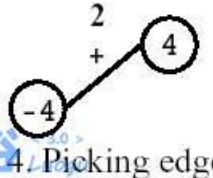

Figure 4. Picking edge **4**

Figure 5. Picking edge **2**

P4342 Polygon

第一次删掉哪条边可以枚举，之后便转变为正常的区间DP。

但要注意，如果只保存 $[l,r]$ 的最大值，这样DP的结果不满足最优子结构与无后效性，因为对于乘法存在负负得正的情形。

为了解决这个问题，我们需要保存 $[l,r]$ 的最大值与最小值。

	加法	乘法
最大值	最大加最大	最大乘最大 最小乘最小
最小值	最小加最小	最大乘最大 最小乘最小 最大乘最小

P4342 Polygon

这题转移和初始化可能显得稍微复杂一点， 并且出现了两个 DP 数组， 展示一下记忆化搜索的写法。

记忆化搜索主要的好处是， 转移很直观， 初值定义也比较直观，在处理复杂问题的时候不容易写错。

一般情况下没有循环快， 但是剪枝方便， 也有可能更快。

P4342 Polygon

```
int F(int l, int r) {
  if (l > r) return 0;
  if (l == r) return a[l];
  if (f[l][r] != 0x3f3f3f3f) return f[l][r];
  int ans = -1e9;
  for (int k = l; k < r; k ++) {
    if (opt[k + 1] == 'x')
      ans = max ({ans,
                  F(l, k) * F(k + 1, r),
                  G(l, k) * G(k + 1, r),
                  F(l, k) * G(k + 1, r),
                  G(l, k) * F(k + 1, r)});
    else
      ans = max (ans, F(l, k) + F(k + 1, r));
  }
  return f[l][r] = ans;
}
```

P4342 Polygon

```
int G(int l, int r) {
  if (l > r) return 0;
  if (l == r) return a[l];
  if (g[l][r] != 0x3f3f3f3f) return g[l][r];
  int ans = 1e9;
  for (int k = l; k < r; k ++) {
    if (opt[k + 1] == 'x')
      ans = min ({ans,
                  F(l, k) * F(k + 1, r),
                  G(l, k) * G(k + 1, r),
                  F(l, k) * G(k + 1, r),
                  G(l, k) * F(k + 1, r)});
    else
      ans = min (ans, G(l, k) + G(k + 1, r));
  }
  return g[l][r] = ans;
}
```

P4342 Polygon

可以看到，出现了“F 调用 G，G 调用 F”的情况，怎么成功通过编译呢？

可以在定义 F 函数内容的前面添加这样两行声明帮助我们通过编译。

```
int F(int l, int r);
int G(int l, int r);
```

总结

动态规划(DP)总结

1.状态类型/结构:

序列、背包、区间、坐标、环形……

2.转移方式: 递推、记忆化搜索

3.优化:

(1)预处理、前缀和

(2)状态量: 跳过无用状态、改进状态表示、利用约束关系

(3)空间: 滚动数组

Thanks