北京亦庄实验中学

研究性学习课程
结题报告

课题名称 浅析复杂动态规划模型及其优化

课题组成员 尹玉文东 张钰晨 蔡越同 李灏冬

课题组组长 尹玉文东

指 导 教 师 张 敏

管 理 教 师 张 敏

研 究 领 域 信 息 学

2022 年 5 月 6 日

摘 要

本文总结了一类以集合信息为状态、状态总数为指数级的动态规划的状态设计方法，即状态压缩法与一类有关于数位的计数类动态规划问题，即数位动态规划。对于一些时间复杂度在指数级上大于空间复杂度的动态规划，本文介绍了单调队列优化，斜率优化，四边形不等式优化及 CDQ 分治优化四种优化方式，可以在较大程度上使时间复杂度靠近空间复杂度。

关键词：动态规划；状态压缩；单调队列；四边形不等式；CDQ 分治

Abstract

An abstract of a dissertation is a summary and extraction of research work and contributions. Included in an abstract should be description of research topic and research objective, brief introduction to methodology and research process, and summary of conclusion and contributions of the research. An abstract should be characterized by independence and clarity and carry identical information with the dissertation. It should be such that the general idea and major contributions of the dissertation are conveyed without reading the dissertation.

An abstract should be concise and to the point. It is a misunderstanding to make an abstract an outline of the dissertation and words “the first chapter”, “the second chapter” and the like should be avoided in the abstract.

Keywords are terms used in a dissertation for indexing, reflecting core information of the dissertation. An abstract may contain a maximum of 5 keywords, with semi-colons used in between to separate one another.

Keywords: Dynamic Programming; State Compression; Monotonic Queue; Quadrilateral Inequality; CDQ divide-conquer

目 录

第 1 章　引言

近年来，动态规划（DP, Dynamic Programming）在信息学中的应用愈发广泛。简单的动态规划模型如线性 DP、背包 DP、区间 DP 已被多数人熟练掌握并运用，本文将不再赘述。而与之形成对比的一些复杂动态规划问题由于其算法本身的难度，难以被应用或被信息学竞赛选手解答出，故本文将这些算法进行了整合并且进行了详细地解析，可以作为信息学竞赛选手及相关领域的参考资料使用。

对于一些时间复杂度本身较劣的 DP 思路，硬件运行速度的提升往往是常量级的，在数据规模快速增长时效果甚微，而直接对 DP 的状态设计进行本质优化往往具有局限性，无法系统整理并在其他 DP 上如法炮制。故本文还将介绍四种被广泛应用的 DP 转移优化方式，在保持原有状态设计和转移方式的基础上最大限度地省去不必要的计算，已达到优化时间复杂度的目的。

第 2 章　基于状态压缩的动态规划

2.1　状态压缩

状态压缩一般是将一个较小的集合内每个元素的状态通过特定的计算方法，转换为单个整数。这样的转换必须是双射。

在信息学竞赛中，状态的压缩和解压缩一般由二进制运算完成。究其主要原因，一方面是因为二进制可以很直接的表达一个元素集合内每个元素选中或不选中的状态，如二进制串 **01001** 可以表示一个大小为 5 的集合 S 中第 2 和 5 个元素被选中，第 1、3、4 个元素未被选中的状态。另一方面，是因为计算机在存储数据时是以二进制方式存储，直接在二进制下对数据进行处理和计算往往比其他进制要快的多。

2.2　位运算

在二进制下对数据的处理和计算与数学中常用的加减乘除和逻辑运算略有不同，对整数在内存中的二进制位进行操作，就是位运算。下面是对几类位运算的定义解释和约定。

- 与（&）：A 和 B 的按位与运算中，对 A 和 B 二进制下的每一位的值进行数学中的逻辑与运算，得到结果中对应位的值。
- 或（|）：A 和 B 的按位或运算中，对 A 和 B 二进制下的每一位的值进行数学中的逻辑或运算，得到结果中对应位的值。
- 取反（~）：A 的按位取反运算中，对 A 二进制下的每一位的值进行数学中的逻辑非运算，得到结果中对应位的值。
- 异或（^）：A 和 B 的按位异或计算中，对 A 和 B 二进制下的每一位的值进行如下计算：

A	B	A ⊕ B
0	0	0
0	1	1
1	0	1
1	1	0

得到结果中对应位的值。

- A 的第 i 位表示 A 在二进制下从最低往最高位数第 $i+1$ 位，即 $\frac{A}{2^i}$ 除以 2 所得余数。
- 左移（<<），右移（>>）：x<<y 表示 $x \times 2^y$，x 的第 i 位为 (x<<y) 的第 $i+y$ 位，(x<<y) 的 $0 \sim (y-1)$ 位均为 0。x>>y 表示 $\frac{x}{2^y}$，x 的第 i 位（$i \geqslant y$）为 (x>>y) 的第 $i-y$ 位。

设全集为 Ω，$|\Omega| = n$，用 n 位二进制整数 x 描述 Ω 的一个子集 S，那么 x 的第 $i-1$ 位为 1 则表示 Ω 中第 i 个元素属于 S，为 0 则表示 Ω 中第 i 个元素不属于 S。基于此，对于数学中对集合之间的运算和关系的命题，位运算也有对应计算方式。

- A 与 B 交集：A&B。对于表示第 $i+1$ 个元素是否在交集中的第 i 位，该位值为 1 当且仅当 A 的第 i 位为 1 且 B 的第 i 位为 1，即 A 包含第 $i+1$ 个元素且 B 包含第 $i+1$ 个元素。
- A 与 B 并集：A|B。对于表示第 $i+1$ 个元素是否在并集中的第 i 位，该位值为 1 当且仅当 A 的第 i 位为 1 或 B 的第 i 位为 1，即 A 包含第 $i+1$ 个元素或 B 包含第 $i+1$ 个元素。
- A 的补集：设 O=((1<<n)-1)，这样有 O 的 0 到 $n-1$ 位均为 1，则 A 的补集为 O^A。对于表示第 $i+1$ 个元素是否在补集中的第 i 位，因为 O 第 i 位为 1，因此该位为 1 当且仅当 A 的第 i 位为 0，即 A 不包含第 $i+1$ 个元素。
- A/B：A^(A&B)。对于表示第 $i+1$ 个元素是否在 A/B 中的第 i 位，该位值为 1 当且仅当 A 第 i 位为 1 且 $A \cap B$ 第 i 位为 0。
- $p : A$ 是 B 的子集：(A|B)==B。若该表达式为真，那么对于第 i 位，若 B 第 i 位为 0，则 A 的第 i 位为 0，若 B 第 i 位为 1，则 A 的第 i 位可取 0/1。即第 i 个元素若不属于 B，则第 i 个元素也不属于 A，命题 p 为真。

2.3　适用问题类型

为满足 DP 的无后效性，存储对应值的索引往往需要包括可以描述该阶段状态的全部信息。而对于一类需要记录整个集合内每个元素状态的问题，朴素的对某一数组指针进行中括号运算在程序中略显乏力，设计的状态往往会冗长且在转移时拖沓，因此状态压缩优化状态就成为了更优的选择。

以一类经典的 NP 问题，旅行商问题（TSP）为例，设计传统的动态规划状态需要描述每一个点是否被到达过，因此描述需要 n 维，其中 n 为点数。n 若不为常数，以动态的维数来实现 DP 的转移本身就较为困难，而就算假设 n 为常数，例

如 6，状态也需写成 $f(0/1,0/1,0/1,0/1,0/1,0/1)$，转移时枚举转移前状态和转移后状态更是需要 7 个循环以上。而采取状态压缩，将每一个点是否被到达过表示为映射到的整数第 i 位是否为 1，则状态只需写成 $f(S)$，转移最少只需 2 个循环。

2.4　应用实例一

2.4.1　题目来源

题目名称：互不侵犯。

题目选自：NOI2009 四川省省队选拔活动。

2.4.2　题目描述

在国际象棋中，国王能攻击它上、下、左、右、左上、左下、右上、右下八个方向上附近各一个格子，共 8 个。

现在在 $n\times n$ 的国际象棋棋盘上摆放 k 个国王，使他们互不攻击，试求共有多少种摆放方案。

数据范围：$1\leqslant n\leqslant 9, 0\leqslant k\leqslant n\times n$.

2.4.3　解题思路

定义 $f[i][j][S]$ 表示从第 1 行摆放到第 i 行，已经摆放了 j 个国王，第 i 行每个格子摆放状态为 S 的合法摆放方案数。其中 S 的第 i 位为 1 则表示此行从左往右数第 $i+1$ 个格子摆放了国王，反之则没有摆放国王。

判断单行状态 S 单独出现是否合法，考虑判断 `(S&(S<<1))==0` 是否为真。若为真则代表对于任意 i，`S&(S<<1)` 的第 i 位均为 0，意味着 S 第 i 位为 1 和 S 的第 $i-1$ 位为 1 不同时成立，即没有在同一行摆放左右相邻的国王。

判断两个状态 S_1、S_2 是否可以作为合法的相邻两行出现，考虑判断 `((S1&S2)||(S1&(S2<<1))||(S1&(S2>>1)))==0` 是否为真。假设为假，那么如下三条至少存在一条为真

- `(S1&S2)` 存在一位 i 使得其值为 1，则 S_1 第 i 位为 1 且 S_2 第 i 位为 1，即上下相邻，不合法；
- `(S1&(S2<<1))` 存在一位 i 使得其值为 1，则 S_1 第 i 位为 1 且 S_2 第 $i-1$ 位为 1，即互为左下和右上的关系，不合法；
- `(S1&(S2>>1))` 存在一位 i 使得其值为 1，则 S_1 第 i 位为 1 且 S_2 第 $i+1$ 位为 1，即互为左上和右下的关系，不合法。

这三条判断和单行是否合法的判断覆盖了所有可能的不合法情况，提供了

$\mathcal{O}(1)$ 的计算方法。而 C++ 中 `__builtin_popcount(x)` 函数可以以近似 $\mathcal{O}(1)$ 的效率计算 x 二进制中 1 的个数，以下简写为 $\mathrm{pc}(x)$

转移考虑从小到大枚举 $i \in [1,n]$，枚举 $j \in [0,k]$，枚举合法单行状态 $S_1 \in [0,2^n)$、$S_2 \in [0,2^n)$。在 S_1 和 S_2 可以作为合法的相邻两行出现时，有转移

$$f[i][j+\mathrm{pc}(S_2)][S_2] := f[i][j+\mathrm{pc}(S_2)][S_2] + f[i-1][j][S_1]$$

初始状态为 $f[0][0][0]=1$，即没有填任何行时，仅在没有摆放任何国王且该行为空时存在一个方案。最终答案为

$$Ans = \sum_{S\in[0,2^n)} f[n][k][S]$$

即摆放了所有行，一共摆放了 k 个国王，最后一行为任意状态的合法方案总和。整体时间复杂度为 $\mathcal{O}(nkA+2^n)$，空间复杂度为 $\mathcal{O}(nk2^n)$，其中 A 为合法单行 S_1、S_2 组成的不同合法相邻行个数，在 $n \leqslant 9$ 时满足 $A \leqslant 683$.

2.4.4　代码实现

```cpp
#include <bits/stdc++.h>
using namespace std;
long long n, k, f[15][105][(1 << 9)];
#define pc __builtin_popcount
int main()
{
    cin >> n >> k;
    f[0][0][0] = 1;
    for(int i = 1; i <= n; i++)
    for(int S1 = 0; S1 < (1 << n); S1++)
        if((S1 & (S1 << 1)) == 0)
        for(int S2 = 0; S2 < (1 << n); S2++)
            if(((S2 & (S2 << 1)) == 0) && (((S1 & S2) || (S1 & (S2 << 1)) || (S1 & (S2 >> 1))) == 0))
                for(int j = 0; j <= k; j++)
                {
                    f[i][j + pc(S2)][S2] += f[i - 1][j][S1];
                }
    long long ans = 0;
    for (int S = 0; S < (1 << n); S++)
        ans += f[n][k][S];
    cout << ans << endl;
    return 0;
}
```

2.5　应用实例二

2.5.1　题目来源

题目名称：寿司晚宴。

题目选自：NOI2015。

2.5.2　题目描述

给定 n，设全集为 $\Omega = \{x \mid x \in [2, n]\}$.

求有多少个不同的二元组 (A, B)，满足：

- A, B 为 Ω 的子集，
- 对于任意 $x \in A, y \in B$，有 $\gcd(x, y) = 1$.

答案对 p 取模。

数据范围：$2 \leqslant n \leqslant 500$.

2.5.3　解题思路

显然不能对 500 以内的 95 个质因数进行状态压缩和转移，因为这样时空复杂度至少是 $\mathcal{O}(2^{95})$ 的，远超一般计算机算力。

但可以发现对于任意 $x \in [2, 500]$，x 分解质因数后包含的 > 19 的质因子不会超过 1 个，$\leqslant 19$ 的质因数仅 8 个，可以状态压缩。而我们只关注 Ω 中每个数分解质因数后的质因子集合，因此一个数 i 可以被表示为二元组 (p_i, S_i)，其中 p_i 表示 i 分解质因数后 > 19 的质因子，没有则为 1；S_i 表示 i 分解质因数后包含的 $\leqslant 19$ 的质因子的集合，S_i 共 8 位，第 x 位为 1 则表示从小到大第 $x + 1$ 个质数是 i 的因子，反之则不是。

按二元组第一位给 $n - 1$ 个数分组后，可以看出除 $p_i = 1$ 的组，其余组不能即有元素属于 A、又有元素属于 B。且将 A 集合中的所有元素二元组第二位按位或后，B 集合对应值与之进行按位与应为 0。

因此可以对整体设计状态，对 $p_i = 1$ 的组合 $p_i > 1$ 的组分别进行转移。

设 $f[a][b]$ 表示 A 集合中所有元素第二位按位或的值为 a，B 集合中所有元素第二位按位或的值为 b 且没有任意一个元素同时属于 A, B，没有任意一组中即有元素属于 A、又有元素属于 B 的方案数。

- 对于 $p_i = 1$ 的组，有

$$f'[a|S_i][b] \leftarrow f[a][b]$$
$$f'[a][b|S_i] \leftarrow f[a][b]$$

  其中 f' 为新的 f 数组，在枚举完 a, b 后替换 f。即 f 数组不是边枚举 a, b 边更新的。下同。

- 对于每一组，设 $\mathrm{dp}_1[a][b]$ 为当前组没有任一元素在 B 中的方案数，$\mathrm{dp}_2[a][b]$ 为当前组没有任一元素在 A 中的方案数，有

$$\mathrm{dp}_1'[a|S_i][b] \leftarrow \mathrm{dp}_1[a][b]$$
$$\mathrm{dp}_2'[a][b|S_i] \leftarrow \mathrm{dp}_2[a][b]$$

在这一组更新初，将 $\mathrm{dp}_1, \mathrm{dp}_2$ 的值赋为 f。在更新完这一组所有元素后，有

$$f'[a][b] = \mathrm{dp}_1[a][b] + \mathrm{dp}_2[a][b] - f[a][b]$$

因为这一组即没有元素在 A 中，也没有元素在 B 中会被算重。

最后有

$$Ans = \sum_{S_1 \cap S_2 = \emptyset} f[S_1][S_2]$$

整体时间复杂度为 $\mathcal{O}(n \cdot 2^{16})$，空间复杂度为 $\mathcal{O}(2^{16})$。

2.5.4　代码实现

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int maxn = 505, maxs = (1 << 8), prn = 8, pr[prn] = {2, 3, 5, 7, 11, 13, 17, 19};
int n, mod;
struct num
{
    int v, key, S;
} a[maxn];
bool cmp(num a, num b) { return a.key == b.key ? a.v < b.v : a.key < b.key; }
int f[maxs][maxs], dp1[maxs][maxs], dp2[maxs][maxs], _dp1[maxs][maxs], _dp2[maxs][maxs];
inline void upd(int &x, int y)
{
    x = x + y;
    if(x >= mod) x -= mod;
}
int main()
{
    cin >> n >> mod;
    for(int i = 1; i < n; i++)
    {
        a[i].v = a[i].key = i + 1;
        for(int j = 0; j < prn; j++)
        {
            if(!(a[i].key % pr[j])) a[i].S |= (1 << j);
            while(!(a[i].key % pr[j])) a[i].key /= pr[j];
        }
    }
    sort(a + 1, a + n, cmp);
    f[0][0] = 1;
    int st = 1;
    while(a[st].key == 1)
    {
        memset(_dp1, 0, sizeof(_dp1));
        for(int s1 = 0; s1 < (1 << prn); s1++)
            for(int s2 = 0; s2 < (1 << prn); s2++)
            {
                upd(_dp1[s1 | a[st].S][s2], f[s1][s2]);
                upd(_dp1[s1][s2 | a[st].S], f[s1][s2]);
            }
        for(int s1 = 0; s1 < (1 << prn); s1++)
            for(int s2 = 0; s2 < (1 << prn); s2++)
                upd(f[s1][s2], _dp1[s1][s2]);
        st++;
    }
    for(int i = st; i < n; i++)
    {
        if(a[i].key != a[i - 1].key)
            for(int s1 = 0; s1 < (1 << prn); s1++)
                for(int s2 = 0; s2 < (1 << prn); s2++)
                    dp1[s1][s2] = dp2[s1][s2] = f[s1][s2];
        memset(_dp1, 0, sizeof(_dp1));
        memset(_dp2, 0, sizeof(_dp2));
```

```
54          for(int s1 = 0; s1 < (1 << prn); s1++)
55              for(int s2 = 0; s2 < (1 << prn); s2++)
56              {
57                  upd(_dp1[s1 | a[i].S][s2], dp1[s1][s2]);
58                  upd(_dp2[s1][s2 | a[i].S], dp2[s1][s2]);
59              }
60          for(int s1 = 0; s1 < (1 << prn); s1++)
61              for(int s2 = 0; s2 < (1 << prn); s2++)
62                  upd(dp1[s1][s2], _dp1[s1][s2]),
63                      upd(dp2[s1][s2], _dp2[s1][s2]);
64          if(a[i].key != a[i + 1].key)
65              for(int s1 = 0; s1 < (1 << prn); s1++)
66                  for(int s2 = 0; s2 < (1 << prn); s2++)
67                      f[s1][s2] = ((dp1[s1][s2] + dp2[s1][s2]) % mod - f[s1][s2] + mod) %
    ↪  mod;
68      }
69      int ans = 0;
70      for(int s1 = 0; s1 < (1 << prn); s1++)
71          for(int s2 = 0; s2 < (1 << prn); s2++)
72              if((s1 & s2) == 0) upd(ans, f[s1][s2]);
73      cout << ans << '\n';
74      return 0;
75  }
```

第 3 章　按数位顺序进行的动态规划

本部分将介绍按数位顺序进行的动态规划（后文简称为数位 DP）

3.1　适用问题类型

给定区间 $[L, R]$，求区间内满足一定条件的数的个数。

问题有如下特征：

- 给定的条件一般与数的组成有关，比如数位之间的关系。
- 此类问题的数据规模一般较大。$1 \leqslant L < R \leqslant 10^{100000}$ 在某些情况下也是可以接受的，因为是按位 DP。
- 最终目的为计数，统计满足某一性质的数的个数。

为了方便，数位 DP 一般使用记忆化搜索实现。

3.2　算法简述

通常情况下，题目可转化为求区间 $[L, R]$ 内满足性质 p 的数的个数。利用前缀和的思想，先将求 $\text{ans}_{[L,R]}$ 的问题转化为 $\text{ans}_{[1,R]} - \text{ans}_{[1,l-1]}$.

数位 DP 的核心思想，是基于：在从 1 开始逐个向更大的数遍历的过程中，过程中有很多性质重复的部分。

比如：求 1 ~ 10000 中，存在某一数位为 7 的数的个数。在 3000 ~ 3999，4000 ~ 4999，5000 ~ 5999 中，它们的后三位都是 _000 ~ _999，是完全相同的，因此 $\text{ans}_{[3000,3999]} = \text{ans}_{[4000,4999]} = \text{ans}_{[5000,5999]}$，对这些部分加以记录，相同的部分只计算一次。这与动态规划利用数组记录已经统计过的部分，相同的部分不再重复计算的性质一致。

统计答案的过程再加上记忆过程，这就是记忆化搜索，与动态规划本质无差别，因而算法名为数位 DP。

3.3　应用实例

3.3.1　题目来源

题目名称：windy 数。

题目选自：NOI2009 四川省省队选拔活动。

3.3.2　题目描述

定义 windy 数：不含前导零且相邻两个数字之差至少为 2 的正整数。

给定 a, b，求 $[a, b]$ 中有多少个 windy 数。

数据范围：$1 \leqslant a \leqslant b \leqslant 2 \times 10^9$.

3.3.3　解题思路

设 $\text{ans}_{[a,b]}$ 代表题目所求，把 $\text{ans}_{[a,b]}$ 化为 $\text{ans}_{[a,b]} = \text{ans}_{[1,b]} - \text{ans}_{[1,a-1]}$.

下面求 $\text{ans}_{[1,r]}$，称 r 为上界。

定义四元组为动态规划状态：$(\text{pos}, \text{lst}, \text{lim}, \text{zero})$，代表的含义分别为：

- int pos：当前正在尝试确定第几位，个位是第 1 位，十位是第 2 位，以此类推。
- int lst：本题中，需要记录上一位选的是什么数，以约束当前位置的选取。
- bool lim：当前选出的前缀是否与上界相同，其用于约束当前数位的选取不能超过上界。比如上界为 273967，当前枚举出的前缀为 2739__，则 pos = 2 时只能选取 [0, 6] 的数。
- bool zero：当前选出的前缀是否都为 0. 对于上界 273967，009000 也是可以取到的，这时有 2 个前导零。这与 lim 的作用类似，都用于标记特殊情况。

容易发现，只要两个状态的四元组均相同，则他们所对应的答案均相同。

为了方便调试，代码使用记忆化搜索实现动态规划过程。

在记忆化数组的声明时，选择了只记录非特殊情况。因为当 lim = true 或 zero = true 时，状态只会出现一次，无需记录。

3.3.4　代码实现

```
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 20;
int a, b, num[MAXN];
int dp[MAXN][MAXN];
int dfs(int pos, int lst, bool lim, bool zero)
{
    if(pos == 0) return 1;
    if(!lim && !zero && dp[pos][lst] != -1) return dp[pos][lst];
    int ret = 0;
    int mx = lim ? num[pos] : 9;
    for(int i = 0; i <= mx; i++)
        if(abs(i - lst) >= 2)
            ret += dfs(pos - 1, (zero & (i == 0)) ? -2 : i,
                       lim & (i == num[pos]), zero & (i == 0));
    if(!lim && !zero) dp[pos][lst] = ret;
    return ret;
}
int solve(int x)
{
    int p = 0;
    while(x)
        num[++p] = x % 10, x /= 10;
```

```
24      memset(dp, -1, sizeof(dp));
25      return dfs(p, -2, 1, 1);
26  }
27  int main()
28  {
29      cin >> a >> b;
30      cout << solve(b) - solve(a - 1) << endl;
31      return 0;
32  }
```

第 4 章　单调队列优化动态规划

4.1　单调队列

队列是一种先进先出的数据结构，其队尾可加入元素，队首可取出或弹出元素。单调队列则是在队列的基础上，让队尾可以弹出元素以使队列中从队首到队尾相邻元素均满足定义的偏序关系。在信息学竞赛中，这样的元素多为二元组或三元组，其中一维为数组下标。

如经典问题 “滑动窗口” 中，题目给定一长度为 n 的数组 a，定义 $s_i = \min_{j\in(\max(1,i-k),i]} a_j$，要求出对于任意 $i \in [1,n], s_i$ 的值。

较为常用的解决区间最值问题的数据结构，如线段树、树状数组、ST 表等，均需要 $\mathcal{O}(n\log n)$ 的时间复杂度。

使用单调队列，则可做到 $\mathcal{O}(n)$ 的时间复杂度和空间复杂度。

具体而言，维护二元组 (a_i,i) 使队列中元素满足严格偏序关系 $\leqslant \{((x_1,y_1),(x_2,y_2)) \mid x_1 < x_2 \wedge y_1 < y_2\}$，下标 i 从小到大遍历 $[1,n]$ 时，加入元素 (a_i,i)。若加入元素前队列非空且队尾元素不满足偏序关系，则一直弹出队尾元素直到队列为空或队尾元素满足偏序关系为止。加入元素 (a_i,i) 后，在队首元素第二维 id 不满足 $id > i-k$ 时弹出队首元素直至满足条件。因为第二维加入时单调递增，第一维在队列中也单调递增，所以队列中最小值在队首。而在弹出队首操作中满足了队列中所有元素均在 $(i-k,i]$ 中，且先前在队列中且第二维属于 $(i-k,i]$ 的不会因为队首弹出操作在此之前弹出，因此队首元素 (a_h,h) 的第一维即为 s_i.

4.2　适用问题类型

使用单调队列优化 DP 时，往往是题目中有一些条件使得在没有该条件时的最优决策不合法了，因此使用单调队列排除不可能在接下来成为最优决策点的元素，保留其余元素。

4.3　应用实例一

4.3.1　题目来源

题目名称：PTA-Little Bird.

题目选自：POI2014.

4.3.2　题目描述

给定 n 颗树，第 i 颗树的高度为 h_i，有一只鸟要从第 1 颗树飞到第 n 颗树，它的初始劳累值为 0.

如果这只鸟当前在第 i 颗树，那么它接下来可以飞到 $i+1, i+2, \cdots, i+k$ 颗树。

如果它从一颗高度为 h_i 的树飞到高度为 h_j 的树，且 $h_i \leqslant h_j$，那么它的劳累值会 $+1$，反之若 $h_i > h_j$ 那么劳累值不变。

有 Q 次询问，每次询问指定 k，求这只鸟从第 1 颗树飞到第 n 颗树的劳累值最少是多少。

数据范围：$2 \leqslant n \leqslant 10^6$，$1 \leqslant Q \leqslant 25$.

4.3.3　解题思路

设 $\mathrm{dp}[i]$ 表示从第 1 颗树飞到第 i 颗树所需的最小劳累值，那么转移有

$$\mathrm{dp}[i] = \min_{j \in [\max(1, i-k), i]} (\mathrm{dp}[j] + [h[j] \leqslant h[i]])$$

其中 $[p]$ 表示若命题 p 为真则值为 1，否则为 0。这样直接转移是 $\mathcal{O}(Qn^2)$ 的，考虑优化。

设 $m_i = \min_{j \in [\max(1, i-k), i]} \mathrm{dp}[j]$，不难看出有

$$\min_{j \in [\max(1, i-k), i] \wedge \mathrm{dp}[j] = m_i} (\mathrm{dp}[j] + [h[j] \leqslant h[i]]) \leqslant \min_{j \in [\max(1, i-k), i] \wedge \mathrm{dp}[j] > m_i} (\mathrm{dp}[j] + [h[j] \leqslant h[i]])$$

因为

$$\text{左式} \leqslant \min_{j \in [\max(1, i-k), i] \wedge \mathrm{dp}[j] = m_i} (\mathrm{dp}[j] + 1) \leqslant m_i + 1$$

$$\text{右式} \geqslant \min_{j \in [\max(1, i-k), i] \wedge \mathrm{dp}[j] > m_i} (\mathrm{dp}[j]) \geqslant m_i + 1$$

于是得证。

因此考虑在每次询问时用类似“滑动窗口”的方式，$\mathcal{O}(n)$ 维护 $[\max(1, i-k), i]$ 中 dp 最小值所对应下标。考虑到要取最小值，还要 $+[h[j] \leqslant h[i]]$，对下标递增，dp 值相同的元素，在队列中需要使其 h 值递减。这样取到的队首元素就是合法转移点中，dp 值最小且 h 最大的元素。

整体时间复杂度为 $\mathcal{O}(qn)$，空间复杂度为 $\mathcal{O}(n)$.

4.3.4　代码实现

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int maxn = 1e6 + 5;
int n, Q, k, h[maxn], dp[maxn];
int q[maxn], hd, tl;
int main()
{
    cin >> n;
    for(int i = 1; i <= n; i++) cin >> h[i];
    cin >> Q;
    while(Q--)
    {
        cin >> k;
        q[hd = tl = 1] = 1;
        for(int i = 2; i <= n; i++)
        {
            while(hd <= tl && i - q[hd] > k)
                hd++;
            dp[i] = dp[q[hd]] + (h[q[hd]] <= h[i]);
            while(hd <= tl && (dp[q[tl]] > dp[i] || (dp[q[tl]] == dp[i] && h[q[tl]] <= h[i])))
                tl--;
            q[++tl] = i;
        }
        printf("%d\n", dp[n]);
    }
    return 0;
}
```

4.4　应用实例二

4.4.1　题目来源

题目名称：Pogo-Cow S.

题目选自：USACO 2013 November Contest.

4.4.2　题目描述

给定 n 个目标点，每个目标点在数轴上有一个坐标 x_i 和一个可获得分数 p_i。

定义一个长为 $|p|$ 的序列 p 合法当且仅当 x_{p_i} 关于 i 单调递增或单调递减，且 $\forall i \in [1, |p|-2], |x_{p_i} - x_{p_{i+1}}| \leqslant |x_{p_{i+1}} - x_{p_{i+2}}|$，该序列权值为 $\sum_{i\in[1,|p|]} w_{p_i}$。

试求出所有合法序列的权值的最大值。

数据范围：$1 \leqslant n \leqslant 10^3$.

4.4.3　解题思路

单调递减只需令所有 $x_i \leftarrow -x_i$ 就可以归约到单调递增的情况，因此下文将仅讨论序列中 x_{p_i} 单调递增的情况。

首先将目标点按 x 排序，记录 $\mathrm{dp}[i][\Delta x]$ 表示满足最后一个值为 i 且倒数第二个值与最后一个值对应目标点 x 差值为 Δx 的序列中序列权值的最大值，那么应

当有

$$dp[i][x_i - x_j] = \max_{j\in[1,i)\wedge x'\leqslant x_i - x_j} (\mathrm{dp}[j][x']) + p_i$$

但就算是离散地考虑 x' 一维，这样转移也是 $\mathcal{O}(n^3)$ 的，考虑优化。

发现若对于同一 i，有 $a \leqslant b$ 且 $\mathrm{dp}[i][a] \geqslant \mathrm{dp}[i][b]$，那么忽略 b 不会对任何 $\mathrm{dp}[i][x]$ 有影响。而对同一 j，在 i 单调递增的情况下，限制产生贡献的 $\mathrm{dp}[j][x]$ 的 x 值单调不减。因此考虑对每一个 i 建立一个单调队列，每个元素 $(\Delta x, \mathrm{dp}[i][\Delta x])$ 两维均单调递增。在队列内元素个数大于 1，且从队首数第二个元素第一维满足限制时，弹出队首元素。这样能够找到可能对 $\mathrm{dp}[i][x_i - x_j]$ 产生贡献的最大的满足限制 $x' \leqslant x_i - x_j$ 的 x'，对应 $\mathrm{dp}[i][x']$ 也是满足限制下的最大值。

这样一共需要 $\mathcal{O}(n)$ 个单调队列，每个队列中元素至多 $\mathcal{O}(n)$ 个，转移整体时间复杂度为 $\mathcal{O}(n^2)$，空间复杂度为 $\mathcal{O}(n^2)$。

4.4.4　代码实现

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int maxn = 1e3 + 5;
int n;
struct node
{
    int x, w;
} a[maxn];
bool cmp(node a, node b) { return a.x < b.x; }
struct Que
{
    node v[maxn];
    int tail, head;
    inline void push(node x)
    {
        while(tail && v[tail].x == x.x && v[tail].w <= x.w) tail--;
        if(x.w > v[tail].w || !tail) v[++tail] = x;
    }
    inline void pop(int x)
    {
        while(head < tail && x >= v[head + 1].x) head++;
    }
    inline void clear()
    {
        tail = head = 0;
    }
} s[maxn];
int main()
{
    cin >> n;
    for(int i = 1; i <= n; i++) cin >> a[i].x >> a[i].w;
    sort(a + 1, a + 1 + n, cmp);
    int ans = 0;
    for(int i = 1; i <= n; i++)
    {
        s[i].push((node){-1, a[i].w});
        for(int j = i - 1; j >= 1; j--)
        {
            s[j].pop(a[i].x - a[j].x);
            s[i].push((node){a[i].x - a[j].x, s[j].v[s[j].head].w + a[i].w});
        }
        ans = max(ans, s[i].v[s[i].tail].w);
    }
```

```
45      for(int i = 1; i <= n; i++) s[i].clear();
46      for(int i = n; i >= 1; i--)
47      {
48          s[i].push((node){-1, a[i].w});
49          for(int j = i + 1; j <= n; j++)
50          {
51              s[j].pop(a[j].x - a[i].x);
52              s[i].push((node){a[j].x - a[i].x, s[j].v[s[j].head].w + a[i].w});
53          }
54          ans = max(ans, s[i].v[s[i].tail].w);
55      }
56      cout << ans << '\n';
57      return 0;
58  }
```

第 5 章　斜率优化动态规划

5.1　斜率优化

斜率优化可以在将转移式变形后，通过考察几何意义，将可能成为最优转移点的集合缩小到一个凸包上，再根据题目条件将转移的时间复杂度缩小到 $\mathcal{O}(n)$ 或 $\mathcal{O}(n\log n)$。

5.2　适用问题类型

可以应用斜率优化的动态规划模型往往某一维度的状态数为 $\mathcal{O}(n)$ 级别，而为找到最优转移点，单次的状态转移需考察 $\mathcal{O}(n)$ 个子阶段，这使该维度的转移的时间复杂度开销达到 $\mathcal{O}(n^2)$。

5.3　应用实例

5.3.1　题目来源

题目名称：玩具装箱。

题目选自：NOI2008 湖南省省队选拔活动。

5.3.2　题目描述

有 n 个玩具，第 i 个玩具价值为 c_i。要求将这 n 个玩具排成一排，分成若干段。对于一段 $[l,r]$，它的代价为：

$$(r-l+\sum_{i=l}^{r}c_i-L)^2$$

求分段的最小代价。

数据范围：$1\leqslant n\leqslant 5\times 10^4, 1\leqslant L, 0\leqslant c_i\leqslant 10^7$.

5.3.3　解题思路

令 f_i 表示前 i 个物品，分若干段的最小代价。有状态转移方程：

$$f_i=\min_{j<i}\{f_j+(pre_i-pre_j+i-j-1-L)^2\}$$

其中 $pre_i=\sum_{j=1}^{i}c_j$。直接转移时间复杂度为 $O(n^2)$，无法解决本题。

为简化状态转移方程式，令 $s_i = pre_i + i, L' = L + 1$，则

$$f_i = \min_{j<i}\{f_j + (s_i - s_j - L')^2\}$$

设 j 为使 f_i 最小的转移点，有：

$$f_i = f_j + (s_i - s_j - L')^2$$

考虑一次函数的斜截式 $y = kx + b$，将方程转化为这个形式。

其中变量 x, y 与 j 有关，b, k 与 i 有关，且要求 x 随 j 单调递增，仅 b 中包含 f_i。

按照上面的规则，对方程进行整理得到：

$$f_j + (s_j + L')^2 = 2s_i(s_j + L) + f_i - {s_i}^2$$

$$\begin{cases} y = f_j + (s_j + L')^2 \\ k = 2s_i \\ x = s_j + L \\ b = f_i - {s_i}^2 \end{cases}$$

(x_j, y_j) 的几何意义为直线 $y = kx + b$ 上的一个点，又因为转移时目的是最小化 f_i，在上面的表示当中，f_i 只与直线的截距 b 有关。所以问题可转化为如何选取 j，使得过点 (x_j, y_j) 的直线的截距 b 最小。

注意到直线的斜率不变。如图 5.1，相当于平移直线 $y = kx$，直到其经过图中的一个点。

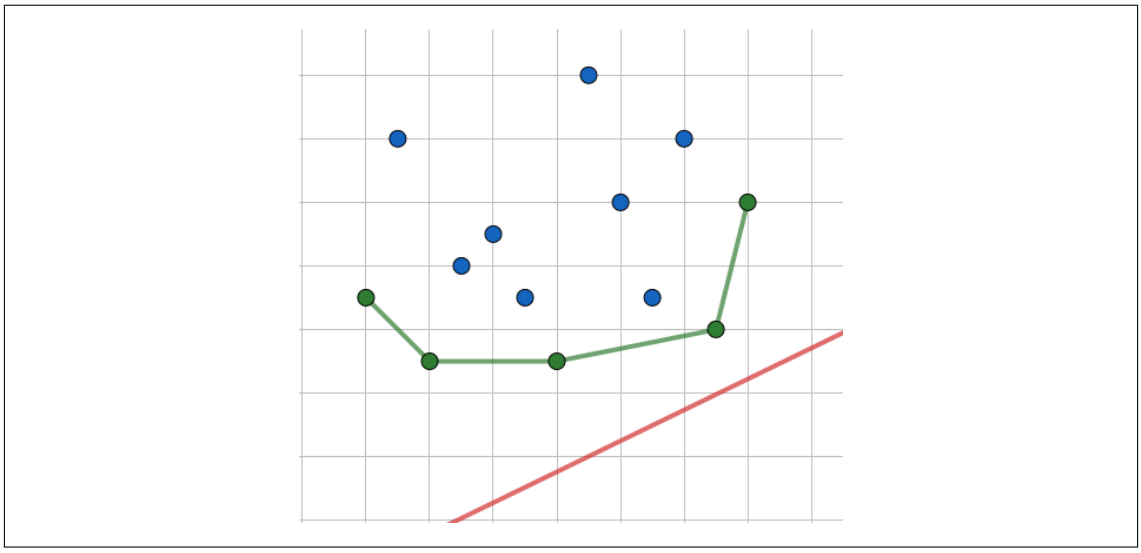

图 5.1

于是可以在转移的同时维护 (x_i, y_i) 构成的凸包，利用单调性二分得到时间复杂度为 $\mathcal{O}(n \log n)$ 的算法。发现 $k = 2s_i$ 关于 i 单调递增，因此可以利用单调队列来维护得到时间复杂度为 $\mathcal{O}(n)$ 的算法。

5.3.4　代码实现

```cpp
#include <bits/stdc++.h>
#define int long long
using namespace std;
const int MAXN = 5e4 + 10;
int n, L;
int c[MAXN], s[MAXN], f[MAXN];
double slope(int i, int j)
{
    return ((f[j] + (s[j] + L) * (s[j] + L)) - (f[i] + (s[i] + L) * (s[i] + L))) / (double)(s[j] - s[i]);
}
int head, tail, q[MAXN];
signed main()
{
    scanf("%lld%lld", &n, &L);
    L += 1;
    for(int i = 1; i <= n; i++)
    {
        scanf("%lld", &c[i]);
        s[i] = s[i - 1] + c[i] + 1;
    }
    head = tail = 1;
    for(int i = 1; i <= n; i++)
    {
        while(head < tail && slope(q[head], q[head + 1]) <= 2 * s[i])
        {
            head++;
        }
        f[i] = f[q[head]] + (s[i] - s[q[head]] - L) * (s[i] - s[q[head]] - L);
        while(head < tail && slope(q[tail - 1], q[tail]) >= slope(q[tail], i))
        {
            tail--;
        }
        q[++tail] = i;
    }
    printf("%lld", f[n]);
    return 0;
}
```

第 6 章　四边形不等式优化动态规划

6.1　四边形不等式

若对于任意整数 l_1, l_2, r_2, r_1 满足 $l_1 \leqslant l_2 \leqslant r_2 \leqslant r_1$，若二元函数 $w(x, y)$ 满足

$$w(l_1, r_1) + w(l_2, r_2) \geqslant w(l_1, r_2) + w(l_2, r_1) \tag{1}$$

则称 w 满足四边形不等式。四边形不等式有一个经典的等价形式：

$$w(l, r+1) + w(l+1, r) \geqslant w(l, r) + w(l+1, r+1) \tag{2}$$

其中 $l < r$。由（1）得到（2）较为显然，由（2）推出（1）可以通过数学归纳法证得。

假设对于 $l + x < r$ 有

$$w(l, r+1) + w(l+x, r) \geqslant w(l, r) + w(l+x, r+1)$$

在 $x = 1$ 时此式为（1）。

对于 $l + x + 1 < r$，根据（2）有

$$w(l+x, r+1) + w(l+x+1, r) \geqslant w(l+x, r) + w(l+x+1, r+1)$$

两式相加可得

$$w(l, r+1) + w(l+x+1, r) \geqslant w(l, r) + w(l+x+1, r+1)$$

同理可得 $w(l, r+y) + w(l+x, r) \geqslant w(l, r) + w(l+x, r+y)$，即证。

6.2　四边形不等式对一维 DP 的优化

在优化一维 DP 时，DP 的形式往往为 $f[i] = \min\limits_{0 \leqslant j < i} \{f[j] + w(j, i)\}$。不妨设令 $f[i]$ 取得最小值的 j 为 $d[i]$，那么若 w 满足四边形不等式，则 d 单调不减，称 f 具有决策单调性。利用此性质可以使时间复杂度为 $\mathcal{O}(n^2)$ 的计算简化为 $\mathcal{O}(n \log n)$。本文将不对这类优化进行细致探讨。

6.3　四边形不等式对二维 DP 的优化

在优化二维 DP 时，DP 的形式则多为 $f[i][j] = \min\limits_{i \leqslant k < j} \{f[i][k] + f[k+1][j] + w(i, j)\} (i < j)$，$f[i][i] = 0$。

不妨设令 $f[i][j]$ 取得最小值的 k 为 $d[i][j]$，那么

- 若
  1. w 满足四边形不等式，
  2. 对于任意 $l_1 \leqslant l_2 \leqslant r_2 \leqslant r_1$，有 $w(l_1, r_1) \geqslant w(l_2, r_2)$，
- 则
  1. f 也满足四边形不等式，
  2. $d[i][j] \leqslant d[i+1][j]$ 且 $d[i][j] \leqslant d[i][j+1]$.

对于满足 $d[i][j] \leqslant \min\{d[i+1][j], d[i][j+1]\}$ 的，我们称 dp 具有二维决策单调性。在对 $f[i][j]$ 进行计算时，仅考察位于 $[d[i][j-1], d[i+1][j]]$ 中的 k 可以使时间复杂度为 $\mathcal{O}(n^3)$ 的计算优化为 $\mathcal{O}(n^2)$.

6.3.1　对满足四边形不等式的证明

关于 f 满足四边形不等式的证明，考虑使用数学归纳法。

当 $r - l = 1$ 时，

$$f[l][r+1] + f[l+1][r] = f[l][l+2] \geqslant w(l, l+2)$$

$$f[l][r] + f[l+1][r+1] = f[l][l+1] + f[l+1][l+2] = w(l, l+1) + w(l+1, l+2)$$

$$f[l][r+1] + f[l+1][r] \geqslant f[l][r] + f[l+1][r+1]$$

设在 $r - l < k$ 时，f 满足四边形不等式。

为方便起见，设 $x = d[l][r+1], y = d[l+1][r]$，那么：

$$f[l][r+1]+f[l+1][r] = (f[l][x]+f[x+1][r+1]+w(l,r+1))+(f[l+1][y]+f[y+1][r]+w(l+1,r))$$

$$f[l][r]+f[l+1][r+1] \leqslant (f[l][x]+f[x+1][r]+w(l,r))+(f[l+1][y]+f[y+1][r+1]+w(l+1,r+1))$$

欲证:

$$f[l][r+1] + f[l+1][r] \geqslant f[l][r] + f[l+1][r+1]$$

只需证：

$$f[x+1][r+1]+f[y+1][r]+w(l,r+1)+w(l+1,r) \geqslant f[x+1][r]+f[y+1][r+1]+w(l,r)+w(l+1,r+1)$$

只需证：

$$f[x+1][r+1] + f[y+1][r] \geqslant f[x+1][r] + f[y+1][r+1]$$

$$w(l, r+1) + w(l+1, r) \geqslant w(l, r) + w(l+1, r+1)$$

两式分别可以由归纳假设以及 w 满足四边形不等式得到，即证 f 满足四边形不等式。

6.3.2　对二维决策单调性的证明

关于二维决策单调性的证明，设 $x = d[i][j]$，$i \leqslant k < x$。根据 f 满足四边形不等式和 $d[i][j]$ 的最优性，有

$$f[i][x] + f[i+1][k] \geqslant f[i][k] + f[i+1][x] f[i][k] + f[k+1][j] \geqslant f[i][x] + f[x+1][j]$$

可以得到

$$f[i+1][k] + f[k+1][j] \geqslant f[i+1][x] + f[x+1][j]$$

因此对于 $f[i+1][j]$ 的转移，$k \in [i, d[i][j])$ 一定不优于 $d[i][j]$，有 $d[i+1][j] \geqslant d[i][j]$。类似的，设 $x < k \leqslant j$，有

$$f[x+1][j]+f[k+1][j-1] \geqslant f[x+1][j-1]+f[k+1][j] f[i][k]+f[k+1][j] \geqslant f[i][x]+f[x+1][j]$$

可以得到

$$f[i][k] + f[k+1][j-1] \geqslant f[i][x] + f[x+1][j-1]$$

即 $d[i][j-1] \leqslant d[i][j]$.

6.3.3　对时间复杂度的证明

关于时间复杂度是 $\mathcal{O}(n^2)$ 的证明，考虑

$$\begin{aligned}
&\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (d[i+1][j] - d[i][j-1]) \\
=&\sum_{i=2}^{n} \sum_{j=i+1}^{n} d[i][j] - \sum_{i=1}^{n-1} \sum_{j=i}^{n-1} d[i][j] \\
=&\sum_{i=1}^{n-1} d[i][n] - \sum_{j=2}^{n} d[1][j] - \sum_{i=1}^{n} d[i][i]
\end{aligned}$$

并且 $d[i][j] \in [1, n]$，因此 $\mathcal{O}(\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (d[i+1][j] - d[i][j-1])) = \mathcal{O}(n^2)$.

6.4　应用实例一

6.4.1　题目描述

有 n 堆直线排列的石子，第 i 堆石子有 $a[i]$ 个。

规定每次只能合并任意相邻的两堆石子，并产生两堆石子数量之和的疲劳值。现在要将石子有序的合并成一堆，试求最小总疲劳值。

数据范围：$1 \leqslant n \leqslant 5000$.

6.4.2　解题思路

设 $\mathrm{dp}[i][j]$ 为将下标在区间 $[i,j]$ 内的石子合并为一堆所需的最小疲劳值，$w(i,j)=\sum_{k=i}^{j}a[k]$，那么有

$$\mathrm{dp}[i][j]=\begin{cases}\min\limits_{i\leqslant k<j}\{\mathrm{dp}[i][k]+\mathrm{dp}[k+1][j]+w(i,j)\}, & i<j\\ 0, & i=j\\ +\infty, & i>j\end{cases}$$

直接转移时间复杂度为 $\mathcal{O}(n^3)$，不能接受。发现 w 满足

$$w(l,r+1)+w(l+1,r)=w(l,r)+w(l+1,r+1)$$

即复合四边形不等式条件，因此可以对上述 DP 进行优化，达到 $\mathcal{O}(n^2)$ 的时间复杂度。

6.4.3　代码实现

```cpp
#include <bits/stdc++.h>
using namespace std;
const int maxn = 5005;
int n, a[maxn], s[maxn], dp[maxn][maxn], d[maxn][maxn];
#define w(i, j) (s[j] - s[i - 1])
int main()
{
    cin >> n;
    for(int i = 1; i <= n; i++)
        cin >> a[i];
    for(int i = 1; i <= n; i++)
        s[i] = s[i - 1] + a[i];
    memset(dp, 0x3f, sizeof(dp));
    for(int i = 1; i <= n; i++)
        dp[i][i] = 0, d[i][i] = i;
    for(int l = 2; l <= n; l++)
    {
        for(int i = 1, j; i + l - 1 <= n; i++)
        {
            j = i + l - 1;
            for(int k = d[i][j - 1]; k <= min(j - 1, d[i + 1][j]); k++)
            {
                int nx = dp[i][k] + dp[k + 1][j] + w(i, j);
                if(nx < dp[i][j])
                {
                    dp[i][j] = nx;
                    d[i][j] = k;
                }
            }
        }
    }
    cout << dp[1][n] << endl;
    return 0;
}
```

6.5　应用实例二

6.5.1　题目来源

题目名称：邮局。

题目选自：IOI2000.

6.5.2　题目描述

给定 n 个村庄在一条直线上的坐标。现在要选一些村庄建立 k 个邮局，使得每个村庄与其最近的邮局之间的距离总和最小。试求这个最小距离和。

数据范围：用 p_i 表示村庄坐标，则 $1 \leqslant k \leqslant 300$，$k \leqslant n \leqslant 3000$，$1 \leqslant p_i \leqslant 10000$.

6.5.3　解题思路

首先将村庄按其在坐标轴上的位置排序，设第 i 个村庄的坐标为 x_i。

设 $\mathrm{dp}[i][j]$ 表示前 i 个村庄建立 j 个邮局的最小距离和，$w(l,r)$ 表示在第 p 个村庄建立一个邮局所得到的 $\sum_{i=l}^{r}|x_i-x_p|$ 的最小值。不难看出 $p=\lfloor\frac{l+r}{2}\rfloor$，因此

$$w(l,r)=(\sum_{i=p+1}^{r}x_i-x_p(r-p))+(x_p(p-l)-\sum_{i=l}^{p-1}x_i)$$

$$w(l,r)=w(l,r-1)+x_r-x_p$$

$$w(l,r)=w(l+1,r)+x_p-x_l$$

可以在 DP 转移前 $\mathcal{O}(n^2)$ 递推，也可以使用前缀和每次转移时 $\mathcal{O}(1)$ 计算。

同时 DP 递推式有

$$\mathrm{dp}[i][j]=\begin{cases}\min\limits_{1\leqslant k\leqslant i}\{\mathrm{dp}[k-1][j-1]+w(k,i)\}, & \text{if } i\neq 0, j\neq 0\\ 1, & \text{if } i=0 j=0\\ 0, & \text{otherwise}\end{cases}$$

直接递推是 $\mathcal{O}(n^2k)$ 的，不能接受。发现

$$w(l,r)+w(l+1,r-1)=2w(l+1,r-1)+x_r-x_l$$

$$w(l+1,r)+w(l,r-1)=2w(l+1,r-1)+x_r-x_l-(x_{\lfloor\frac{l+r+1}{2}\rfloor}-x_{\lfloor\frac{l+r-1}{2}\rfloor})$$

因为 x 递增，有 $(x_{\lfloor\frac{l+r+1}{2}\rfloor}-x_{\lfloor\frac{l+r-1}{2}\rfloor})\geqslant 0$，所以 $w(l,r)+w(l+1,r-1)\geqslant w(l+1,r)+w(l,r-1)$，$w$ 满足四边形不等式，dp 满足四边形不等式。因此对转移时计算的区间进行优化就能达到 $\mathcal{O}(nk)$ 的时间复杂度。

6.5.4　代码实现

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int maxn = 3e3 + 5, maxp = 305;
int n, p, dp[maxn][maxp], d[maxn][maxp], x[maxn], s[maxn];
inline int w(int l, int r)
{
    int mid = l + r >> 1;
    int res = s[r] - s[mid] - x[mid] * (r - mid);
    res += x[mid] * (mid - l) - (s[mid - 1] - s[l - 1]);
    return res;
}
int main()
{
    cin >> n >> p;
    for(int i = 1; i <= n; i++) cin >> x[i];
    sort(x + 1, x + 1 + n);
    for(int i = 1; i <= n; i++) s[i] = s[i - 1] + x[i];
    for(int i = 1; i <= p; i++) d[n + 1][i] = n;
    memset(dp, 0x3f, sizeof(dp));
    dp[0][0] = 0;
    for(int j = 1; j <= p; j++)
        for(int i = n; i >= 1; i--)
            for(int k = d[i][j - 1]; k <= d[i + 1][j]; k++)
            {
                int nx = dp[k][j - 1] + w(k + 1, i);
                if(nx < dp[i][j])
                    dp[i][j] = nx, d[i][j] = k;
            }
    cout << dp[n][p] << '\n';
    return 0;
}
```

第 7 章　CDQ 分治优化 DP

7.1　CDQ 分治

CDQ 分治最早为陈丹琦在《从 <cash> 谈一类分治算法的应用》中整理的，因此以 CDQ 命名。

陈丹琦本人整理的 CDQ 分治应用方法主要为先将在线维护的问题转换为离线问题，并添加一个操作时间的维度，再对高维的偏序问题进行分治处理。后来引申的 CDQ 分治应用范围则包括了可离线的高维偏序问题以及转移包括高维偏序条件的动态规划问题。

下文将探讨 CDQ 分治对转移包括高维偏序条件的动态规划问题的优化。

7.2　应用实例一

7.2.1　题目描述

给定一个长度为 n 的排列 a，其中有若干位置上的数字已经确定了，剩下位置上的数字不确定。

你需要钦定未被确定的位置上的数字，使得得到的排列的最长上升子序列（LIS，Longest Increasing Subsequence）长度尽量长。试求出这个最长长度。

数据范围：$n \leqslant 10^5$.

7.2.2　解题思路

注意到 “是一个排列” 这个限制，事实上仅仅限制了那些未被确定的位置上的数字不能与已经确定的位置相同，而我们在计算最优答案时并不需要特别处理未被确定的数字之间是否相同，因为 LIS 中不能出现两个相同的数字，因此剩下位置上出现相同数字一定不优。

设 $f(i,j)$ 表示对于 a 数组长度为 i 的前缀，在所有钦定方案中，以 j 结尾的最长的 LIS 长度。有转移：

- 若位置 i 被确定了，则 $f(i,x)=\begin{cases}\max_{j<p_i}(f(i-1,j))+1 & x=p_i\\ 0 & x\neq p_i\end{cases}$；
- 否则 $f(i,j)=\max\{\max_{k<j}(f(i-1,k))+1,f(i-1,j)\}$，这里的 j 必须不在已经确定的数字中出现过。

这样直接转移是 $\mathcal{O}(n^2)$ 的。

但是二维 dp 是不方便使用 CDQ 分治优化的，仍然考虑转成一维 dp 的形式。

设 $f(i)$ 表示以位置 i 结尾的最长 LIS，此处位置 i 的数字必须被确定了。

则枚举 LIS 中上一个有确定数字的位置。用 $\mathrm{cnt}[i]$ 表示 a 长度为 i 的前缀中没被确定的位置个数，$\mathrm{rem}[p_i]$ 表示 $< p_i$ 且未出现在被确定的位置的数字个数，可以得到：

$$f(i) = \max_{p_j<p_i}\{f(j) + \min\{\mathrm{cnt}[i] - \mathrm{cnt}[j], \mathrm{rem}[p_i] - \mathrm{rem}[p_j]\}\} + 1$$

注意到这里同时有 $p_j < p_i$ 和 $j < i$ 两个限制，且中间的 min 可以分类讨论为两种情况：

1. $\mathrm{cnt}[i] - \mathrm{cnt}[j] \geqslant \mathrm{rem}[p_i] - \mathrm{rem}[p_j] \iff \mathrm{cnt}[i] - \mathrm{rem}[p_i] \geqslant \mathrm{cnt}[j] - \mathrm{rem}[p_j]$
2. $cnt[i] - cnt[j] < \mathrm{rem}[p_i] - \mathrm{rem}[p_j] \iff cnt[i] - \mathrm{rem}[p_i] < cnt[j] - \mathrm{rem}[p_j]$

这也就相当于第三个限制，加上前两个就是三维偏序。因此我们先按下标分治，然后左右两边分别按 $\mathrm{cnt}[i] - \mathrm{rem}[p_i]$ 排序，以第一种情况为例，双指针扫的时候用树状数组维护 $f(j) - \mathrm{rem}[p_j]$ 即可。

整体时间复杂度 $\mathcal{O}(n \log^2 n)$，空间复杂度 $\mathcal{O}(n)$。

7.2.3　代码实现

```
// a[i].x=id,a[i].y=cnt[i]-rem[p[i]],a[i].tp=(p[i]>0);
void cdq(int l, int r)
{
    if(l >= r) return;
    int mid = (l + r) >> 1;
    cdq(l, mid);
    sort(a + l, a + mid + 1, cmpy);
    sort(a + mid + 1, a + r + 1, cmpy);
    int i = l, j = mid + 1;
    for(; j <= r; j++)
    {
        if(!a[j].tp) continue;
        while(i <= mid && a[i].y <= a[j].y)
        {
            if(a[i].tp) upd(a[i].p, a[i].f - rem[a[i].p]);
            i++;
        }
        a[j].f = max(a[j].f, qry(a[j].p) + rem[a[j].p] + 1);
    }
    for(j = l; j < i; j++)
        if(a[j].tp) clr(a[j].p);
    i = mid;
    j = r;
    for(; j > mid; j--)
    {
        while(i >= l && a[i].y > a[j].y)
        {
            if(a[i].tp) upd(a[i].p, a[i].f - cnt[a[i].p]);
            i++;
        }
        a[j].f = max(a[j].f, qry(a[j].p) + cnt[a[j].p] + 1);
    }
    for(j = mid; j > i; j--)
        if(a[j].tp) clr(a[j].p);
    sort(a + l, a + r + 1, cmpx);
    cdq(mid + 1, r);
}
```

7.3　应用实例二

7.3.1　题目来源

题目名称：Building Bridges.

题目选自：CEOI2017.

7.3.2　题目描述

给定 n 个柱子，每个柱子用 (h_i, w_i) 描述。拆除第 i 根柱子的代价为 w_i。在 i, j 之间架桥的代价为 $(h_i - h_j)^2$，同时还需要拆除 $[i+1, j-1]$ 之间的所有柱子。求通过桥梁把 1 和 n 连通的最小代价。

数据范围：$2 \leqslant n \leqslant 10^5$，$0 \leqslant h_i, |w_i| \leqslant 10^6$.

7.3.3　解题思路

不妨设 $\mathrm{dp}[i]$ 为通过桥梁将 1 和 i 联通的最小代价，$s_i = \sum_{i \in [1,i]} w_i$，不难得到

$$\mathrm{dp}[i] = \min_{j<i}(dp[j] + s_{i-1} - s_j + (h_i - h_j)^2)$$

将 min 去掉可以得到

$$(\mathrm{dp}[j] - s_j + h_j^2) = h_i \times (2h_j) - (s_{i-1} + h_i^2) + \mathrm{dp}[i]$$

发现能斜率优化，然而直接进行斜率优化需要满足斜率 $2h_j$ 单调，因此考虑 CDQ 分治。具体来讲，以第一维为下标，计算 $\mathrm{cdq}(l, r)$ 时，令 $\mathrm{mid} = \lfloor \frac{l+r}{2} \rfloor$，对 $[l, \mathrm{mid}]$ 中的下标 i 直接计算出由 $(\mathrm{dp}[i] - s_i + h_i^2, h_i)$ 构成的下凸包，$[\mathrm{mid}+1, r]$ 中的元素按照 h 排序然后做斜率优化 DP 计算贡献即可。

7.3.4　代码实现

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
const int maxn = 1e5 + 5;
const double eps = 1e-6;
inline int read()
{
    int x = 0, f = 1, ch = getchar();
    while(ch < '0' || ch > '9')
    {
        if(ch == '-') f = -1;
        ch = getchar();
    }
    while(ch >= '0' && ch <= '9')
    {
        x = (x << 3) + (x << 1) + ch - 48;
        ch = getchar();
    }
    return x * f;
}
int n;
ll f[maxn], s[maxn];
```

```cpp
struct node
{
    int h, id;
    ll s;
} a[maxn];
bool cmph(node a, node b) { return a.h < b.h; }
bool cmpid(node a, node b) { return a.id < b.id; }
int q[maxn], hd = 1, tl;
double slope(int u, int v)
{
    ll uy = f[a[u].id] - a[u].s + (1ll) * a[u].h * a[u].h;
    ll vy = f[a[v].id] - a[v].s + (1ll) * a[v].h * a[v].h;
    if(a[u].h == a[v].h)
    {
        return vy > uy ? 1e18 : -1e18;
    }
    return (uy - vy) * 1.0 / (a[u].h - a[v].h);
}
void cdq(int l, int r)
{
    if(l >= r) return;
    int mid = l + r >> 1;
    cdq(l, mid);
    sort(a + l, a + mid + 1, cmph);
    sort(a + mid + 1, a + r + 1, cmph);
    //斜率优化
    for(int i = l; i <= mid; i++)
    {
        while(tl > 1 && slope(q[tl], i) < slope(q[tl - 1], q[tl])) tl--;
        q[++tl] = i;
    }
    for(int j = mid + 1; j <= r; j++)
    {
        while(hd < tl && slope(q[hd], q[hd + 1]) <= 2.0 * a[j].h + eps) hd++;
        if(hd <= tl) f[a[j].id] = min(f[a[j].id],
            s[a[j].id - 1] + (1ll) * a[j].h * a[j].h + f[a[q[hd]].id] - a[q[hd]].s +
    ↪  (1ll) * a[q[hd]].h * a[q[hd]].h - 2ll * a[j].h * a[q[hd]].h);
    }
    hd = 1, tl = 0;
    sort(a + mid + 1, a + r + 1, cmpid);
    cdq(mid + 1, r);
}
int main()
{
    n = read();
    for(int i = 1; i <= n; i++) a[i].h = read(), a[i].id = i;
    for(int i = 1; i <= n; i++) a[i].s = a[i - 1].s + read(), s[i] = a[i].s;
    memset(f, 0x3f, sizeof(f));
    f[1] = 0;
    cdq(1, n);
    cout << f[n] << '\n';
    return 0;
}
```

第 8 章　结语

在解决动态规划问题时，人们常常会受算法难度和思路的限制。经过此次学习，本组详细整理并掌握了两种复杂的动态规划算法以及四种可以被广泛应用的算法优化方式，形成文字化资料，可以被后人利用。而其中本组所附上的典例解析可以作为参考资料，有利于后人对于该算法的学习，可以更好地理解并应用该算法。

通过本次课题研究，我们对文献法的掌握更加深入。本组在后续的成文过程中也参考了各资料的成文方式，使我们的文章深入浅出，更为简洁易懂。本小组成员各司其职，分工合理。这使得我们的研学速度快，成果质量高。我们的心态也逐渐变得稳健，铸就了迎难而上，不放弃的精神。

本次研学我们的创新意识稍有不足，主要进行了资料的整合而非独立，全新的创作。以后需加以改进。

参考文献