

## 目录 (Table of Contents)

### 专题算法

---

LCA问题 . . . . .	3
RMQ问题 . . . . .	5
回文串 . . . . .	12
树的重心 . . . . .	14
离线LCA . . . . .	16

### 其他算法

---

CDQ分治 . . . . .	18
三分法 . . . . .	50
二分答案 . . . . .	52
位运算 . . . . .	54
分数规划 . . . . .	56
括号序列 . . . . .	62
整体二分 . . . . .	64
模拟退火 . . . . .	77
离散化 . . . . .	81
离线算法 . . . . .	83
表达式求值 . . . . .	85
随机化算法 . . . . .	94

### 动态规划

---

凸包优化DP . . . . .	103
分治优化DP . . . . .	105
动态规划基础 . . . . .	107
区间动态规划 . . . . .	114
单调队列优化DP . . . . .	117
四边形不等式优化DP . . . . .	119
序列动态规划 . . . . .	121
数位动态规划 . . . . .	123
最长公共子序列 . . . . .	125
最长递增子序列 . . . . .	127
树形动态规划 . . . . .	129
概率动态规划 . . . . .	136

## 专题算法

---

LCA问题	3
RMQ问题	5
回文串	12
树的重心	14
离线LCA	16

## 其他算法

---

CDQ分治	18
三分法	50
二分答案	52
位运算	54
分数规划	56
括号序列	62
整体二分	64
模拟退火	77
离散化	81
离线算法	83
表达式求值	85
随机化算法	94

## 动态规划

---

凸包优化DP	103
分治优化DP	105
动态规划基础	107
区间动态规划	114
单调队列优化DP	117
四边形不等式优化DP	119
序列动态规划	121
数位动态规划	123
最长公共子序列	125
最长递增子序列	127
树形动态规划	129
概率动态规划	136



# 404 Not Found

🔧 本页面最近更新：无更新，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：(自动生成)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用



# RMQ

## 简介

RMQ 是英文 Range Maximum/Minimum Query 的缩写，表示区间最大（最小）值。

在接下来的描述中，默认初始数组大小为  $n$ ，询问次数为  $m$ 。

在接下来的描述中，默认时间复杂度标记方式为  $O(A) \sim O(B)$ ，其中  $O(A)$  表示预处理时间复杂度，而  $O(B)$  表示单次询问的时间复杂度。

## 单调栈

由于 **OI Wiki** 中已有此部分的描述，本文仅给出 [链接](#)。这部分不再展开。

时间复杂度  $O(m \log m) \sim O(\log n)$ ，空间复杂度  $O(n)$ 。

## ST 表

由于 **OI Wiki** 中已有此部分的描述，本文仅给出 [链接](#)。这部分不再展开。

时间复杂度  $O(n \log n) \sim O(1)$ ，空间复杂度  $O(n \log n)$ 。

## 线段树

由于 **OI Wiki** 中已有此部分的描述，本文仅给出 [链接](#)。这部分不再展开。

时间复杂度  $O(n) \sim O(\log n)$ ，空间复杂度  $O(n)$ 。

## Four Russian

Four russian 是一个由四位俄罗斯籍的计算机科学家提出来的基于 ST 表的算法。

在 ST 表的基础上 Four russian 算法对其做出的改进是序列分块。

具体来说，我们将原数组——我们将其称之为数组 A——每  $S$  个分成一块，总共  $n/S$  块。

对于每一块我们预处理出来块内元素的最小值，建立一个长度为  $n/S$  的数组 B，并对数组 B 采用 ST 表的方式预处理。

同时，我们对于数组 A 的每一个零散块也建立一个 ST 表。

询问的时候，我们可以将询问区间划分为不超过 1 个数组 B 上的连续块区间和不超过 2 个数组 A 上的整块内的连续区间。显然这些问题我们通过 ST 表上的区间查询解决。

在  $S = \log n$  时候，预处理复杂度达到最优，为

$$O((n / \log n) \log n + (n / \log n) \times \log n \times \log \log n) = O(n \log \log n)。$$

时间复杂度  $O(n \log \log n) \sim O(1)$ ，空间复杂度  $O(n \log \log n)$ 。

当然询问由于要跑三个 ST 表，该实现方法的常数较大。

### 一些小小的算法改进

我们发现，在询问的两个端点在数组 A 中属于不同的块的时候，数组 A 中块内的询问是关于每一块前缀或者后缀的询问。

显然这些询问可以通过预处理答案在  $O(n)$  的时间复杂度内被解决。

这样子我们只需要在询问的时候进行至多一次 ST 表上的查询操作了。

### 一些玄学的算法改进

由于 Four russian 算法以 ST 表为基础，而算法竞赛一般没有非常高的时间复杂度要求，所以 Four russian 算法一般都可以被 ST 表代替，在算法竞赛中并不实用。这里提供一种在算法竞赛中更加实用的 Four russian 改进算法。

我们将块大小设为  $\sqrt{n}$ ，然后预处理出每一块内前缀和后缀的 RMQ，再暴力预处理出任意连续的整块之间的 RMQ，时间复杂度为  $O(n)$ 。

查询时，对于左右端点不在同一块内的询问，我们可以直接  $O(1)$  得到左端点所在块的后缀 RMQ，左端点和右端点之间的连续整块 RMQ，和右端点所在块的前缀 RMQ，答案即为三者之间的最值。

而对于左右端点在同一块内的询问，我们可以暴力求出两点之间的 RMQ，时间复杂度为  $O(\sqrt{n})$ ，但是单个询问的左右端点在同一块内的期望为  $O(\frac{\sqrt{n}}{n})$ ，所以这种方法的时间复杂度为期望  $O(n)$ 。

而在算法竞赛中，我们并不用非常担心出题人卡掉这种算法，因为我们可以随机微调块大小，很大程度上避免算法在根据特定块大小构造的数据中出现最坏情况。并且如果出题人想要卡掉这种方法，则暴力有可能可以通过。

这是一种期望时间复杂度达到下界，并且代码实现难度和算法常数均较小的算法，因此在算法竞赛中比较实用。

以上做法参考了 [P3793 由乃救爷爷](#) 中的题解。

## 加减 1RMQ

若序列满足相邻两元素相差为 1，在这个序列上做 RMQ 可以成为加减 1RMQ，根据这个特性可以改进 Four Russian 算法，做到  $O(n) \sim O(1)$  的时间复杂度， $O(n)$  的空间复杂度。

由于 Four Russian 算法的瓶颈在于块内 RMQ 问题，我们重点去讨论块内 RMQ 问题的优化。

由于相邻两个数字的差值为  $\pm 1$ ，所以在固定左端点数字时 长度不超过  $\log n$  的右侧序列种类数为  $\sum_{i=1}^{\log n} 2^{i-1}$ ，而这个式子显然不超过  $n$ 。

这启示我们可以预处理所有不超过  $n$  种情况的 最小值 - 第一个元素 的值。

在预处理的时候我们需要去预处理同一块内相邻两个数字之间的差，并且使用二进制将其表示出来。

在询问的时候我们找到询问区间对应的二进制表示，查表得出答案。

这样子 Four Russian 预处理的时间复杂度就被优化到了  $O(n)$ 。

## 笛卡尔树在 RMQ 上的应用

不了解笛卡尔树的朋友请移步 [笛卡尔树](#)。

不难发现，原序列上两个点之间的 min/max，等于笛卡尔树上两个点的 LCA 的权值。根据这一点就可以借助  $O(n) \sim O(1)$  求解树上两个点之间的 LCA 进而求解 RMQ。 $O(n) \sim O(1)$  树上 LCA 在 [LCA - 标准 RMQ](#) 已经有描述，这里不再展开。

总结一下，笛卡尔树在 RMQ 上的应用，就是通过将普通 RMQ 问题转化为 LCA 问题，进而转化为加减 1 RMQ 问题进行求解，时间复杂度为  $O(n) \sim O(1)$ 。当然由于转化步数较多， $O(n) \sim O(1)$  RMQ 常数较大。

如果数据随机，还可以暴力在笛卡尔树上查找。此时的时间复杂度为期望  $O(n) \sim O(\log n)$ ，并且实际使用时这种算法的常数往往很小。

## 例题 [Luogu P3865 【模板】ST 表](#)

## 基于状压的线性 RMQ 算法

### 隐性要求

- 序列的长度  $n$  满足  $\log_2 n \leq 64$ 。

### 前置知识

- Sparse Table
- 基本位运算
- 前后缀极值

## 算法原理

将原序列  $A[1 \dots n]$  分成每块长度为  $O(\log_2 n)$  的  $O(\frac{n}{\log_2 n})$  块。

听说令块长为  $1.5 \times \log_2 n$  时常数较小。

记录每块的最大值，并用 ST 表维护块间最大值，复杂度  $O(n)$ 。

记录块中每个位置的前、后缀最大值  $Pre[1 \dots n], Sub[1 \dots n]$  ( $Pre[i]$  即  $A[i]$  到其所在块的块首的最大值)，复杂度  $O(n)$ 。

若查询的  $l, r$  在两个不同块上，分别记为第  $bl, br$  块，则最大值为  $[bl + 1, br - 1]$  块间的最大值，以及  $Sub[l]$  和  $Pre[r]$  这三个数的较大值。

现在的问题在于若  $l, r$  在同一块中怎么办。

将  $A[1 \dots r]$  依次插入单调栈中，记录下标和值，满足值从栈底到栈顶递减，则  $A[l, r]$  中的最大值为从栈底往上，单调栈中第一个满足其下标  $p \geq l$  的值。

由于  $A[p]$  是  $A[l, r]$  中的最大值，因而在插入  $A[p]$  时， $A[l \dots p - 1]$  都被弹出，且在插入  $A[p + 1 \dots r]$  时不可能将  $A[p]$  弹出。

而如果用 0/1 表示每个数是否在栈中，就可以用整数状压，则  $p$  为第  $l$  位后的第一个 1 的位置。

由于块大小为  $O(\log_2 n)$ ，因而最多不超过 64 位，可以用一个整数存下（即隐性条件的原因）。

## 参考代码

```
1 #include <algorithm>
2 #include <cmath>
3 #include <cstdio>
4
5 constexpr int MAXN = 1e5 + 5;
6 constexpr int MAXM = 20;
7
8 struct RMQ {
9     int N, A[MAXN];
10    int blockSize;
11    int S[MAXN][MAXM], Pow[MAXM], Log[MAXN];
12    int Belong[MAXN], Pos[MAXN];
13    int Pre[MAXN], Sub[MAXN];
14    int F[MAXN];
15
16    void buildST() {
17        int cur = 0, id = 1;
18        Pos[0] = -1;
19        for (int i = 1; i <= N; ++i) {
20            S[id][0] = std::max(S[id][0], A[i]);
21            Belong[i] = id;
22            if (Belong[i - 1] != Belong[i])
23                Pos[i] = 0;
24            else
25                Pos[i] = Pos[i - 1] + 1;
26            if (++cur == blockSize) {
27                cur = 0;
28                ++id;
29            }
30        }
31        if (N % blockSize == 0) --id;
32        Pow[0] = 1;
33        for (int i = 1; i < MAXM; ++i) Pow[i] = Pow[i - 1] * 2;
34        for (int i = 2; i <= id; ++i) Log[i] = Log[i / 2] + 1;
35        for (int i = 1; i <= Log[id]; ++i) {
36            for (int j = 1; j + Pow[i] - 1 <= id; ++j) {
37                S[j][i] = std::max(S[j][i - 1], S[j + Pow[i - 1]][i - 1]);
38            }
39        }
40    }
41
42    void buildSubPre() {
43        for (int i = 1; i <= N; ++i) {
44            if (Belong[i] != Belong[i - 1])
45                Pre[i] = A[i];
46            else
47                Pre[i] = std::max(Pre[i - 1], A[i]);
48        }
49    }
}
```

```

50     for (int i = N; i >= 1; --i) {
51         if (Belong[i] != Belong[i + 1])
52             Sub[i] = A[i];
53         else
54             Sub[i] = std::max(Sub[i + 1], A[i]);
55     }
56 }
57
58 void buildBlock() {
59     static int S[MAXN], top;
60     for (int i = 1; i <= N; ++i) {
61         if (Belong[i] != Belong[i - 1])
62             top = 0;
63         else
64             F[i] = F[i - 1];
65         while (top > 0 && A[S[top]] <= A[i]) F[i] |= ~(1 << Pos[S[top--]]);
66         S[++top] = i;
67         F[i] |= (1 << Pos[i]);
68     }
69 }
70 }
71
72 void init() {
73     for (int i = 1; i <= N; ++i) scanf("%d", &A[i]);
74     blockSize = log2(N) * 1.5;
75     buildST();
76     buildSubPre();
77     buildBlock();
78 }
79
80 int queryMax(int l, int r) {
81     int bl = Belong[l], br = Belong[r];
82     if (bl != br) {
83         int ans1 = 0;
84         if (br - bl > 1) {
85             int p = Log[br - bl - 1];
86             ans1 = std::max(S[bl + 1][p], S[br - Pow[p]][p]);
87         }
88         int ans2 = std::max(Sub[l], Pre[r]);
89         return std::max(ans1, ans2);
90     } else {
91         return A[l + __builtin_ctz(F[r] >> Pos[l])];
92     }
93 }
94 } R;
95
96 int M;
97
98 int main() {
99     scanf("%d%d", &R.N, &M);
100    R.init();
101    for (int i = 0, l, r; i < M; ++i) {

```

```
102     scanf( "%d%d", &l, &r);
103     printf("%d\n", R.queryMax(l, r));
104 }
return 0;
}
```

## 习题

[BJOI 2020] 封印：SAM+RMQ

🕒 本页面最近更新：2024/10/9 22:38:42，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：[StudyingFather](#), [Ir1d](#), [zhouyuyang2002](#), [Enter-tainer](#), [kfy666](#), [Backl1ght](#), [billchenchina](#), [Chrogeek](#), [countercurrent-time](#), [diauweb](#), [Henry-ZHR](#), [hsfzLZH1](#), [ksyx](#), [Mooos-MoSheng](#), [orzAtalod](#), [ouuan](#), [ranwen](#), [SkqLiao](#), [sshwy](#), [Tiphereth-A](#), [Xeonacid](#), [zzjbb](#)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用



# 404 Not Found

🔧 本页面最近更新：无更新，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：(自动生成)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用





# 404 Not Found

🔧 本页面最近更新：无更新，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：(自动生成)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用





# 404 Not Found

🔧 本页面最近更新：无更新，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：(自动生成)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用





# CDQ 分治

本页面将介绍 CDQ 分治。

## 简介

CDQ 分治是一种思想而不是具体的算法，与 [动态规划](#) 类似。目前这个思想的拓展十分广泛，依原理与写法的不同，大致分为三类：

- 解决和点对有关的问题。
- 1D 动态规划的优化与转移。
- 通过 CDQ 分治，将一些动态问题转化为静态问题。

| CDQ 分治的思想最早由 IOI2008 金牌得主陈丹琦在高中时整理并总结，它也因此得名。<sup>1</sup>

## 解决和点对有关的问题

这类问题多数类似于「给定一个长度为  $n$  的序列，统计有一些特性的点对  $(i, j)$  的数量/找到一对点  $(i, j)$  使得一些函数的值最大」。

CDQ 分治解决这类问题的算法流程如下：

1. 找到这个序列的中点  $mid$ ；
2. 将所有点对  $(i, j)$  划分为 3 类：
  - a.  $1 \leq i \leq mid, 1 \leq j \leq mid$  的点对；
  - b.  $1 \leq i \leq mid, mid + 1 \leq j \leq n$  的点对；
  - c.  $mid + 1 \leq i \leq n, mid + 1 \leq j \leq n$  的点对。
3. 将  $(1, n)$  这个序列拆成两个序列  $(1, mid)$  和  $(mid + 1, n)$ 。此时第一类点对和第三类点对都在这两个序列之中；
4. 递归地处理这两类点对；
5. 设法处理第二类点对。

可以看到 CDQ 分治的思想就是不断地把点对通过递归的方式分给左右两个区间。

在实际应用时，我们通常使用一个函数 `solve(l, r)` 处理  $l \leq i \leq r, l \leq j \leq r$  的点对。上述算法流程中的递归部分便是通过 `solve(l, mid)` 与 `solve(mid, r)` 来实现的。剩下的第二类点对则需要额外设计算法解决。

例題

## 三维偏序

给定一个序列，每个点有  $a_i, b_i, c_i$  三个属性，试求：这个序列里有多少对点对  $(i, j)$  满足  $a_j \leq a_i$  且  $b_j \leq b_i$  且  $c_j \leq c_i$  且  $j \neq i$ 。

## 解题思路

三维偏序是 CDQ 分治的经典问题。

题目要求统计序列里点对的个数，那试一下用 CDQ 分治。

首先将序列按  $a$  排序。

假设我们现在写好了 `solve(l, r)`，并且通过递归搞定了 `solve(l, mid)` 和 `solve(mid+1, r)`。现在我们要做的，就是统计满足  $l \leq i \leq mid, mid + 1 \leq j \leq r$  的点对  $(i, j)$  中，有多个点对还满足  $a_i \leq a_j, b_i \leq b_j, c_i \leq c_j$  的限制条件。

稍微思考一下就会发现，那个  $a_i \leq a_j$  的限制条件没啥用了：既然  $i$  比  $mid$  小， $j$  比  $mid$  大，那  $i$  肯定比  $j$  要小；已经将序列按  $a$  排序，就一定有  $a_i \leq a_j$ 。现在还剩下两个限制条件： $b_i \leq b_j$  与  $c_i \leq c_j$ 。根据这个限制条件我们就可以枚举  $j$ ，求出有多少个满足条件的  $i$ 。

为了方便枚举，我们把  $(l, mid)$  和  $(mid + 1, r)$  中的点全部按照  $b$  的值从小到大排个序。之后我们依次枚举每一个  $j$ ，把所有  $b_i \leq b_j$  的点  $i$  全部插入到某种数据结构里（这里我们选择 [树状数组](#)）。此时只要查询树状数组里有多少个点的  $c$  值是小于等于  $c_j$  的，我们就求出了对于这个点  $j$ ，有多少个  $i$  可以合法匹配它了。

当我们插入一个  $c$  值等于  $x$  的点时，我们就令树状数组的  $x$  这个位置单点加一，而查询树状数组里有多少个点小于  $x$  的操作实际上就是在求 [前缀和](#)，只要我们事先对于所有的  $c$  值做了 [离散化](#)，我们的复杂度就是对的。

对于每一个  $j$ ，我们都需要将所有  $b_i \leq b_j$  的点  $i$  插入树状数组中。由于所有的  $i$  和  $j$  都已事先按照  $b$  值排好序，这样的话只要以双指针的方式在树状数组里插入点，则对树状数组的插入操作就能从  $O(n^2)$  次降到  $O(n)$  次。

通过这样一个算法流程，我们就用  $O(n \log n)$  的时间处理完了关于第二类点对的信息了。此时算法的时间复杂度是  $T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + O(n \log n) = O(n \log^2 n)$ 。

## 示例代码

```
1 #include <algorithm>
2 #include <iostream>
3
4 constexpr int MAXN = 1e5 + 10;
5 constexpr int MAXK = 2e5 + 10;
6
7 int n, k;
8
9 struct Element {
10     int a, b, c;
11     int cnt;
12     int res;
13
14     bool operator!=(Element other) const {
15         if (a != other.a) return true;
16         if (b != other.b) return true;
17         if (c != other.c) return true;
18         return false;
19     }
20 };
21
22 Element e[MAXN];
23 Element ue[MAXN];
24 int m, t;
25 int res[MAXN];
26
27 struct BinaryIndexedTree {
28     int node[MAXK];
29
30     int lowbit(int x) { return x & -x; }
31
32     void Add(int pos, int val) {
33         while (pos <= k) {
34             node[pos] += val;
35             pos += lowbit(pos);
36         }
37         return;
38     }
39
40     int Ask(int pos) {
41         int res = 0;
42         while (pos) {
43             res += node[pos];
44             pos -= lowbit(pos);
45         }
46         return res;
47     }
48 } BIT;
49
```

```

50  bool cmpA(Element x, Element y) {
51      if (x.a != y.a) return x.a < y.a;
52      if (x.b != y.b) return x.b < y.b;
53      return x.c < y.c;
54  }
55
56  bool cmpB(Element x, Element y) {
57      if (x.b != y.b) return x.b < y.b;
58      return x.c < y.c;
59  }
60
61  void CDQ(int l, int r) {
62      if (l == r) return;
63      int mid = (l + r) / 2;
64      CDQ(l, mid);
65      CDQ(mid + 1, r);
66      std::sort(ue + l, ue + mid + 1, cmpB);
67      std::sort(ue + mid + 1, ue + r + 1, cmpB);
68      int i = l;
69      int j = mid + 1;
70      while (j <= r) {
71          while (i <= mid && ue[i].b <= ue[j].b) {
72              BIT.Add(ue[i].c, ue[i].cnt);
73              i++;
74          }
75          ue[j].res += BIT.Ask(ue[j].c);
76          j++;
77      }
78      for (int k = l; k < i; k++) BIT.Add(ue[k].c, -ue[k].cnt);
79      return;
80  }
81
82  using std::cin;
83  using std::cout;
84
85  int main() {
86      cin.tie(nullptr)->sync_with_stdio(false);
87      cin >> n >> k;
88      for (int i = 1; i <= n; i++) cin >> e[i].a >> e[i].b >>
e[i].c;
89      std::sort(e + 1, e + n + 1, cmpA);
90      for (int i = 1; i <= n; i++) {
91          t++;
92          if (e[i] != e[i + 1]) {
93              m++;
94              ue[m].a = e[i].a;
95              ue[m].b = e[i].b;
96              ue[m].c = e[i].c;
97              ue[m].cnt = t;
98              t = 0;
99          }
100     }
101 }

```

```
102     CDQ(1, m);
103     for (int i = 1; i <= m; i++) res[ue[i].res + ue[i].cnt - 1]
104     += ue[i].cnt;
105     for (int i = 0; i < n; i++) cout << res[i] << '\n';
        return 0;
    }
```



## CQOI2011 动态逆序对



对于序列  $a$ , 它的逆序对数定义为集合  $\{(i, j) | i < j \wedge a_i > a_j\}$  中的元素个数。

现在给出  $1 \sim n$  的一个排列, 按照某种顺序依次删除  $m$  个元素, 你的任务是在每次删除一个元素之前统计整个序列的逆序对数。

## 示例代码

```
1 // 仔细推一下就是和三维偏序差不多的式子了，基本就是一个三维偏序的
2 板子
3 #include <algorithm>
4 #include <iostream>
5 using namespace std;
6 using ll = long long;
7 int n;
8 int m;
9
10 struct treearray {
11     int ta[200010];
12
13     void ub(int& x) { x += x & (-x); }
14
15     void db(int& x) { x -= x & (-x); }
16
17     void c(int x, int t) {
18         for (; x <= n + 1; ub(x)) ta[x] += t;
19     }
20
21     int sum(int x) {
22         int r = 0;
23         for (; x > 0; db(x)) r += ta[x];
24         return r;
25     }
26 } ta;
27
28 struct data_ {
29     int val;
30     int del;
31     int ans;
32 } a[100010];
33
34 int rv[100010];
35 ll res;
36
37 // 重写两个比较
38 bool cmp1(const data_& a, const data_& b) { return a.val < b.val; }
39
40 bool cmp2(const data_& a, const data_& b) { return a.del < b.del; }
41
42 void solve(int l, int r) { // 底下是具体的式子，套用
43     if (r - l == 1) {
44         return;
45     }
46     int mid = (l + r) / 2;
47     solve(l, mid);
48 }
```

```

50     solve(mid, r);
51     int i = l + 1;
52     int j = mid + 1;
53     while (i <= mid) {
54         while (a[i].val > a[j].val && j <= r) {
55             ta.c(a[j].del, 1);
56             j++;
57         }
58         a[i].ans += ta.sum(m + 1) - ta.sum(a[i].del);
59         i++;
60     }
61     i = l + 1;
62     j = mid + 1;
63     while (i <= mid) {
64         while (a[i].val > a[j].val && j <= r) {
65             ta.c(a[j].del, -1);
66             j++;
67         }
68         i++;
69     }
70     i = mid;
71     j = r;
72     while (j > mid) {
73         while (a[j].val < a[i].val && i > l) {
74             ta.c(a[i].del, 1);
75             i--;
76         }
77         a[j].ans += ta.sum(m + 1) - ta.sum(a[j].del);
78         j--;
79     }
80     i = mid;
81     j = r;
82     while (j > mid) {
83         while (a[j].val < a[i].val && i > l) {
84             ta.c(a[i].del, -1);
85             i--;
86         }
87         j--;
88     }
89     sort(a + l + 1, a + r + 1, cmp1);
90     return;
91 }
92
93 int main() {
94     cin.tie(nullptr)->sync_with_stdio(false);
95     cin >> n >> m;
96     for (int i = 1; i <= n; i++) {
97         cin >> a[i].val;
98         rv[a[i].val] = i;
99     }
100    for (int i = 1; i <= m; i++) {
101        int p;

```

```

102     cin >> p;
103     a[rv[p]].del = i;
104 }
105 for (int i = 1; i <= n; i++) {
106     if (a[i].del == 0) a[i].del = m + 1;
107 }
108 for (int i = 1; i <= n; i++) {
109     res += ta.sum(n + 1) - ta.sum(a[i].val);
110     ta.c(a[i].val, 1);
111 }
112 for (int i = 1; i <= n; i++) {
113     ta.c(a[i].val, -1);
114 }
115 solve(0, n);
116 sort(a + 1, a + n + 1, cmp2);
117 for (int i = 1; i <= m; i++) {
118     cout << res << '\n';
119     res -= a[i].ans;
}
return 0;
}

```

## CDQ 分治优化 1D/1D 动态规划的转移

相关内容：[CDQ 分治优化 DP](#)

1D/1D 动态规划指的是一类特定的 DP 问题，该类题目的特征是 DP 数组是一维的，转移是  $O(n)$  的。如果条件良好的话，有时可以通过 CDQ 分治来把它们的时间复杂度由  $O(n^2)$  降至  $O(n \log^2 n)$ 。

例如，给定一个序列，每个元素有两个属性  $a, b$ 。我们希望计算一个 DP 式子的值，它的转移方程如下：

$$dp_i = 1 + \max_{j=1}^{i-1} dp_j[a_j < a_i][b_j < b_i]$$

这是一个二维最长上升子序列的 DP 方程，即只有  $j < i, a_j < a_i, b_j < b_i$  的点  $j$  可以更新点  $i$  的 DP 值。

直接转移显然是  $O(n^2)$  的。以下是使用 CDQ 分治优化转移过程的讲解。

我们发现  $dp_j$  转移到  $dp_i$  这种转移关系也是一种点对间的关系，所以我们用类似 CDQ 分治处理点对关系的方式来处理它。

这个转移过程相对来讲比较套路。假设现在正在处理的区间是  $(l, r)$ ，算法流程大致如下：

1. 如果  $l = r$ ，说明  $dp_r$  值已经被计算好了。直接令  $dp_r ++$  然后返回即可；
2. 递归使用 `solve(l, mid);`；
3. 处理所有  $l \leq j \leq mid, mid + 1 \leq i \leq r$  的转移关系；

#### 4. 递归使用 `solve(mid+1, r)`。

第三步的做法与 CDQ 分治求三维偏序差不多。处理  $l \leq j \leq mid$ ,  $mid + 1 \leq i \leq r$  的转移关系的时候，我们会发现已经不用管  $j < i$  这个限制条件了。因此，我们依然先将所有的点  $i$  和点  $j$  按  $a$  值进行排序处理，然后用双指针的方式将  $j$  点插入到树状数组里，最后查一下前缀最大值更新一下  $dp_i$  就可以了。

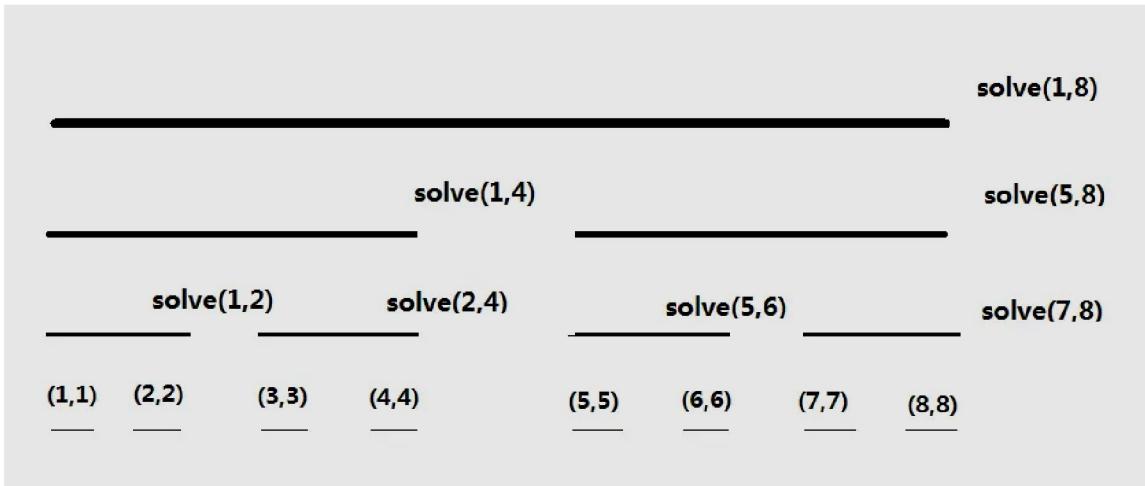
### 转移过程的正确性证明

该 CDQ 写法和处理点对间关系的 CDQ 写法最大的不同就是处理  $l \leq j \leq mid$ ,  $mid + 1 \leq i \leq r$  的点对这一部分。处理点对间关系的 CDQ 写法中，这一部分放到哪里都是可以的。但是，在用 CDQ 分治优化 DP 的时候，这个流程却必须夹在 `solve(l, mid), solve(mid + 1, r)` 的中间。原因是 DP 的转移是 **有序的**，它必须满足两个条件，否则就是不对的：

1. 用来计算  $dp_i$  的所有  $dp_j$  值都必须是已经计算完毕的，不能存在「半成品」；
2. 用来计算  $dp_i$  的所有  $dp_j$  值都必须能更新到  $dp_i$ ，不能存在没有更新到的  $dp_j$  值。

上述两个条件可能在  $O(n^2)$  暴力的时候是相当容易满足的，但是使用 CDQ 分治后，转移顺序很显然已经乱掉了，所以有必要考察转移的正确性。

CDQ 分治的递归树如下所示。



执行刚才的算法流程的话，以 8 这个点为例，它的 DP 值是在 `solve(1,8)`、`solve(5,8)`、`solve(7,8)` 这 3 个函数中更新完成的，而三次用来更新它的点分别是  $(1, 4)$ 、 $(5, 6)$ 、 $(7, 7)$  这三个不相交的区间；又以 5 这个点为例，它的 DP 值是在 `solve(1,4)` 函数中解决的，更新它的区间是  $(1, 4)$ 。仔细观察就会发现，一个  $i$  点的 DP 值被更新了  $\log$  次，而且，更新它的区间刚好是  $(1, i)$  在线段树上被拆分出来的  $\log$  个区间。因此，我们的确保证了所有合法的  $j$  都更新过点  $i$ ，满足第 2 个条件。

接着分析我们算法的执行流程：

1. 第一个结束的函数是 `solve(1,1)`。此时我们发现  $dp_1$  的值已经计算完毕了；

2. 第一个执行转移过程的函数是 `solve(1,2)`。此时我们发现  $dp_2$  的值已经被转移好了；
3. 第二个结束的函数是 `solve(2,2)`。此时我们发现  $dp_2$  的值已经计算完毕了；
4. 接下来 `solve(1,2)` 结束， $(1,2)$  这段区间的  $dp$  值均被计算好；
5. 下一个执行转移流程的函数是 `solve(1,4)`。这次转移结束之后我们发现  $dp_3$  的值已经被转移好了；
6. 接下来结束的函数是 `solve(3,3)`。我们会发现  $dp_3$  的  $dp$  值被计算好了；
7. 接下来执行的转移是 `solve(2,4)`。此时  $dp_4$  在 `solve(1,4)` 中被  $(1,2)$  转移了一次，这次又被  $(3,3)$  转移了，因此  $dp_4$  的值也被转移好了；
8. `solve(4,4)` 结束， $dp_4$  的值计算完毕；
9. `solve(3,4)` 结束， $(3,4)$  的值计算完毕；
10. `solve(1,4)` 结束， $(1,4)$  的值计算完毕。
11. .....

通过模拟函数流程，我们发现一件事：每次 `solve(l,r)` 结束的时候， $(l,r)$  区间的 DP 值会被全部计算好。由于我们每一次执行转移函数的时候，`solve(l,mid)` 已经结束，因此我们每一次执行的转移过程都是合法的，满足第 1 个条件。

在刚才的过程我们发现，如果将 CDQ 分治的递归树看成一颗线段树，那么 CDQ 分治就是这个线段树的 **中序遍历函数**，因此我们相当于按顺序处理了所有的 DP 值，只是转移顺序被拆开了而已，所以算法是正确的。

## 例题



## SDOI2011 拦截导弹



某国为了防御敌国的导弹袭击，发展出一种导弹拦截系统。但是这种导弹拦截系统有一个缺陷：虽然它的第一发炮弹能够到达任意的高度、并且能够拦截任意速度的导弹，但是以后每一发炮弹都不能高于前一发的高度，其拦截的导弹的飞行速度也不能大于前一发。某天，雷达捕捉到敌国的导弹来袭。由于该系统还在试用阶段，所以只有一套系统，因此有可能不能拦截所有的导弹。

在不能拦截所有的导弹的情况下，我们当然要选择使国家损失最小、也就是拦截导弹的数量最多的方案。但是拦截导弹数量的最多的方案有可能有多个，如果有多个最优方案，那么我们会随机选取一个作为最终的拦截导弹行动蓝图。

我方间谍已经获取了所有敌军导弹的高度和速度，你的任务是计算出在执行上述决策时，每枚导弹被拦截掉的概率。

## 参考代码

```
1 // 一道二维最长上升子序列的题
2 // 为了确定某一个元素是否在最长上升子序列中可以正反跑两遍 CDQ
3 #include <algorithm>
4 #include <iomanip>
5 #include <iostream>
6 using namespace std;
7 using db = double;
8 constexpr int N = 1e6 + 10;
9
10 struct data_ {
11     int h;
12     int v;
13     int p;
14     int ma;
15     db ca;
16 } a[2][N];
17
18 int n;
19 bool tr;
20
21 // 底下是重写比较
22 bool cmp1(const data_& a, const data_& b) {
23     if (tr)
24         return a.h > b.h;
25     else
26         return a.h < b.h;
27 }
28
29 bool cmp2(const data_& a, const data_& b) {
30     if (tr)
31         return a.v > b.v;
32     else
33         return a.v < b.v;
34 }
35
36 bool cmp3(const data_& a, const data_& b) {
37     if (tr)
38         return a.p < b.p;
39     else
40         return a.p > b.p;
41 }
42
43 bool cmp4(const data_& a, const data_& b) { return a.v ==
44 b.v; }
45
46 struct treearray {
47     int ma[2 * N];
48     db ca[2 * N];
49 }
```

```

50 void c(int x, int t, db c) {
51     for (; x <= n; x += x & (-x)) {
52         if (ma[x] == t) {
53             ca[x] += c;
54         } else if (ma[x] < t) {
55             ca[x] = c;
56             ma[x] = t;
57         }
58     }
59 }
60
61 void d(int x) {
62     for (; x <= n; x += x & (-x)) {
63         ma[x] = 0;
64         ca[x] = 0;
65     }
66 }
67
68 void q(int x, int& m, db& c) {
69     for (; x > 0; x -= x & (-x)) {
70         if (ma[x] == m) {
71             c += ca[x];
72         } else if (m < ma[x]) {
73             c = ca[x];
74             m = ma[x];
75         }
76     }
77 }
78 } ta;
79
80 int rk[2][N];
81
82 void solve(int l, int r, int t) { // 递归跑
83     if (r - l == 1) {
84         return;
85     }
86     int mid = (l + r) / 2;
87     solve(l, mid, t);
88     sort(a[t] + mid + 1, a[t] + r + 1, cmp1);
89     int p = l + 1;
90     for (int i = mid + 1; i <= r; i++) {
91         for (; (cmp1(a[t][p], a[t][i]) || a[t][p].h == a[t][i].h)
92 && p <= mid;
93             p++) {
94             ta.c(a[t][p].v, a[t][p].ma, a[t][p].ca);
95         }
96         db c = 0;
97         int m = 0;
98         ta.q(a[t][i].v, m, c);
99         if (a[t][i].ma < m + 1) {
100             a[t][i].ma = m + 1;
101             a[t][i].ca = c;

```

```

102     } else if (a[t][i].ma == m + 1) {
103         a[t][i].ca += c;
104     }
105 }
106 for (int i = l + 1; i <= mid; i++) {
107     ta.d(a[t][i].v);
108 }
109 sort(a[t] + mid, a[t] + r + 1, cmp3);
110 solve(mid, r, t);
111 sort(a[t] + l + 1, a[t] + r + 1, cmp1);
112 }
113
114 void ih(int t) {
115     sort(a[t] + 1, a[t] + n + 1, cmp2);
116     rk[t][1] = 1;
117     for (int i = 2; i <= n; i++) {
118         rk[t][i] = (cmp4(a[t][i], a[t][i - 1])) ? rk[t][i - 1] :
119 i;
120     }
121     for (int i = 1; i <= n; i++) {
122         a[t][i].v = rk[t][i];
123     }
124     sort(a[t] + 1, a[t] + n + 1, cmp3);
125     for (int i = 1; i <= n; i++) {
126         a[t][i].ma = 1;
127         a[t][i].ca = 1;
128     }
129 }
130
131 int len;
132 db ans;
133
134 int main() {
135     cin.tie(nullptr)->sync_with_stdio(false);
136     cin >> n;
137     for (int i = 1; i <= n; i++) {
138         cin >> a[0][i].h >> a[0][i].v;
139         a[0][i].p = i;
140         a[1][i].h = a[0][i].h;
141         a[1][i].v = a[0][i].v;
142         a[1][i].p = i;
143     }
144     ih(0);
145     solve(0, n, 0);
146     tr = true;
147     ih(1);
148     solve(0, n, 1);
149     tr = true;
150     sort(a[0] + 1, a[0] + n + 1, cmp3);
151     sort(a[1] + 1, a[1] + n + 1, cmp3);
152     for (int i = 1; i <= n; i++) {
153         len = max(len, a[0][i].ma);

```

```

154     }
155     cout << len << '\n';
156     for (int i = 1; i <= n; i++) {
157         if (a[0][i].ma == len) {
158             ans += a[0][i].ca;
159         }
160     }
161     cout << fixed << setprecision(5);
162     for (int i = 1; i <= n; i++) {
163         if (a[0][i].ma + a[1][i].ma - 1 == len) {
164             cout << (a[0][i].ca * a[1][i].ca) / ans << ' ';
165         } else {
166             cout << "0.00000 ";
167         }
168     }
169     return 0;
}

```

## 将动态问题转化为静态问题

前两种情况使用 CDQ 分治的目的是将序列折半之后递归处理点对间的关系，来获得良好的复杂度。不过在本节中，折半的不是一般的序列，而是时间序列。

它适用于一些「需要支持做  $xxx$  修改然后做  $xxx$  询问」的数据结构题。该类题目有两个特点：

- 如果把询问 [离线](#)，所有操作会按照时间自然地排成一个序列。
- 每一个修改均与之后的询问操作息息相关。而这样的「修改 - 询问」关系一共会有  $O(n^2)$  对。

我们可以使用 CDQ 分治对于这个操作序列进行分治，处理修改和询问之间的关系。

与处理点对关系的 CDQ 分治类似，假设正在分治的序列是  $(l, r)$ ，我们先递归地处理  $(l, mid)$  和  $(mid, r)$  之间的修改 - 询问关系，再处理所有  $l \leq i \leq mid, mid + 1 \leq j \leq r$  的修改 - 询问关系，其中  $i$  是一个修改， $j$  是一个询问。

注意，如果各个修改之间是 [独立](#) 的话，我们无需处理  $l \leq i \leq mid$  和  $mid + 1 \leq j \leq r$ ，以及 `solve(l, mid)` 和 `solve(mid+1, r)` 之间的时序关系（比如普通的加减法问题）。但是如果各个修改之间并不独立（比如说赋值操作），做完这个修改后，序列长什么样可能依赖于之前的序列。此时处理所有跨越  $mid$  的修改 - 询问关系的步骤就必须放在 `solve(l, mid)` 和 `solve(mid+1, r)` 之间。理由和 CDQ 分治优化 1D/1D 动态规划的原因是一样的：按照中序遍历序进行分治才能保证每一个修改都是严格按照时间顺序执行的。

## 例题

## 矩形加矩形求和

维护一个二维平面，然后支持在一个矩形区域内加一个数字，每次询问一个矩形区域的和。

### 解题思路

对于这个问题的静态版本，即「二维平面里有一堆矩形，我们希望询问一个矩形区域的和」，有一个经典做法叫线段树 + 扫描线。具体的做法是先将每个矩形拆成插入和删除两个操作，接着将每个询问拆成两个前缀和相减的形式，最后离线。然而，原题目是动态的，不能直接使用这种做法。

尝试对其使用 CDQ 分治。我们将所有的询问和修改操作全部离线。这些操作形成了一个序列，并且有  $O(N^2)$  对修改 - 询问的关系。依然使用 CDQ 分治的一般流程，将所有的关系分成三类，在这一层分治过程当中只处理跨越  $mid$  的修改 - 询问关系，剩下的修改 - 询问关系通过递归的方式解决。

我们发现，所有的修改在询问之前就已完成。这时，原问题等价于「平面上有静态的一堆矩形，不停地询问一个矩形区域的和」。

使用一个扫描线在  $O(n \log n)$  的时间内处理好所有跨越  $mid$  的修改 - 询问关系，剩下的事情就是递归地分治左右两侧的修改 - 询问关系了。

在这样实现的 CDQ 分治中，同一个询问被处理了  $O(\log n)$  次。不过没关系，因为每次贡献这个询问的修改是互不相交的。全套流程的时间复杂度为

$$T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + O(n \log n) = O(n \log^2 n).$$

观察上述的算法流程，我们发现一开始我们只能解决静态的矩形加矩形求和问题，但只是简单地使用 CDQ 分治后，我们就可以离线地解决一个动态的矩形加矩形求和问题了。将动态问题转化为静态问题的精髓就在于 CDQ 分治每次仅仅处理跨越某一个点的修改和询问关系，这样的话我们就只需要考虑「所有询问都在修改之后」这个简单的问题了。也正是因为这一点，CDQ 分治被称为「动态问题转化为静态问题的工具」。



## [Ynoi2016] 镜中的昆虫



维护一个长为  $n$  的序列  $a_i$ , 有  $m$  次操作。

1. 将区间  $[l, r]$  的值修改为  $x$ ;
2. 询问区间  $[l, r]$  出现了多少种不同的数, 也就是说同一个数出现多次只算一个。



## 解题思路



一句话题意: 区间赋值区间数颜色。

维护一下每个位置左侧第一个同色点的位置, 记为  $pre_i$ , 此时区间数颜色就被转化为了一个经典的二维数点问题。

通过将连续的一段颜色看成一个点的方式, 可以证明  $pre$  的变化量是  $O(n + m)$  的, 即单次操作仅仅引起  $O(1)$  的  $pre$  值变化, 那么我们可以用 CDQ 分治来解决动态的单点加矩形求和问题。

$pre$  数组的具体变化可以使用 `std::set` 来进行处理。这个用 `set` 维护连续的区间的技巧也被称之为 [old driver tree](#)。

## 参考代码

```
1 #include <algorithm>
2 #include <iostream>
3 #include <map>
4 #include <set>
5 #define SNI set<nod>::iterator
6 #define SDI set<data>::iterator
7 using namespace std;
8 constexpr int N = 1e5 + 10;
9 int n;
10 int m;
11 int pre[N];
12 int npre[N];
13 int a[N];
14 int tp[N];
15 int lf[N];
16 int rt[N];
17 int co[N];
18
19 struct modi {
20     int t;
21     int pos;
22     int pre;
23     int va;
24
25     friend bool operator<(modi a, modi b) { return a.pre <
26 b.pre; }
27 } md[10 * N];
28
29 int tp1;
30
31 struct qry {
32     int t;
33     int l;
34     int r;
35     int ans;
36
37     friend bool operator<(qry a, qry b) { return a.l < b.l; }
38 } qr[N];
39
40 int tp2;
41 int cnt;
42
43 bool cmp(const qry& a, const qry& b) { return a.t < b.t; }
44
45 void modify(int pos, int co) // 修改函数
46 {
47     if (npre[pos] == co) return;
48     md[++tp1] = modi{++cnt, pos, npre[pos], -1};
49     md[++tp1] = modi{++cnt, pos, npre[pos] = co, 1};
```

```

50 }
51
52 namespace prew {
53 int lst[2 * N];
54 map<int, int> mp; // 提前离散化
55
56 void prew() {
57     cin.tie(nullptr)->sync_with_stdio(false);
58     cin >> n >> m;
59     for (int i = 1; i <= n; i++) cin >> a[i], mp[a[i]] = 1;
60     for (int i = 1; i <= m; i++) {
61         cin >> tp[i] >> lf[i] >> rt[i];
62         if (tp[i] == 1) cin >> co[i], mp[co[i]] = 1;
63     }
64     map<int, int>::iterator it, it1;
65     for (it = mp.begin(), it1 = it, ++it1; it1 != mp.end();
66     ++it, ++it1)
67         it1->second += it->second;
68     for (int i = 1; i <= n; i++) a[i] = mp[a[i]];
69     for (int i = 1; i <= n; i++)
70         if (tp[i] == 1) co[i] = mp[co[i]];
71     for (int i = 1; i <= n; i++) pre[i] = lst[a[i]], lst[a[i]]
72     = i;
73     for (int i = 1; i <= n; i++) npre[i] = pre[i];
74 }
75 } // namespace prew
76
77 namespace colist {
78 struct data {
79     int l;
80     int r;
81     int x;
82
83     friend bool operator<(data a, data b) { return a.r < b.r; }
84 };
85
86 set<data> s;
87
88 struct nod {
89     int l;
90     int r;
91
92     friend bool operator<(nod a, nod b) { return a.r < b.r; }
93 };
94
95 set<nod> c[2 * N];
96 set<int> bd;
97
98 void split(int mid) { // 将一个节点拆成两个节点
99     SDI it = s.lower_bound(data{0, mid, 0});
100    data p = *it;
101    if (mid == p.r) return;

```

```

102     s.erase(p);
103     s.insert(data{p.l, mid, p.x});
104     s.insert(data{mid + 1, p.r, p.x});
105     c[p.x].erase(nod{p.l, p.r});
106     c[p.x].insert(nod{p.l, mid});
107     c[p.x].insert(nod{mid + 1, p.r});
108 }
109
110 void del(set<data>::iterator it) { // 删除一个迭代器
111     bd.insert(it->l);
112     SNI it1, it2;
113     it1 = it2 = c[it->x].find(nod{it->l, it->r});
114     ++it2;
115     if (it2 != c[it->x].end()) bd.insert(it2->l);
116     c[it->x].erase(it1);
117     s.erase(it);
118 }
119
120 void ins(data p) { // 插入一个节点
121     s.insert(p);
122     SNI it = c[p.x].insert(nod{p.l, p.r}).first;
123     ++it;
124     if (it != c[p.x].end()) {
125         bd.insert(it->l);
126     }
127 }
128
129 void stv(int l, int r, int x) { // 区间赋值
130     if (l != 1) split(l - 1);
131     split(r);
132     int p = l; // split两下之后删掉所有区间
133     while (p != r + 1) {
134         SDI it = s.lower_bound(data{0, p, 0});
135         p = it->r + 1;
136         del(it);
137     }
138     ins(data{l, r, x}); // 扫一遍set处理所有变化的pre值
139     for (set<int>::iterator it = bd.begin(); it != bd.end();
140     ++it) {
141         SDI it1 = s.lower_bound(data{0, *it, 0});
142         if (*it != it1->l)
143             modify(*it, *it - 1);
144         else {
145             SNI it2 = c[it1->x].lower_bound(nod{0, *it});
146             if (it2 != c[it1->x].begin())
147                 --it2, modify(*it, it2->r);
148             else
149                 modify(*it, 0);
150         }
151     }
152     bd.clear();
153 }

```

```

154
155     void ih() {
156         int nc = a[1];
157         int ccnt = 1; // 将连续的一段插入到set中
158         for (int i = 2; i <= n; i++)
159             if (nc != a[i]) {
160                 s.insert(data{i - ccnt, i - 1, nc}), c[nc].insert(node{i - ccnt, i - 1});
161                 nc = a[i];
162                 ccnt = 1;
163             } else {
164                 ccnt++;
165             }
166             s.insert(data{n - ccnt + 1, n, a[n]}), c[a[n]].insert(node{n - ccnt + 1, n});
167         }
168     } // namespace colist
169
170 namespace CDQ {
171     struct treearray // 树状数组
172     {
173         int ta[N];
174
175         void c(int x, int t) {
176             for (; x <= n; x += x & (-x)) ta[x] += t;
177         }
178
179         void d(int x) {
180             for (; x <= n; x += x & (-x)) ta[x] = 0;
181         }
182
183         int q(int x) {
184             int r = 0;
185             for (; x; x -= x & (-x)) r += ta[x];
186             return r;
187         }
188
189         void clear() {
190             for (int i = 1; i <= n; i++) ta[i] = 0;
191         }
192     } ta;
193
194     int srt[N];
195
196     bool cmp1(const int& a, const int& b) { return pre[a] < pre[b]; }
197
198     void solve(int l1, int r1, int l2, int r2, int L, int R) {
199         // CDQ
200         if (l1 == r1 || l2 == r2) return;
201         int mid = (L + R) / 2;
202         int mid1 = l1;

```

```

206     while (mid1 != r1 && md[mid1 + 1].t <= mid) mid1++;
207     int mid2 = l2;
208     while (mid2 != r2 && qr[mid2 + 1].t <= mid) mid2++;
209     solve(l1, mid1, l2, mid2, L, mid);
210     solve(mid1, r1, mid2, r2, mid, R);
211     if (l1 != mid1 && mid2 != r2) {
212         sort(md + l1 + 1, md + mid1 + 1);
213         sort(qr + mid2 + 1, qr + r2 + 1);
214         for (int i = mid2 + 1, j = l1 + 1; i <= r2; i++) { // 考
215             虑左侧对右侧贡献
216             while (j <= mid1 && md[j].pre < qr[i].l)
217                 ta.c(md[j].pos, md[j].va), j++;
218                 qr[i].ans += ta.q(qr[i].r) - ta.q(qr[i].l - 1);
219             }
220             for (int i = l1 + 1; i <= mid1; i++) ta.d(md[i].pos);
221         }
222     }
223
224 void mainsolve() {
225     colist::ih();
226     for (int i = 1; i <= m; i++)
227         if (tp[i] == 1)
228             colist::stv(lf[i], rt[i], co[i]);
229         else
230             qr[++tp2] = qry{++cnt, lf[i], rt[i], 0};
231             sort(qr + 1, qr + tp2 + 1);
232             for (int i = 1; i <= n; i++) srt[i] = i;
233             sort(srt + 1, srt + n + 1, cmp1);
234             for (int i = 1, j = 1; i <= tp2; i++) { // 初始化一下每个询
235            问的值
236                 while (j <= n && pre[srt[j]] < qr[i].l) ta.c(srt[j], 1),
237             j++;
238                 qr[i].ans += ta.q(qr[i].r) - ta.q(qr[i].l - 1);
239             }
240             ta.clear();
241             sort(qr + 1, qr + tp2 + 1, cmp);
242             solve(0, tp1, 0, tp2, 0, cnt);
243             sort(qr + 1, qr + tp2 + 1, cmp);
244             for (int i = 1; i <= tp2; i++) cout << qr[i].ans << '\n';
245         }
246     } // namespace CDQ
247
248     int main() {
249         prew::prew();
250         CDQ::mainsolve();
251         return 0;
252     }

```



## [HNOI2010] 城市建设



PS 国是一个拥有诸多城市的大国。国王 Louis 为城市的交通建设可谓绞尽脑汁。Louis 可以在某些城市之间修建道路，在不同的城市之间修建道路需要不同的花费。

Louis 希望建造最少的道路使得国内所有的城市连通。但是由于某些因素，城市之间修建道路需要的花费会随着时间而改变。Louis 会不断得到某道路的修建代价改变的消息。他希望每得到一条消息后能立即知道使城市连通的最小花费总和。Louis 决定求助于你来完成这个任务。

## 解题思路

▼

一句话题意：给定一张图支持动态的修改边权，要求在每次修改边权之后输出这张图的最小生成树的最小代价和。

事实上，有一个线段树分治套 lct 的做法可以解决这个问题，但是这个实现方式的常数过大，可能需要精妙的卡常技巧才可以通过本题，因此不妨考虑 CDQ 分治来解决这个问题。

和一般的 CDQ 分治解决的问题不同，此时使用 CDQ 分治的时候并没有修改和询问的关系来让我们进行分治，因为无法单独考虑「修改一个边对整张图的最小生成树有什么贡献」。传统的 CDQ 分治思路似乎不是很好使。

通过刚才的例题可以发现，一般的 CDQ 分治和线段树有着特殊的联系：我们在 CDQ 分治的过程中其实隐式地建了一棵线段树出来（因为 CDQ 分治的递归树就是一颗线段树）。通常的 CDQ 是考虑线段树左右儿子之间的联系。而对于这道题，我们需要考虑的是父亲和孩子之间的关系；换句话来讲，我们在  $\$solve(l, r)$  这段区间的时候，如果可以想办法使图的规模变成和区间长度相关的一个变量的话，就可以解决这个问题了。

那么具体来讲如何设计算法呢？

假设我们正在构造  $(l, r)$  这段区间的最小生成树边集，并且我们已知它父亲最小生成树的边集。我们将在  $(l, r)$  这段区间中发生变化的边分别赋与  $+\infty$  和  $-\infty$  的边权，并各跑一遍 kruskal，求出在最小生成树里的那些边。

对于一条边来讲：

- 如果最小生成树里所有被修改的边权都被赋成了  $+\infty$ ，而它未出现在树中，则证明它不可能出现在  $(l, r)$  这些询问的最小生成树当中。所以我们仅仅在  $(l, r)$  的边集中加入最小生成树的树边。
- 如果最小生成树里所有被修改的边权都被赋成了  $-\infty$ ，而它未出现在树中，则证明它一定会出现  $(l, r)$  这段的区间的最小生成树当中。这样的话我们就可以使用并查集将这些边对应的点缩起来，并且将答案加上这些边的边权。

这样我们就将  $(l, r)$  这段区间的边集构造出来了。用这些边求出来的最小生成树和直接求原图的最小生成树等价。

那么为什么我们的复杂度是对的呢？

首先，修改过的边一定会加进我们的边集，这些边的数目是  $O(len)$  级别的。

接下来我们需要证明边集当中不会有过多的未被修改的边。我们只会加入所有边权取  $+\infty$  最小生成树的树边，因此我们加入的边数目不会超过当前图的点数。

现在我们只需证明每递归一层图的点数是  $O(len)$  级别的，就可以说明图的边数是  $O(len)$  级别的了。

证明点数是  $O(len)$  几倍就变得十分简单了。我们每次向下递归的时候缩掉的边是在  $-\infty$  生成树中出现的未被修改边，反过来想就是，我们割掉了出现在  $-\infty$  生成树当中的所有的被修改边。显然我们最多割掉  $len$  条边，整张图最多分裂成  $O(len)$  个连通块，这样的话新图点数就

是  $O(len)$  级别的了。所以我们就证明了每次我们用来跑 kruskal 的图都是  $O(len)$  级别的了，从而每一层的时间复杂度都是  $O(n \log n)$  了。

时间复杂度是  $T(n) = T(\lfloor \frac{n}{2} \rfloor) + T(\lceil \frac{n}{2} \rceil) + O(n \log n) = O(n \log^2 n)$ 。

代码实现上可能会有一些难度。需要注意的是并查集不能使用路径压缩，否则就不支持回退操作了。执行缩点操作的时候也没有必要真的执行，而是每一层的 kruskal 都在上一层的并查集中直接做就可以了。



## 示例代码

```
1 #include <algorithm>
2 #include <iostream>
3 #include <stack>
4 #include <vector>
5 using namespace std;
6 using ll = long long;
7 int n;
8 int m;
9 int ask;
10
11 struct bcj {
12     int fa[20010];
13     int size[20010];
14
15     struct opt {
16         int u;
17         int v;
18     };
19
20     stack<opt> st;
21
22     void ih() {
23         for (int i = 1; i <= n; i++) fa[i] = i, size[i] = 1;
24     }
25
26     int f(int x) { return (fa[x] == x) ? x : f(fa[x]); }
27
28     void u(int x, int y) { // 带撤回
29         int u = f(x);
30         int v = f(y);
31         if (u == v) return;
32         if (size[u] < size[v]) swap(u, v);
33         size[u] += size[v];
34         fa[v] = u;
35         opt o;
36         o.u = u;
37         o.v = v;
38         st.push(o);
39     }
40
41     void undo() {
42         opt o = st.top();
43         st.pop();
44         fa[o.v] = o.v;
45         size[o.u] -= size[o.v];
46     }
47
48     void clear(int tim) {
49         while (st.size() > tim) {
```

```

50         undo();
51     }
52 }
53 } s, s1;
54
55 struct edge // 静态边
56 {
57     int u;
58     int v;
59     ll val;
60     int mrk;
61
62     friend bool operator<(edge a, edge b) { return a.val <
63     b.val; }
64 } e[50010];
65
66 struct moved {
67     int u;
68     int v;
69 }; // 动态边
70
71 struct query {
72     int num;
73     ll val;
74     ll ans;
75 } q[50010];
76
77 bool book[50010]; // 询问
78 vector<edge> ve[30];
79 vector<moved> vq;
80 vector<edge> tr;
81 ll res[30];
82 int tim[30];
83
84 void pushdown(int dep) // 缩边
85 {
86     tr.clear(); // 这里要复制一份，以免无法回撤操作
87     for (int i = 0; i < ve[dep].size(); i++) {
88         tr.push_back(ve[dep][i]);
89     }
90     sort(tr.begin(), tr.end());
91     for (int i = 0; i < tr.size(); i++) { // 无用边
92         if (s1.f(tr[i].u) == s1.f(tr[i].v)) {
93             tr[i].mrk = -1;
94             continue;
95         }
96         s1.u(tr[i].u, tr[i].v);
97     }
98     s1.clear(0);
99     res[dep + 1] = res[dep];
100    for (int i = 0; i < vq.size(); i++) {
101        s1.u(vq[i].u, vq[i].v);

```

```

102     }
103     vq.clear();
104     for (int i = 0; i < tr.size(); i++) { // 必须边
105         if (tr[i].mrk == -1 || s1.f(tr[i].u) == s1.f(tr[i].v))
106             continue;
107         tr[i].mrk = 1;
108         s1.u(tr[i].u, tr[i].v);
109         s.u(tr[i].u, tr[i].v);
110         res[dep + 1] += tr[i].val;
111     }
112     s1.clear(0);
113     ve[dep + 1].clear();
114     for (int i = 0; i < tr.size(); i++) { // 缩边
115         if (tr[i].mrk != 0) continue;
116         edge p;
117         p.u = s.f(tr[i].u);
118         p.v = s.f(tr[i].v);
119         if (p.u == p.v) continue;
120         p.val = tr[i].val;
121         p.mrk = 0;
122         ve[dep + 1].push_back(p);
123     }
124     return;
125 }
126
127 void solve(int l, int r, int dep) {
128     tim[dep] = s.st.size();
129     int mid = (l + r) / 2;
130     if (r - l == 1) { // 终止条件
131         edge p;
132         p.u = s.f(e[q[r].num].u);
133         p.v = s.f(e[q[r].num].v);
134         p.val = q[r].val;
135         e[q[r].num].val = q[r].val;
136         p.mrk = 0;
137         ve[dep].push_back(p);
138         pushdown(dep);
139         q[r].ans = res[dep + 1];
140         s.clear(tim[dep - 1]);
141         return;
142     }
143     for (int i = l + 1; i <= mid; i++) {
144         book[q[i].num] = true;
145     }
146     for (int i = mid + 1; i <= r; i++) { // 动转静
147         if (book[q[i].num]) continue;
148         edge p;
149         p.u = s.f(e[q[i].num].u);
150         p.v = s.f(e[q[i].num].v);
151         p.val = e[q[i].num].val;
152         p.mrk = 0;
153         ve[dep].push_back(p);

```

```

154     }
155     for (int i = l + 1; i <= mid; i++) { // 询问转动态
156         moved p;
157         p.u = s.f(e[q[i].num].u);
158         p.v = s.f(e[q[i].num].v);
159         vq.push_back(p);
160     }
161     pushdown(dep); // 下面的是回撤
162     for (int i = mid + 1; i <= r; i++) {
163         if (book[q[i].num]) continue;
164         ve[dep].pop_back();
165     }
166     for (int i = l + 1; i <= mid; i++) {
167         book[q[i].num] = false;
168     }
169     solve(l, mid, dep + 1);
170     for (int i = 0; i < ve[dep].size(); i++) {
171         ve[dep][i].mrk = 0;
172     }
173     for (int i = mid + 1; i <= r; i++) {
174         book[q[i].num] = true;
175     }
176     for (int i = l + 1; i <= mid; i++) { // 动转静
177         if (book[q[i].num]) continue;
178         edge p;
179         p.u = s.f(e[q[i].num].u);
180         p.v = s.f(e[q[i].num].v);
181         p.val = e[q[i].num].val;
182         p.mrk = 0;
183         ve[dep].push_back(p);
184     }
185     for (int i = mid + 1; i <= r; i++) { // 询问转动
186         book[q[i].num] = false;
187         moved p;
188         p.u = s.f(e[q[i].num].u);
189         p.v = s.f(e[q[i].num].v);
190         vq.push_back(p);
191     }
192     pushdown(dep);
193     solve(mid, r, dep + 1);
194     s.clear(tim[dep - 1]);
195     return; // 时间倒流至上一层
196 }
197
198 int main() {
199     cin.tie(nullptr)->sync_with_stdio(false);
200     cin >> n >> m >> ask;
201     s.ih();
202     s1.ih();
203     for (int i = 1; i <= m; i++) {
204         cin >> e[i].u >> e[i].v >> e[i].val;
205     }

```

```
206     for (int i = 1; i <= ask; i++) {
207         cin >> q[i].num >> q[i].val;
208     }
209     for (int i = 1; i <= ask; i++) { // 初始动态边
210         book[q[i].num] = true;
211         moved p;
212         p.u = e[q[i].num].u;
213         p.v = e[q[i].num].v;
214         vq.push_back(p);
215     }
216     for (int i = 1; i <= m; i++) { // 初始静态
217         if (book[i]) continue;
218         ve[1].push_back(e[i]);
219     }
220     for (int i = 1; i <= ask; i++) {
221         book[q[i].num] = false;
222     }
223     solve(0, ask, 1);
224     for (int i = 1; i <= ask; i++) {
225         cout << q[i].ans << '\n';
226     }
227     return 0;
228 }
```

## 参考资料与注释

### 1. 从《Cash》谈一类分治算法的应用 ↶

🔧 本页面最近更新：2025/8/28 16:12:41，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：[Ir1d](#), [H-J-Granger](#), [NachtgeistW](#), [StudyingFather](#), [countercurrent-time](#), [hsfzLZH1](#), [CCXXXI](#), [Early0v0](#), [Enter-tainer](#), [Tiphereth-A](#), [AngelKitty](#), [c-forrest](#), [Chrogeek](#), [cjsoft](#), [diauweb](#), [ezoixx130](#), [GekkaSaori](#), [Konano](#), [LovelyBuggies](#), [lyccrius](#), [Makkiy](#), [mgt](#), [minghu6](#), [P-Y-Y](#), [PotassiumWings](#), [rtxu](#), [SamZhangQingChuan](#), [sshwy](#), [Suyun514](#), [weiyong1024](#), [1804040636](#), [abc1763613206](#), [Backl1ght](#), [flylai](#), [GavinZhengOI](#), [Gesrua](#), [Great-designer](#), [i207M](#), [kenlig](#), [kxccc](#), [Luckyblock233](#), [lychees](#), [ouuan](#), [Peanut-Tang](#), [Planet6174](#), [Revltalize](#), [shadowice1984](#), [Siyuan](#), [SukkaW](#), [Xarfa](#), [Xeonacid](#), [ylxmf2005](#)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用



# 404 Not Found

🔧 本页面最近更新：无更新，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：(自动生成)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用





# 404 Not Found

🔧 本页面最近更新：无更新，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：(自动生成)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用





# 404 Not Found

🔧 本页面最近更新：无更新，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：(自动生成)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用





# 分数规划

分数规划用来求一个分式的极值。

形象一点就是，给出  $a_i$  和  $b_i$ ，求一组  $w_i \in \{0, 1\}$ ，最小化或最大化

$$\frac{\sum_{i=1}^n a_i \times w_i}{\sum_{i=1}^n b_i \times w_i}$$

另外一种描述：每种物品有两个权值  $a$  和  $b$ ，选出若干个物品使得  $\frac{\sum a}{\sum b}$  最小/最大。

一般分数规划问题还会有一些奇怪的限制，比如『分母至少为  $W$ 』。

## 求解

### 二分法

分数规划问题的通用方法是二分。

假设我们要求最大值。二分一个答案  $mid$ ，然后推式子（为了方便少写了上下界）：

$$\begin{aligned} & \frac{\sum a_i \times w_i}{\sum b_i \times w_i} > mid \\ \Rightarrow & \sum a_i \times w_i - mid \times \sum b_i \cdot w_i > 0 \\ \Rightarrow & \sum w_i \times (a_i - mid \times b_i) > 0 \end{aligned}$$

那么只要求出不等号左边的式子的最大值就行了。如果最大值比 0 要大，说明  $mid$  是可行的，否则不可行。

求最小值的方法和求最大值的方法类似，读者不妨尝试着自己推一下。

### Dinkelbach 算法

Dinkelbach 算法的大概思想是每次用上一轮的答案当做新的  $L$  来输入，不断地迭代，直至答案收敛。

---

分数规划的主要难点就在于如何求  $\sum w_i \times (a_i - mid \times b_i)$  的最大值/最小值。下面通过一系列实例来讲解该式子的最大值/最小值的求法。

## 实例

### 模板

有  $n$  个物品，每个物品有两个权值  $a$  和  $b$ 。求一组  $w_i \in \{0, 1\}$ ，最大化  $\frac{\sum a_i \times w_i}{\sum b_i \times w_i}$  的值。

把  $a_i - mid \times b_i$  作为第  $i$  个物品的权值，贪心地选所有权值大于 0 的物品即可得到最大值。

为了方便初学者理解，这里放上完整代码：

## 参考代码

```
1 #include <algorithm>
2 #include <cmath>
3 #include <cstdio>
4 #include <cstdlib>
5 #include <cstring>
6 #include <iostream>
7 using namespace std;
8
9 int read() {
10     int X = 0, w = 1;
11     char c = getchar();
12     while (c < '0' || c > '9') {
13         if (c == '-') w = -1;
14         c = getchar();
15     }
16     while (c >= '0' && c <= '9') X = X * 10 + c - '0', c =
17     getchar();
18     return X * w;
19 }
20
21 constexpr int N = 1000000 + 10;
22 constexpr double eps = 1e-6;
23
24 int n;
25 double a[N], b[N];
26
27 bool check(double mid) {
28     double s = 0;
29     for (int i = 1; i <= n; ++i)
30         if (a[i] - mid * b[i] > 0) // 如果权值大于 0
31             s += a[i] - mid * b[i]; // 选这个物品
32     return s > 0;
33 }
34
35 int main() {
36     // 输入
37     n = read();
38     for (int i = 1; i <= n; ++i) a[i] = read();
39     for (int i = 1; i <= n; ++i) b[i] = read();
40     // 二分
41     double L = 0, R = 1e9;
42     while (R - L > eps) {
43         double mid = (L + R) / 2;
44         if (check(mid)) // mid 可行, 答案比 mid 大
45             L = mid;
46         else // mid 不可行, 答案比 mid 小
47             R = mid;
48     }
49     // 输出
```

```
50     printf("%.6lf\n", L);
51     return 0;
}
```

为了节省篇幅，下面的代码只保留 `check` 部分。主程序和本题是类似的。

## POJ2976 Dropping tests

有  $n$  个物品，每个物品有两个权值  $a$  和  $b$ 。

你可以选  $n - k$  个物品  $p_1, p_2, \dots, p_{n-k}$ ，使得  $\frac{\sum a_{p_i}}{\sum b_{p_i}}$  最大。

输出答案乘 100 后四舍五入到整数的值。

把第  $i$  个物品的权值设为  $a_i - mid \times b_i$ ，然后选最大的  $n - k$  个即可得到最大值。

```
1  bool cmp(double x, double y) { return x > y; }
2
3  bool check(double mid) {
4      int s = 0;
5      for (int i = 1; i <= n; ++i) c[i] = a[i] - mid * b[i];
6      sort(c + 1, c + n + 1, cmp);
7      for (int i = 1; i <= n - k; ++i) s += c[i];
8      return s > 0;
9  }
```

## 洛谷 4377 Talent Show

有  $n$  个物品，每个物品有两个权值  $a$  和  $b$ 。

你需要确定一组  $w_i \in \{0, 1\}$ ，使得  $\frac{\sum w_i \times a_i}{\sum w_i \times b_i}$  最大。

要求  $\sum w_i \times b_i \geq W$ 。

本题多了分母至少为  $W$  的限制，因此无法再使用上一题的贪心算法。

可以考虑 01 背包。把  $b_i$  作为第  $i$  个物品的重量， $a_i - mid \times b_i$  作为第  $i$  个物品的价值，然后问题就转化为背包了。

那么  $dp[n][W]$  就是最大值。

一个要注意的地方： $\sum w_i \times b_i$  可能超过  $W$ ，此时直接视为  $W$  即可。（想一想，为什么？）

```
1  double f[1010];
2
3  bool check(double mid) {
```

```

4     for (int i = 1; i <= W; i++) f[i] = -1e9;
5     for (int i = 1; i <= n; i++)
6         for (int j = W; j >= 0; j--) {
7             int k = min(W, j + b[i]);
8             f[k] = max(f[k], f[j] + a[i] - mid * b[i]);
9         }
10    return f[W] > 0;
11 }
```

## POJ2728 Desert King

每条边有两个权值  $a_i$  和  $b_i$ , 求一棵生成树  $T$  使得  $\frac{\sum_{e \in T} a_e}{\sum_{e \in T} b_e}$  最小。

把  $a_i - mid \times b_i$  作为每条边的权值, 那么最小生成树就是最小值,

代码就是求最小生成树, 故省略。

## [HNOI2009] 最小圈

每条边的边权为  $w$ , 求一个环  $C$  使得  $\frac{\sum_{e \in C} w}{|C|}$  最小。

把  $a_i - mid$  作为边权, 那么权值最小的环就是最小值。

因为我们只需要判最小值是否小于 0, 所以只需要判断图中是否存在负环即可。

另外本题存在一种复杂度  $O(nm)$  的算法, 如果有兴趣可以阅读 [这篇文章](#)。

```

1 int SPFA(int u, double mid) { // 判负环
2     vis[u] = 1;
3     for (int i = head[u]; i; i = e[i].nxt) {
4         int v = e[i].v;
5         double w = e[i].w - mid;
6         if (dis[u] + w < dis[v]) {
7             dis[v] = dis[u] + w;
8             if (vis[v] || SPFA(v, mid)) return 1;
9         }
10    }
11    vis[u] = 0;
12    return 0;
13 }
14
15 bool check(double mid) { // 如果有负环返回 true
16     for (int i = 1; i <= n; ++i) dis[i] = 0, vis[i] = 0;
17     for (int i = 1; i <= n; ++i)
18         if (SPFA(i, mid)) return true;
19     return false;
20 }
```

## 总结

分数规划问题是一类既套路又灵活的题目，一般使用二分解决。

分数规划问题的主要难点在于推出式子后想办法求出  $\sum w_i \times (a_i - mid \times b_i)$  的最大值/最小值，而这个需要具体情况具体分析。

## 习题

- [JSOI2016 最佳团体](#)
- [SDOI2017 新生舞会](#)
- [UVa1389 Hard Life](#)
- [洛谷 P2868 \[USACO07DEC\] Sightseeing Cows G](#)

🔧 本页面最近更新：2025/7/24 11:14:37，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：[StudyingFather](#), [Ir1d](#), [H-J-Granger](#), [countercurrent-time](#), [greyqz](#), [NachtgeistW](#), [Tiphereth-A](#), [Early0v0](#), [Enter-tainer](#), [Mout-sea](#), [AngelKitty](#), [banglee13](#), [CCXXXI](#), [cjsoft](#), [diauweb](#), [ezoixx130](#), [GekkaSaori](#), [hsfzLZH1](#), [huaruoji](#), [Konano](#), [LovelyBuggies](#), [Makkiiy](#), [mgt](#), [minghu6](#), [P-Y-Y](#), [PotassiumWings](#), [SamZhangQingChuan](#), [sshwy](#), [Suyun514](#), [weiyong1024](#), [alphagocc](#), [ChungZH](#), [GavinZhengOl](#), [Gesrua](#), [Henry-ZHR](#), [ksyx](#), [kxcc](#), [lyccrius](#), [lychees](#), [MicDZ](#), [ouuan](#), [Peanut-Tang](#), [r-value](#), [SukkaW](#), [Xeonacid](#)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用



# 404 Not Found

🔧 本页面最近更新：无更新，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：(自动生成)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用





# 整体二分

## 引入

在信息学竞赛中，有一部分题目可以使用二分的办法来解决。但是当这种题目有~~多次~~询问且我们每次查询都直接二分可能导致 TLE 时，就会用到整体二分。整体二分的主体思路就是把多个查询一起解决。（所以这是一个离线算法）

可以使用整体二分解决的题目需要满足以下性质：

1. 询问的答案具有可二分性
2. **修改对判定答案的贡献互相独立**，修改之间互不影响效果
3. 修改如果对判定答案有贡献，则贡献为一确定的与判定标准无关的值
4. 贡献满足交换律，结合律，具有可加性
5. 题目允许使用离线算法

——许昊然《浅谈数据结构题几个非经典解法》

## 解释

记  $[l, r]$  为答案的值域， $[L, R]$  为答案的定义域。（也就是说求答案时仅考虑下标在区间  $[L, R]$  内的操作和询问，这其中询问的答案在  $[l, r]$  内）

- 我们首先把所有操作 **按时间顺序** 存入数组中，然后开始分治。
- 在每一层分治中，利用数据结构（常见的是树状数组）统计当前查询的答案和  $mid$  之间的关系。
- 根据查询出来的答案和  $mid$  间的关系（小于等于  $mid$  和大于  $mid$ ）将当前处理的操作序列分为  $q_1$  和  $q_2$  两份，并分别递归处理。
- 当  $l = r$  时，找到答案，记录答案并返回即可。

需要注意的是，在整体二分过程中，若当前处理的值域为  $[l, r]$ ，则此时最终答案范围不在  $[l, r]$  的询问会在其他时候处理。

## 过程

注：

1. 为可读性，文中代码或未采用实际竞赛中的常见写法。

2. 若觉得某段代码有难以理解之处，请先参考之前题目的解释，因为节省篇幅解释过的内容不再赘述。

从普通二分说起：

## 查询全局第 $k$ 小

题 1 在一个数列中查询第  $k$  小的数。

当然可以直接排序。如果用二分法呢？可以用数据结构记录每个大小范围内有多少个数，然后用二分法猜测，利用数据结构检验。

题 2 在一个数列中多次查询第  $k$  小的数。

可以对于每个询问进行一次二分；但是，也可以把所有的询问放在一起二分。

先考虑二分的本质：假设要猜一个  $[l, r]$  之间的数，猜测之后会知道是猜大了，猜小了还是刚好。当然可以从  $l$  枚举到  $r$ ，但更优秀的方法是二分：猜测答案是  $m = \lfloor \frac{l+r}{2} \rfloor$ ，然后去验证  $m$  的正确性，再调整边界。这样做每次询问的复杂度为  $O(\log n)$ ，若询问次数为  $q$ ，则时间复杂度为  $O(q \log n)$ 。

回过头来，对于当前的所有询问，可以去猜测所有询问的答案都是  $mid$ ，然后去依次验证每个询问的答案应该是小于等于  $mid$  的还是大于  $mid$  的，并将询问分为两个部分（不大于/大于），对于每个部分继续二分。注意：如果一个询问的答案是大于  $mid$  的，则在将其划至右侧前需更新它的  $k$ ，即，如果当前数列中小于等于  $mid$  的数有  $t$  个，则将询问划分后实际是在右区间询问第  $k - t$  小数。如果一个部分的  $l = r$  了，则结束这个部分的二分。利用线段树的相关知识，我们每次将整个答案可能在的区间  $[1, n]$ （假设已经离散化）划分成了若干个部分，这样的划分共进行了  $O(\log n)$  次，一次划分会将整个操作序列操作一次。若对整个序列进行操作，并支持对应的查询的时间复杂度为  $O(T)$ ，则整体二分的时间复杂度为  $O(T \log n)$ 。

参考代码如下：

## 实现

```
1 struct Query {
2     int id, k; // 这个询问的编号，这个询问的 k
3 };
4
5 int ans[N], a[N]; // ans[i] 表示编号为 i 的询问的答案，a 为原数列
6 int val[N], cnt[N]; // 离散化后，记录对应的值及其计数（假设已经处理
7 好）
8
9 // 返回原数列中值域在 [l, r] 中的数的个数
10 int check(int l, int r) {
11     int res = 0;
12     for (int i = l; i <= r; i++) {
13         res += cnt[i];
14     }
15     return res;
16 }
17
18 // 整体二分
19 void solve(int l, int r, vector<Query> q) {
20     int m = (l + r) / 2;
21     if (l == r) {
22         for (unsigned i = 0; i < q.size(); i++) ans[q[i].id] =
23         val[l];
24         return;
25     }
26     vector<Query> q1, q2;
27     int t = check(l, m);
28     for (unsigned i = 0; i < q.size(); i++) {
29         if (q[i].k <= t)
30             q1.push_back(q[i]);
31         else
32             q[i].k -= t, q2.push_back(q[i]);
33     }
34     solve(l, m, q1), solve(m + 1, r, q2);
35     return;
36 }
```

## 查询区间第 $k$ 小

题 3 在一个数列中多次查询区间第  $k$  小的数。

涉及到给定区间的查询，再按之前的方法进行二分就会导致 `check` 函数的时间复杂度爆炸。仍然考虑询问与值域中点  $m$  的关系：若询问区间内小于等于  $m$  的数有  $t$  个，询问的是区间内的  $k$  小数，则当  $k \leq t$  时，答案应小于等于  $m$ ；否则，答案应大于  $m$ 。（注意边界问题）此处需记录一个区间小于等于指定数的数的数量，即单点加，求区间和，可用树状数组快速处理。为提高效率，只对数列中值在值域区间  $[l, r]$  的数进行统计，即，在进一步递归之前，不仅将询问划分，将当前处理的数按值域范围划为两半。

## 参考代码（关键部分）



### 实现

```
1 struct Num {
2     int p, x;
3 }; // 位于数列中第 p 项的数的值为 x
4
5 struct Query {
6     int l, r, k, id;
7 }; // 一个编号为 id, 询问 [l,r] 中第 k 小数的询问
8
9 int ans[N];
10 void add(int p, int x); // 树状数组, 在 p 位置加上 x
11 int query(int p); // 树状数组, 求 [1,p] 的和
12 void clear(); // 树状数组, 清空
13
14 void solve(int l, int r, vector<Num> a, vector<Query> q)
15 // a中为给定数列中值在值域区间 [l,r] 中的数
16 {
17     int m = (l + r) / 2;
18     if (l == r) {
19         for (unsigned i = 0; i < q.size(); i++) ans[q[i].id] = l;
20         return;
21     }
22     vector<Num> a1, a2;
23     vector<Query> q1, q2;
24     for (unsigned i = 0; i < a.size(); i++)
25         if (a[i].x <= m)
26             a1.push_back(a[i]), add(a[i].p, 1);
27         else
28             a2.push_back(a[i]);
29     for (unsigned i = 0; i < q.size(); i++) {
30         int t = query(q[i].r) - query(q[i].l - 1);
31         if (q[i].k <= t)
32             q1.push_back(q[i]);
33         else
34             q[i].k -= t, q2.push_back(q[i]);
35     }
36     clear();
37     solve(l, m, a1, q1), solve(m + 1, r, a2, q2);
38     return;
39 }
```

下面提供 【模板】 可持久化线段树 2 一题使用整体二分的，偏向竞赛风格的写法。



## 参考代码

```
1 #include <algorithm>
2 #include <iostream>
3 using namespace std;
4 constexpr int N = 200020;
5 int n, m;
6 int ans[N];
7 // BIT begin
8 int t[N];
9 int a[N];
10
11 int sum(int p) {
12     int ans = 0;
13     while (p) {
14         ans += t[p];
15         p -= p & (-p);
16     }
17     return ans;
18 }
19
20 void add(int p, int x) {
21     while (p <= n) {
22         t[p] += x;
23         p += p & (-p);
24     }
25 }
26
27 // BIT end
28 int tot = 0;
29
30 struct Query {
31     int l, r, k, id, type; // set values to -1 when they are not
32 used!
33 } q[N * 2], q1[N * 2], q2[N * 2];
34
35 void solve(int l, int r, int ql, int qr) {
36     if (ql > qr) return;
37     if (l == r) {
38         for (int i = ql; i <= qr; i++)
39             if (q[i].type == 2) ans[q[i].id] = l;
40         return;
41     }
42     int mid = (l + r) / 2, cnt1 = 0, cnt2 = 0;
43     for (int i = ql; i <= qr; i++) {
44         if (q[i].type == 1) {
45             if (q[i].l <= mid) {
46                 add(q[i].id, 1);
47                 q1[++cnt1] = q[i];
48             } else
49                 q2[++cnt2] = q[i];
50         }
51     }
52 }
```

```

50     } else {
51         int x = sum(q[i].r) - sum(q[i].l - 1);
52         if (q[i].k <= x)
53             q1[++cnt1] = q[i];
54         else {
55             q[i].k -= x;
56             q2[++cnt2] = q[i];
57         }
58     }
59 }
60 // rollback changes
61 for (int i = 1; i <= cnt1; i++)
62     if (q1[i].type == 1) add(q1[i].id, -1);
63 // move them to the main array
64 for (int i = 1; i <= cnt1; i++) q[i + ql - 1] = q1[i];
65 for (int i = 1; i <= cnt2; i++) q[i + cnt1 + ql - 1] = q2[i];
66 solve(l, mid, ql, cnt1 + ql - 1);
67 solve(mid + 1, r, cnt1 + ql, qr);
68 }
69
70 pair<int, int> b[N];
71 int toRaw[N];
72
73 int main() {
74     cin.tie(nullptr)->sync_with_stdio(false);
75     cin >> n >> m;
76     // read and discrete input data
77     for (int i = 1; i <= n; i++) {
78         int x;
79         cin >> x;
80         b[i].first = x;
81         b[i].second = i;
82     }
83     sort(b + 1, b + n + 1);
84     int cnt = 0;
85     for (int i = 1; i <= n; i++) {
86         if (b[i].first != b[i - 1].first) cnt++;
87         a[b[i].second] = cnt;
88         toRaw[cnt] = b[i].first;
89     }
90     for (int i = 1; i <= n; i++) {
91         q[++tot] = {a[i], -1, -1, i, 1};
92     }
93     for (int i = 1; i <= m; i++) {
94         int l, r, k;
95         cin >> l >> r >> k;
96         q[++tot] = {l, r, k, i, 2};
97     }
98     solve(0, cnt + 1, 1, tot);
99     for (int i = 1; i <= m; i++) cout << toRaw[ans[i]] << '\n';
}

```

## 带修区间第 $k$ 小

题 4 Dynamic Rankings 给定一个数列，要支持单点修改，区间查第  $k$  小。

修改操作可以直接理解为从原数列中删去一个数再添加一个数，为方便起见，将询问和修改统称为「操作」。因后面的操作会依附于之前的操作，不能如题 3 一样将统计和处理询问分开，故将所有操作存于一个数组，用标识区分类型，依次处理每个操作。为便于处理树状数组，修改操作可分拆为擦除操作和插入操作。

### 优化

1. 注意到每次对于操作进行分类时，只会更改操作顺序，故可直接在原数组上操作。具体实现，在二分时将记录操作的  $q, a$  数组换为一个大的全局数组，二分时记录信息变为  $L, R$ ，即当前处理的操作是全局数组上的哪个区间。利用临时数组记录当前的分类情况，进一步递归前将临时数组信息写回原数组。
2. 树状数组每次清空会导致时间复杂度爆炸，可采用每次使用树状数组时记录当前修改位置（这已由 1 中提到的临时数组实现），本次操作结束后在原位置加  $-1$  的方法快速清零。
3. 一开始对于数列的初始化操作可简化为插入操作。

参考代码（关键部分）

## 实现

```
1 struct Opt {
2     int x, y, k, type, id;
3     // 对于询问, type = 1, x, y 表示区间左右边界, k 表示询问第 k 小
4     // 对于修改, type = 0, x 表示修改位置, y 表示修改后的值,
5     // k 表示当前操作是插入(1)还是擦除(-1), 更新树状数组时使用.
6     // id 记录每个操作原先的编号, 因二分过程中操作顺序会被打散
7 };
8
9 Opt q[N], q1[N], q2[N];
10 // q 为所有操作,
11 // 二分过程中, 分到左边的操作存到 q1 中, 分到右边的操作存到 q2 中.
12 int ans[N];
13 void add(int p, int x);
14 int query(int p); // 树状数组函数, 含义见题3
15
16 void solve(int l, int r, int L, int R)
17 // 当前的值域范围为 [l,r], 处理的操作的区间为 [L,R]
18 {
19     if (l > r || L > R) return;
20     int cnt1 = 0, cnt2 = 0, m = (l + r) / 2;
21     // cnt1, cnt2 分别为分到左边, 分到右边的操作数
22     if (l == r) {
23         for (int i = L; i <= R; i++)
24             if (q[i].type == 1) ans[q[i].id] = l;
25         return;
26     }
27     for (int i = L; i <= R; i++)
28         if (q[i].type == 1) { // 是询问: 进行分类
29             int t = query(q[i].y) - query(q[i].x - 1);
30             if (q[i].k <= t)
31                 q1[++cnt1] = q[i];
32             else
33                 q[i].k -= t, q2[++cnt2] = q[i];
34         } else
35             // 是修改: 更新树状数组 & 分类
36             if (q[i].y <= m)
37                 add(q[i].x, q[i].k), q1[++cnt1] = q[i];
38             else
39                 q2[++cnt2] = q[i];
40     for (int i = 1; i <= cnt1; i++)
41         if (q1[i].type == 0) add(q1[i].x, -q1[i].k); // 清空树状数组
42     for (int i = 1; i <= cnt1; i++) q[L + i - 1] = q1[i];
43     for (int i = 1; i <= cnt2; i++)
44         q[L + cnt1 + i - 1] = q2[i]; // 将临时数组中的元素合并回原数组
45     solve(l, m, L, L + cnt1 - 1), solve(m + 1, r, L + cnt1, R);
46     return;
47 }
```

## 针对静态序列的优化

题 5 【模板】可持久化线段树 2 给定一个序列，区间查询第  $k$  小。

树套树和整体二分实现带修区间第  $k$  小问题的复杂度都为  $O(n \log^2 n)$ ，但静态区间第  $k$  小问题可以使用可持久化线段树在  $O(n \log n)$  时间复杂度内解决，而几乎所有整体二分实现的静态区间第  $k$  小问题代码时间复杂度都是  $O(n \log^2 n)$ ，面对大数据范围时存在 TLE 的风险。（这里默认值域与序列长度同阶，值域与序列长不同阶的情况可以通过离散化转化为同阶情况）

### 优化

1. 对于每一轮划分，如果当前数列中小于等于  $mid$  的数有  $t$  个，则将询问划分后实际是在右区间询问第  $k - t$  小数，因此对划分到右区间的询问做出了修改。如果答案的原始值域为  $[L, R]$ ，某次划分的答案值域为  $[l, r]$ ，那么对于参与此次划分的询问， $[L, l)$  中所有数值对它们的影响已经在之前被消除了。
2. 由于需要使每轮划分都仅和当前答案值域  $[l, r]$  有关，树状数组需要多次载入和清空。

如果划分不仅仅和当前答案值域有关呢？

由此可以得到一个与全局序列有关的优化方法：维护一个指针  $pos$  追踪每轮划分的  $mid$ （分治中心），将所有  $\leq pos$  的元素对应的下标在树状数组中置为 1，树状数组的其余位置置为 0。每次划分之前移动  $pos$  并更新树状数组。指针  $pos$  移动的次数与  $n \log n$  同阶。划分时对每一个询问查询树状数组中对应区间的值，满足则划分至左区间，否则划分至右区间，**不需要对询问做出修改**。

由于要追踪分治中心，需要让  $pos$  准确地更新树状数组。在整体二分之前将序列按元素大小排序并记录元素对应下标，指针移动时在树状数组中对下标进行相应修改。对于绝大多数 **可以用整体二分解决并且不带修改的问题**，都可以应用此种优化以大幅降低数据结构的使用次数。

由于减少了很多树状数组的载入和清空操作，应用这种优化通常情况下会明显提升整体二分的效率（即使只是常数优化），对于静态区间第  $k$  小值问题而言效率完全不差于时间复杂度更优的可持久化线段树。值得注意的是，对于静态区间第  $k$  小值问题也存在时间复杂度  $O(n \log n)$  的整体二分实现。

参考代码（关键部分）

## 实现

```
1 struct Query {
2     int i, l, r, k;
3 }; // 第 i 次询问查询区间 [l,r] 的第 k 小值
4
5 Query s[200005], t1[200005], t2[200005];
6 int n, m, cnt, pos, p[200005], ans[200005];
7 pair<int, int> a[200005];
8
9 void add(int x, int y); // 树状数组 位置 x 加 y
10 int sum(int x); // 树状数组 [1,x] 前缀和
11
12 // 当前处理的询问为 [l,r], 答案值域为 [ql,qr]
13 void overall_binary(int l, int r, int ql, int qr) {
14     if (l > r) return;
15     if (ql == qr) {
16         for (int i = l; i <= r; i++) ans[s[i].i] = ql;
17         return;
18     }
19     int cnt1 = 0, cnt2 = 0, mid = (ql + qr) >> 1;
20     // 追踪分治中心,认为 [1,pos] 的值已经载入树状数组
21     while (pos <= n - 1 && a[pos + 1].first <= mid)
22         add(a[pos + 1].second, 1), ++pos;
23     while (pos >= 1 && a[pos].first > mid) add(a[pos].second, -1),
24     --pos;
25
26     for (int i = l; i <= r; i++) {
27         int now = sum(s[i].r) - sum(s[i].l - 1);
28         if (s[i].k <= now)
29             t1[++cnt1] = s[i];
30         else
31             t2[++cnt2] = s[i]; // 注意 不应修改询问信息
32     }
33     for (int i = 1; i <= cnt1; i++) s[l + i - 1] = t1[i];
34     for (int i = 1; i <= cnt2; i++) s[l + cnt1 + i - 1] = t2[i];
35
36     overall_binary(l, l + cnt1 - 1, ql, mid);
37     overall_binary(l + cnt1, r, mid + 1, qr);
38 }
39
40 int main() {
41     scanf("%d%d", &n, &m);
42     for (int i = 1; i <= n; i++) {
43         scanf("%d", &a[i].first);
44         a[i].second = i;
45         p[++cnt] = a[i].first;
46     }
47     sort(a + 1, a + n + 1); // 对序列排序 离散化
48     sort(p + 1, p + n + 1);
49     cnt = unique(p + 1, p + n + 1) - p - 1;
```

```

50     for (int i = 1; i <= n; i++)
51         a[i].first = lower_bound(p + 1, p + cnt + 1, a[i].first) - p;
52     // 省略读入询问
53     overall_binary(1, m, 1, cnt);
54     for (int i = 1; i <= n; i++) printf("%d\n", p[ans[i]]);
55     return 0;
}

```

## 区间前驱后继

**题 6** 在一个数列中多次查询  $k$  在区间中的前驱（严格小于  $k$ , 且最大的数）或后继（严格大于  $k$ , 且最小的数），保证存在这样的数。

以前驱为例，使用数据结构解决此种问题的方法一般是先查询区间内有多少严格小于  $k$  的数（设它们的数量为  $x$ ），再查询区间第  $x$  小的数。后继则是查询区间内有多少不大于  $k$  的数（数量为  $x$ ），然后查询区间第  $x + 1$  小的数。

考虑使用整体二分解决这个问题：整体二分是一种高效求解区间第  $k$  小的离线算法，而 [CDQ 分治](#) 可以离线高效求解区间内的排名。先跑一遍 CDQ 分治求出排名就可以使用整体二分得到区间内部的前驱和后继了。

此问题还可以用 CDQ 分治套线段树离线一遍解决，但效率远低于跑两遍的 CDQ 分治 + 整体二分。

## 构造单调性序列

**题 7 Sequence** 给定一个序列，每次操作可以把某个数  $+1$  或  $-1$ 。要求把序列变成单调不降的，并且修改后的数列只能出现修改前的数，输出最小操作次数。

此类题目也可以使用动态规划或反悔贪心解决。

在满足操作次数最小化的前提下，一定存在一种方案使得最后序列中的每个数都是序列修改前存在的，这个结论可以使用数学归纳法证明。由于题目并不需要最终序列的信息，问题转化为求出最小操作次数。

由于要求最终的序列单调不降，可以使用整体二分。每轮整体二分判定最终序列区间  $[l, r]$  的值域，此时答案的值域为  $[ql, qr]$ 。令  $mid = \lfloor \frac{ql+qr}{2} \rfloor$ ，每轮二分开始时默认将所有数划分至  $[mid + 1, qr]$ （要划分到  $[ql, mid]$  的数设为 0 个），初始代价设为将序列区间  $[l, r]$  全部置为  $mid + 1$  的操作次数。依次枚举区间  $[l, r]$  中的数  $i$  并且计算将  $[l, i]$  置为  $mid$ 、将  $[i + 1, r]$  置为  $mid + 1$  的操作次数之和，如果优于之前的操作次数则更新最少操作次数和要划分到  $[ql, mid]$  的数的个数。

划分时已经保证了最终序列的单调性不被破坏，同时因为每次都取最小操作次数，最终被划分至左区间的数取  $mid$  一定比取  $mid + 1$  更优，故整体二分得到的序列一定是单调不降且操作次数最小的。计算操作次数输出即可。

## 参考代码（关键部分）

实现

```
1 int a[500005], ans[500005]; // a:原序列 ans:构造的序列
2
3 void overall_binary(int l, int r, int ql, int qr) {
4     if (l > r) return;
5     if (ql == qr) {
6         for (int i = l; i <= r; i++) ans[i] = ql;
7         return;
8     }
9     int cnt = 0,
10        mid = ql + ((qr - ql) >> 1); // 默认开始都填 mid+1 全部划分
11    到右区间
12    long long res = 0ll, sum = 0ll;
13    for (int i = l; i <= r; i++) sum += abs(a[i] - (mid + 1));
14    res = sum;
15    for (int i = l; i <= r;
16          i++) { // 尝试把 [l,i] 从 mid+1 换成 mid 并且划分到左区间
17        sum -= abs(a[i] - (mid + 1));
18        sum += abs(a[i] - mid);
19        if (sum < res) cnt = i - l + 1, res = sum; // 发现 [l,i] 取
20        mid 更优，更新
21    }
22    overall_binary(l, l + cnt - 1, ql, mid);
23    overall_binary(l + cnt, r, mid + 1, qr);
}
```

## 参考习题

[「国家集训队」矩阵乘法](#)

[「POI2011 R3 Day2」流星 Meteors](#)

[二逼平衡树](#)

[\[BalticOI 2004\] Sequence 数字序列](#)

## 参考资料

- 许昊然《浅谈数据结构题几个非经典解法》

🔧 本页面最近更新：2025/8/3 04:51:50，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

本页面贡献者：[Ir1d](#), [Henry-ZHR](#), [partychicken](#), [Prurite](#), [ScaredQiu](#), [Tiphereth-A](#), [2018-Danny](#), [abc1763613206](#), [c-forrest](#), [CCXXXI](#), [dengxijian](#), [Enter-tainer](#), [GavinZhengOI](#), [HeRaNO](#), [hsfzLZH1](#), [iamtwz](#), [kenlig](#), [ksyx](#), [LingeZ3z](#), [Lynricsy](#), [Marcythm](#), [ouuan](#), [ranwen](#), [Sheng-Horizon](#), [ShizuhaAki](#), [Shyanko](#)

本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用



# 模拟退火

## 引入

模拟退火是一种随机化算法。当一个问题的方案数量极大（甚至是无穷的）而且不是一个单峰函数时，我们常使用模拟退火求解。

## 解释

根据 [爬山算法](#) 的过程，我们发现：对于一个当前最优解附近的非最优解，爬山算法直接舍去了这个解。而很多情况下，我们需要去接受这个非最优解从而跳出这个局部最优解，即为模拟退火算法。



### 什么是退火？（选自 [百度百科](#)）

退火是一种金属热处理工艺，指的是将金属缓慢加热到一定温度，保持足够时间，然后以适宜速度冷却。目的是降低硬度，改善切削加工性；消除残余应力，稳定尺寸，减少变形与裂纹倾向；细化晶粒，调整组织，消除组织缺陷。准确的说，退火是一种对材料的热处理工艺，包括金属材料、非金属材料。而且新材料的退火目的也与传统金属退火存在异同。

由于退火的规律引入了更多随机因素，那么我们得到最优解的概率会大大增加。于是我们可以去模拟这个过程，将目标函数作为能量函数。

## 过程

先用一句话概括：如果新状态的解更优则修改答案，否则以一定概率接受新状态。

我们定义当前温度为  $T$ ，新状态  $S'$  与已知状态  $S$ （新状态由已知状态通过随机的方式得到）之间的能量（值）差为  $\Delta E$  ( $\Delta E \geq 0$ )，则发生状态转移（修改最优解）的概率为

$$P(\Delta E) = \begin{cases} 1, & S' \text{ is better than } S, \\ e^{-\frac{\Delta E}{T}}, & \text{otherwise.} \end{cases}$$

**注意：**我们有时为了使得到的解更有质量，会在模拟退火结束后，以当前温度在得到的解附近多次随机状态，尝试得到更优的解（其过程与模拟退火相似）。

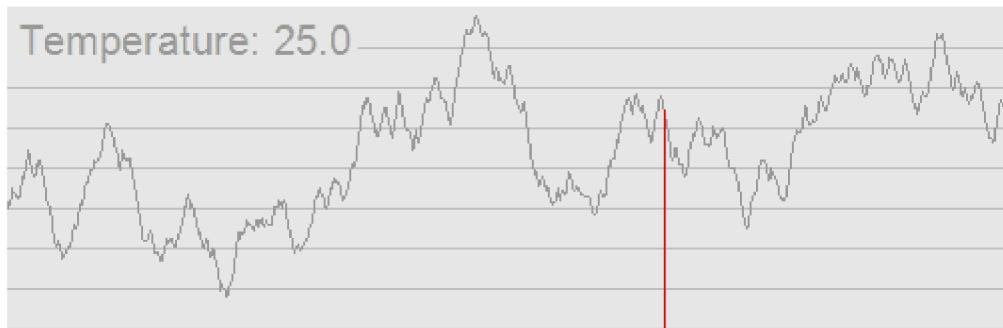
## 如何退火（降温）？

模拟退火时我们有三个参数：初始温度  $T_0$ ，降温系数  $d$ ，终止温度  $T_k$ 。其中  $T_0$  是一个比较大的数， $d$  是一个非常接近 1 但是小于 1 的数， $T_k$  是一个接近 0 的正数。

首先让温度  $T = T_0$ , 然后按照上述步骤进行一次转移尝试, 再让  $T = d \cdot T$ 。当  $T < T_k$  时模拟退火过程结束, 当前最优解即为最终的最优解。

注意为了使得解更为精确, 我们通常不直接取当前解作为答案, 而是在退火过程中维护遇到的所有解的最优值。

引用一张 [Simulated annealing - Wikipedia](#) 的图片 (随着温度的降低, 跳跃越来越不随机, 最优解也越来越稳定)。



## 实现

此处代码以 [「BZOJ 3680」吊打 XXX](#) (求  $n$  个点的带权类费马点) 为例。

```
1 #include <cmath>
2 #include <cstdlib>
3 #include <ctime>
4 #include <iomanip>
5 #include <iostream>
6
7 constexpr int N = 10005;
8 int n, x[N], y[N], w[N];
9 double ansx, ansy, dis;
10
11 double Rand() { return (double)rand() / RAND_MAX; }
12
13 double calc(double xx, double yy) {
14     double res = 0;
15     for (int i = 1; i <= n; ++i) {
16         double dx = x[i] - xx, dy = y[i] - yy;
17         res += sqrt(dx * dx + dy * dy) * w[i];
18     }
19     if (res < dis) dis = res, ansx = xx, ansy = yy;
20     return res;
21 }
22
23 void simulateAnneal() {
24     double t = 100000;
25     double nowx = ansx, nowy = ansy;
26     while (t > 0.001) {
27         double nxty = nowx + t * (Rand() * 2 - 1);
```

```

28     double nxy = nowy + t * (Rand() * 2 - 1);
29     double delta = calc(nxtx, nxy) - calc(nowx, nowy);
30     if (exp(-delta / t) > Rand()) nowx = nxtx, nowy = nxy;
31     t *= 0.97;
32 }
33 for (int i = 1; i <= 1000; ++i) {
34     double nxtx = ansx + t * (Rand() * 2 - 1);
35     double nxy = ansy + t * (Rand() * 2 - 1);
36     calc(nxtx, nxy);
37 }
38 }
39
40 int main() {
41     std::cin.tie(nullptr)->sync_with_stdio(false);
42     srand(0); // 注意，在实际使用中，不应使用固定的随机种子。
43     std::cin >> n;
44     for (int i = 1; i <= n; ++i) {
45         std::cin >> x[i] >> y[i] >> w[i];
46         ansx += x[i], ansy += y[i];
47     }
48     ansx /= n, ansy /= n, dis = calc(ansx, ansy);
49     simulateAnneal();
50     std::cout << std::fixed << std::setprecision(3) << ansx << ' ' << ansy
51             << '\n';
52     return 0;
53 }
```

## 一些技巧

### 分块模拟退火

有时函数的峰很多，模拟退火难以跑出最优解。

此时可以把整个值域分成几段，每段跑一遍模拟退火，然后再取最优解。

### 卡时

有一个 `clock()` 函数，返回程序运行时间。

可以把主程序中的 `simulateAnneal();` 换成 `while ((double)clock() /CLOCKS_PER_SEC < MAX_TIME) simulateAnneal();`。这样子就会一直跑模拟退火，直到用时即将超过时间限制。

这里的 `MAX_TIME` 是一个自定义的略小于时限的数（单位：秒）。

## 习题

- 「BZOJ 3680」吊打 XXX

- 「JSOI 2016」炸弹攻击
  - 「HAOI 2006」均分数据
- 

🔧 本页面最近更新：2025/8/29 18:05:34，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：[lr1d](#), [abc1763613206](#), [Mout-sea](#), [Siyuan](#), [sshyw](#), [Tiphereth-A](#), [7F88FF](#), [ChungZH](#), [Enter-tainer](#), [Ghastlcon](#), [Henry-ZHR](#), [HeRaNO](#), [hsfzLZH1](#), [iamtwz](#), [kenlig](#), [ouuan](#)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用



# 404 Not Found

🔧 本页面最近更新：无更新，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：(自动生成)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用





# 离线算法简介

本章将介绍介绍离线算法（Offline Algorithm）的思想、常见算法及优化。

离线算法是基于「**求解前已知所有数据**」这一假设来设计的，适用于有多组询问的题目。相对的还有 [在线算法](#)（Online Algorithm）。

例如 [选择排序](#) 必须知道数组的全局最小元素才能执行，所以是离线算法，而 [插入排序](#) 可以动态接收数据进行排序，不强制要求执行前已知全部数据，所以是在线算法。

对于相同的问题，在设计难度等方面，离线算法往往优于在线算法。为了阻止选手使用离线算法，有时题目会使用「强制在线」的方式，常见的有需要前一个询问的答案才能得到下一个询问的参数（[交互题](#) 与 [通信题](#) 也属于此类）。

离线算法的常见思路包括将询问统一求解（如 [CDQ 分治](#)）、通过一个询问的答案求出另外相似询问的答案（如 [整体二分](#) 和 [莫队算法](#)）等。

由于离线算法是一种思想而并不是某种具体的算法，因此它会搭配各种各样的数据结构或算法一起使用，与之相关的题目种类也更为繁杂。



本页面最近更新：2025/2/8 15:15:39，[更新历史](#)



发现错误？想一起完善？[在 GitHub 上编辑此页！](#)



本页面贡献者：[Ir1d](#), [Oracynx](#), [Tiphereth-A](#)



© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用



# 表达式求值

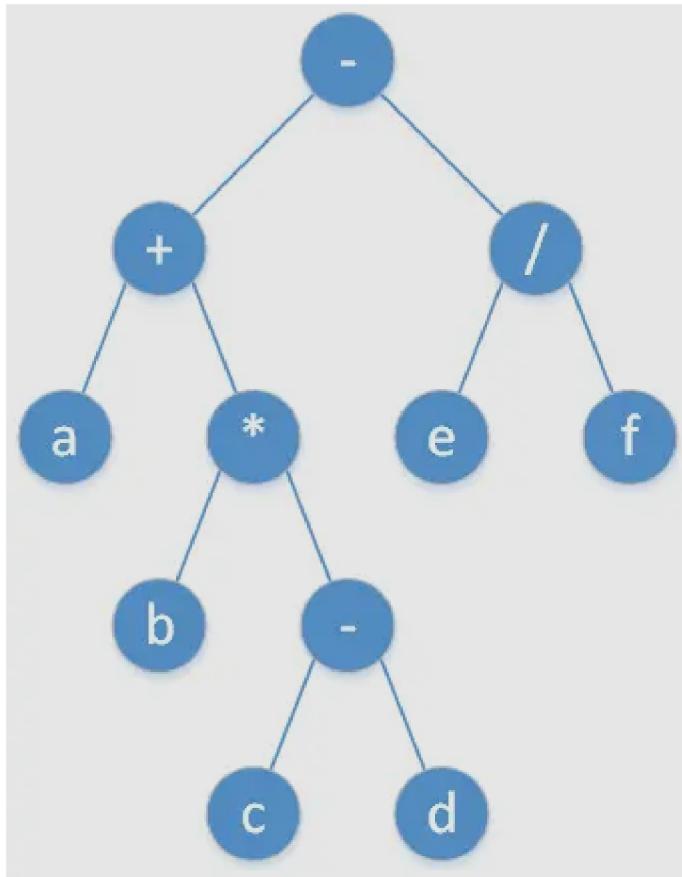
表达式求值要解决的问题一般是输入一个字符串表示的表达式，要求输出它的值。当然也有更复杂的任务，比如表达式中是否包含括号，指数运算，含多少变量，判断多个表达式是否等价，等等。

表达式一般需要先进行语法分析（grammer parsing）再求值，也可以边分析边求值，语法分析的作用是检查输入的字符串是否是一个合法的表达式，一般使用语法分析器（parser）解决。

表达式包含两类字符：运算数和运算符。对于长度为  $n$  的表达式，借助合适的分析方法，可以在  $O(n)$  的时间复杂度内完成分析与求值。

## 表达式树与逆波兰表达式

一种递归分析表达式的方法是，将表达式当成普通的语法规则进行分析，分析后拆分成如图所示的表达式树，然后在树结构上自底向上进行运算。



表达式树上进行 [树的遍历](#) 可以得到不同类型的表达式。算术表达式分为三种，分别是前缀表达式、中缀表达式、后缀表达式。中缀表达式是日常生活中最常用的表达式；后缀表达式是计算机容易理解的表达式。

- 前序遍历对应前缀表达式（波兰式）

- 中序遍历对应中缀表达式
- 后序遍历对应后缀表达式（逆波兰式）

逆波兰表达式（后缀表达式）是书写数学表达式的一种形式，其中运算符位于其操作数之后。例如，以下表达式：

$$a + b * c * d + (e - f) * (g * h + i)$$

可以用逆波兰表达式书写：

$$abc * d * +ef - gh * i + *+$$

因此，逆波兰表达式与表达式树一一对应。逆波兰表达式不需要括号表示，它的运算顺序是唯一确定的。

逆波兰表达式的方便之处在于很容易在线性时间内计算。举个例子：在逆波兰表达式  $3 \ 2 \ * \ 1 \ -$  中，首先计算  $3 \times 2 = 6$ （使用最后一个运算符，即栈顶运算符），然后计算  $6 - 1 = 5$ 。可以看到：对于一个逆波兰表达式，只需要 **维护一个数字栈，每次遇到一个运算符，就取出两个栈顶元素，将运算结果重新压入栈中**。最后，栈中唯一一个元素就是该逆波兰表达式的运算结果。该算法拥有  $O(n)$  的时间复杂度。

采用递归的办法分析表达式是否成功，依赖于语法规则的设计是否合理，即，是否能够成功地得到指定的表达式树。例如：

$$a + b * c$$

根据加号与乘号的运算优先级不同，该中缀表达式可能转化为两种不同的表达式树。可见，语法规则的设计高度依赖于运算符的优先级。借助运算符的优先级设计相应递归的语法规则，事实上是一件不容易的事情。

下文介绍的办法将运算符与它的优先级视为一个整体，采用非递归的办法，直接根据运算符的优先级来分析与计算表达式。

## 只含左结合的二元运算符的含括号表达式

考虑简化的问题。假设所有运算符都是二元的：所有运算符都有两个参数。并且所有运算符都是左结合的：如果运算符的优先级相等，则从左到右执行。允许使用括号。

对于这种类型的中缀表达式的计算，可以将其转化为后缀表达式再进行计算。定义两个 **栈** 来分别存储运算符和运算数，每当遇到一个数直接放进运算数栈。每个运算符块对应于一对括号，运算符栈只对于运算符块的内部单调。每当遇到一个操作符时，要查找运算符栈中最顶部运算符块中的元素，在运算符块的内部保持运算符按照优先级降序进行适当的弹出操作，弹出的同时求出对应的子表达式的值。

以下部分用「输出」表示输出到后缀表达式，即将该数字放在运算数栈上，或者弹出运算符和两个操作数，运算后再将结果压回运算数栈上。从左到右扫描该中缀表达式：

1. 如果遇到数字，直接输出该数字。

2. 如果遇到左括号，那么将其放在运算符栈上。
3. 如果遇到右括号，不断输出栈顶元素，直至遇到左括号，左括号出栈。换句话说，执行一对括号内的所有运算符。
4. 如果遇到其他运算符，不断输出所有运算优先级大于等于当前运算符的运算符。最后，新的运算符入运算符栈。
5. 在处理完整个字符串之后，一些运算符可能仍然在堆栈中，因此把栈中剩下的符号依次输出，表达式转换结束。

以下是四个运算符 +、-、\*、/ 的此方法的实现：



## 示例代码

```
1  bool delim(char c) { return c == ' '; }
2
3  bool is_op(char c) { return c == '+' || c == '-' || c == '*' || c
4  == '/'; }
5
6  int priority(char op) {
7      if (op == '+' || op == '-') return 1;
8      if (op == '*' || op == '/') return 2;
9      return -1;
10 }
11
12 void process_op(stack<int>& st, char op) { // 也可以用于计算后缀表
13     int r = st.top(); // 取出栈顶元素，注意顺序
14     st.pop();
15     int l = st.top();
16     st.pop();
17     switch (op) {
18         case '+':
19             st.push(l + r);
20             break;
21         case '-':
22             st.push(l - r);
23             break;
24         case '*':
25             st.push(l * r);
26             break;
27         case '/':
28             st.push(l / r);
29             break;
30     }
31 }
32 }
33 }
34
35 int evaluate(string& s) { // 也可以改造为中缀表达式转换后缀表达式
36     stack<int> st;
37     stack<char> op;
38     for (int i = 0; i < (int)s.size(); i++) {
39         if (delim(s[i])) continue;
40
41         if (s[i] == '(') {
42             op.push('('); // 2. 如果遇到左括号，那么将其放在运算符栈上
43         } else if (s[i] == ')') { // 3. 如果遇到右括号，执行一对括号内
44             的所有运算符
45             while (op.top() != '(') {
46                 process_op(st, op.top());
47                 op.pop(); // 不断输出栈顶元素，直至遇到左括号
48             }
49             op.pop(); // 左括号出栈
50         }
51     }
52 }
```

```

50     } else if (is_op(s[i])) { // 4. 如果遇到其他运算符
51         char cur_op = s[i];
52         while (!op.empty() && priority(op.top()) >=
53             priority(cur_op)) {
54             process_op(st, op.top());
55             op.pop(); // 不断输出所有运算优先级大于等于当前运算符的运算
56             符
57         }
58         op.push(cur_op); // 新的运算符入运算符栈
59     } else { // 1. 如果遇到数字, 直接输出该数字
60         int number = 0;
61         while (i < (int)s.size() && isalnum(s[i]))
62             number = number * 10 + s[i++] - '0';
63             --i;
64             st.push(number);
65         }
66     }
67 }

while (!op.empty()) {
    process_op(st, op.top());
    op.pop();
}
return st.top();
}

```

这种隐式使用逆波兰表达式计算表达式的值的算法的时间复杂度为  $O(n)$ 。通过稍微修改上述实现，还可以以显式形式获得逆波兰表达式。

## 一元运算符与右结合的运算符

现在假设表达式还包含一元运算符，即只有一个参数的运算符。一元加号和一元减号是一元运算符的常见示例。

这种情况的一个区别是，需要确定当前运算符是一元运算符还是二元运算符。

注意到，在一元运算符之前一般有另一个运算符或开括号，如果一元运算符位于表达式的最开头则没有。在二元运算符之前，总是有一个运算数或右括号。因此，可以标记下一个运算符是否一元运算符。

此外，需要以不同的方式执行一元运算符和二元运算符，让一元运算符的优先级高于所有二元运算符。应注意，一些一元运算符，例如一元加号和一元减号，实际上是右结合的。

右结合意味着，每当优先级相等时，必须从右到左计算运算符。

如上所述，一元运算符通常是右结合的。右结合运算符的另一个示例是求幂运算符。对于  $a \wedge b \wedge c$ ，通常被视为  $a^{b^c}$ ，而不是  $(a^b)^c$ 。

为了正确地处理这类运算符，相应的改动是，如果优先级相等，将推迟运算符的出栈操作。

需要改动的代码如下。将：

```
1 | while (!op.empty() && priority(op.top()) >= priority(cur_op))
```

换成

```
1 | while (!op.empty() &&
2 |         ((left_assoc(cur_op) && priority(op.top()) >= priority(cur_op)) ||
3 |          (!left_assoc(cur_op) && priority(op.top()) > priority(cur_op))))
```

其中 `left_assoc` 是一个函数，它决定运算符是否为左结合的。

这里是二进制运算符 +、-、\*、/ 和一元运算符 + 和 - 的实现：

## 示例代码

```
1  bool delim(char c) { return c == ' '; }
2
3  bool is_op(char c) { return c == '+' || c == '-' || c == '*' || c
4  == '/'; }
5
6  bool is_unary(char c) { return c == '+' || c == '-'; }
7
8  int priority(char op) {
9      if (op < 0) // unary operator
10         return 3;
11     if (op == '+' || op == '-') return 1;
12     if (op == '*' || op == '/') return 2;
13     return -1;
14 }
15
16 void process_op(stack<int>& st, char op) {
17     if (op < 0) {
18         int l = st.top();
19         st.pop();
20         switch (-op) {
21             case '+':
22                 st.push(l);
23                 break;
24             case '-':
25                 st.push(-l);
26                 break;
27         }
28     } else { // 取出栈顶元素，注意顺序
29         int r = st.top();
30         st.pop();
31         int l = st.top();
32         st.pop();
33         switch (op) {
34             case '+':
35                 st.push(l + r);
36                 break;
37             case '-':
38                 st.push(l - r);
39                 break;
40             case '*':
41                 st.push(l * r);
42                 break;
43             case '/':
44                 st.push(l / r);
45                 break;
46         }
47     }
48 }
49 }
```

```

50 int evaluate(string& s) {
51     stack<int> st;
52     stack<char> op;
53     bool may_be_unary = true;
54     for (int i = 0; i < (int)s.size(); i++) {
55         if (delim(s[i])) continue;
56
57         if (s[i] == '(') {
58             op.push('('); // 2. 如果遇到左括号, 那么将其放在运算符栈上
59             may_be_unary = true;
60         } else if (s[i] == ')') { // 3. 如果遇到右括号, 执行一对括号内
61             的所有运算符
62             while (op.top() != '(') {
63                 process_op(st, op.top());
64                 op.pop(); // 不断输出栈顶元素, 直至遇到左括号
65             }
66             op.pop(); // 左括号出栈
67             may_be_unary = false;
68         } else if (is_op(s[i])) { // 4. 如果遇到其他运算符
69             char cur_op = s[i];
70             if (may_be_unary && is_unary(cur_op)) cur_op = -cur_op;
71             while (!op.empty() &&
72                     ((cur_op >= 0 && priority(op.top()) >=
73 priority(cur_op)) ||
74                     (cur_op < 0 && priority(op.top()) >
75 priority(cur_op)))) {
76                 process_op(st, op.top());
77                 op.pop(); // 不断输出所有运算优先级大于等于当前运算符的运算
78                 符
79             }
80             op.push(cur_op); // 新的运算符入运算符栈
81             may_be_unary = true;
82         } else { // 1. 如果遇到数字, 直接输出该数字
83             int number = 0;
84             while (i < (int)s.size() && isalnum(s[i]))
85                 number = number * 10 + s[i++] - '0';
86             --i;
87             st.push(number);
88             may_be_unary = false;
89         }
90     }
91
92     while (!op.empty()) {
93         process_op(st, op.top());
94         op.pop();
95     }
96     return st.top();
97 }
```

## 参考资料

本页面主要译自博文 [Разбор выражений. Обратная польская нотация](#) 与其英文翻译版 [Expression parsing](#)。其中俄文版版权协议为 Public Domain + Leave a Link；英文版版权协议为 CC-BY-SA 4.0。

## 延伸阅读

1. [Operator-precedence\\_parser](#)
2. [Shunting yard algorithm](#)

## 习题

1. [NOIP2013 普及组 表达式求值](#)
2. [后缀表达式](#)
3. [Transform the Expression](#)



本页面最近更新：2025/8/30 13:34:30，[更新历史](#)



发现错误？想一起完善？[在 GitHub 上编辑此页！](#)



本页面贡献者：[Ir1d](#), [Anguei](#), [Yanjun-Zhao](#), [HeRaNO](#), [abc1763613206](#), [c8ef](#), [Henry-ZHR](#), [hsfzLZH1](#), [ksyx](#), [sshyw](#), [0x03A6](#), [CCXXXI](#), [Early0v0](#), [Enter-tainer](#), [Great-designer](#), [Konano](#), [littlefrogfromthenorth](#), [Lynricsy](#), [nanmenyangde](#), [shuzhouliu](#), [Siger Young](#), [siger-young](#), [Tiphereth-A](#), [ttyS0](#), [Xeonacid](#)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用



# 随机函数

## 概述

要想使用随机化技巧，前提条件是能够快速生成随机数。本文将介绍生成随机数的常见方法。

### 随机数与伪随机数

说一个单独的数是「随机数」是无意义的，所以下我们都默认讨论「随机数列」，即使提到「随机数」，指的也是「随机数列中的一个元素」。

现有的计算机的运算过程都是确定性的，因此，仅凭借算法来生成真正 **不可预测、不可重复** 的随机数列是不可能的。

然而在绝大部分情况下，我们都不需要如此强的随机性，而只需要所生成的数列在统计学上具有随机数列的种种特征（比如均匀分布、互相独立等等）。这样的数列即称为 **伪随机数** 序列。

随机数与伪随机数在实际生活和算法中的应用举例：

- 抽样调查时往往只需使用伪随机数。这是因为我们本就只关心统计特征。
- 网络安全中往往要用到（比刚刚提到的伪随机数）更强的随机数。这是因为攻击者可能会利用可预测性做文章。
- OI/ICPC 中用到的随机算法，基本都只需要伪随机数。这是因为，这些算法往往是通过引入随机数来把概率引入复杂度分析，从而降低复杂度。这本质上依然只利用了随机数的统计特征。
- 某些随机算法（例如 [Moser 算法](#)）用到了随机数的熵相关的性质，因此必须使用真正的随机数。

## 实现

### rand

用于生成伪随机数，缺点是比较慢，使用时需要 `#include<cstdlib>`。

调用 `rand()` 函数会返回一个  $[0, \text{RAND\_MAX}]$  中的随机非负整数，其中 `RAND_MAX` 是标准库中的一个宏，在 Linux 系统下 `RAND_MAX` 等于  $2^{31} - 1$ 。可以用取模来限制所生成的数的大小。

使用 `rand()` 需要一个随机数种子，可以使用 `srand(seed)` 函数来将随机种子更改为 `seed`，当然不初始化也是可以的。

同一程序使用相同的 `seed` 两次运行，在同一机器、同一编译器下，随机出的结果将会是相同的。

有一个选择是使用当前系统时间来作为随机种子：`srand(time(nullptr))`。

### ⚠ Warning

在 Windows 系统下 `rand()` 返回值的取值范围为  $[0, 2^{15})$ （即 `RAND_MAX` 等于  $2^{15} - 1$ ），当需要生成的数不小于  $2^{15}$  时建议使用 `(rand() << 15 | rand())` 来生成更大的随机数。

关于 `rand()` 和 `rand()%n` 的随机性：

- C/C++ 标准并未关于 `rand()` 所生成随机数的任何方面的质量做任何规定。
- GCC 编译器对 `rand()` 所采用的实现方式，保证了分布的均匀性等基本性质，但具有 低位周期长度短 等明显缺陷。（例如在笔者的机器上，`rand()%2` 所生成的序列的周期长约  $2 \cdot 10^6$ ）
- 即使假设 `rand()` 是均匀随机的，`rand()%n` 也不能保证均匀性，因为  $[0, n)$  中的每个数在  $0\%n, 1\%n, \dots, RAND\_MAX\%n$  中的出现次数可能不相同。

## 预定义随机数生成器

定义了数个特别的流行算法。如没有特别说明，均定义于头文件 `<random>`。

### ⚠ Warning

预定义随机数生成器仅在于 C++11 标准<sup>2</sup>中开始使用。

### mt19937

是一个随机数生成器类，效用同 `rand()`，随机数的范围同 `unsigned int` 类型的取值范围。

其优点是随机数质量高（一个表现为，出现循环的周期更长；其他方面也都至少不逊于 `rand()`），且速度比 `rand()` 快很多。使用时需要 `#include<random>`。

`mt19937` 基于 32 位梅森缠绕器，由松本与西村设计于 1998 年<sup>3</sup>，使用时用其定义一个随机数生成器即可：`std::mt19937 myrand(seed)`，`seed` 可不填，不填 `seed` 则会使用默认随机种子。

`mt19937` 重载了 `operator ()`，需要生成随机数时调用 `myrand()` 即可返回一个随机数。

另一个类似的生成器是 `mt19937_64`，基于 64 位梅森缠绕器，由松本与西村设计于 2000 年，使用方式同 `mt19937`，但随机数范围扩大到了 `unsigned long long` 类型的取值范围。

## 代码示例

```
1 #include <ctime>
2 #include <iostream>
3 #include <random>
4
5 using namespace std;
6
7 int main() {
8     mt19937 myrand(time(nullptr));
9     cout << myrand() << endl;
10    return 0;
11 }
```

### minstd\_rand0

线性同余算法由 Lewis、Goodman 及 Miller 发现于 1969，由 Park 与 Miller 于 1988 采纳为「最小标准」。

计算公式如下，其中  $A, C, M$  为预定义常数。

$$s_i \equiv s_{i-1} \times A + C \pmod{M}$$

`minstd_rand()` 是较新的「最小标准」，为 Park、Miller 和 Stockmeyer 于 1993 推荐。

对于 `minstd_rand0()`， $s$  的类型取 32 位无符号整数， $A$  取 16807， $C$  取 0， $M$  取 2147483647。

对于 `minstd_rand()`， $s$  的类型取 32 位无符号整数， $A$  取 48271， $C$  取 0， $M$  取 2147483647。

### random\_shuffle

用于随机打乱指定序列。使用时需要 `#include<algorithm>`。

使用时传入指定区间的首尾指针或迭代器（左闭右开）即可：`std::random_shuffle(first, last)` 或 `std::random_shuffle(first, last, myrand)`

内部使用的随机数生成器默认为 `rand()`。当然也可以传入自定义的随机数生成器。

关于 `random_shuffle` 的随机性：

- C++ 标准中要求 `random_shuffle` 在所有可能的排列中 等概率 随机选取，但 GCC<sup>4</sup> 编译器 并未 严格执行。
- GCC 中 `random_shuffle` 随机性上的缺陷的原因之一，是因为它使用了 `rand()%n` 这样的写法。如先前所述，这样生成的不是均匀随机的整数。
- 原因之二，是因为 `rand()` 的值域有限。如果所传入的区间长度超过 `RAND_MAX`，将存在某些排列 不可能 被产生<sup>1</sup>。

## ⚠ Warning

`random_shuffle` 已于 C++14 标准中被弃用，于 C++17 标准中被移除。

### shuffle

效用同 `random_shuffle`。使用时需要 `#include<algorithm>`。

区别在于必须使用自定义的随机数生成器：`std::shuffle(first, last, myrand)`。

GCC<sup>4</sup>实现的 `shuffle` 符合 C++ 标准的要求，即在所有可能的排列中等概率随机选取。

下面是用 `rand()` 及 `random_shuffle()` 编写的一个数据生成器。生成数据为「ZJOI2012」灾难的随机小数据。

```
1 #include <algorithm>
2 #include <cstdlib>
3 #include <ctime>
4 #include <iostream>
5
6 int a[100];
7
8 int main() {
9     srand(time(nullptr));
10    int n = rand() % 99 + 1;
11    for (int i = 1; i <= n; i++) a[i] = i;
12    std::cout << n << '\n';
13    for (int i = 1; i <= n; i++) {
14        std::random_shuffle(a + 1, a + i);
15        int cnt = rand() % i;
16        for (int j = 1; j <= cnt; j++) std::cout << a[j] << ' ';
17        std::cout << 0 << '\n';
18    }
19 }
```

下面是用 `mt19937` 及 `shuffle()` 编写的同一个数据生成器。

```
1 #include <algorithm>
2 #include <ctime>
3 #include <iostream>
4 #include <random>
5
6 int a[100];
7
8 int main() {
9     std::mt19937 rng(time(nullptr));
10    int n = rng() % 99 + 1;
11    for (int i = 1; i <= n; i++) a[i] = i;
12    std::cout << n << '\n';
13    for (int i = 1; i <= n; i++) {
14        std::shuffle(a + 1, a + i, rng);
```

```
15     int cnt = rng() % i;
16     for (int j = 1; j <= cnt; j++) std::cout << a[j] << ' ';
17     std::cout << 0 << '\n';
18 }
19 }
```

下面是随机排列前十个正整数的一个实现。

```
1 #include <algorithm>
2 #include <iostream>
3 #include <iterator>
4 #include <random>
5
6 int main() {
7     std::vector<int> v = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
8
9     std::random_device rd;
10    std::mt19937 g(rd());
11
12    std::shuffle(v.begin(), v.end(), g);
13
14    std::copy(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, " "
15 ));;
16    std::cout << "\n";
}
```

## 非确定随机数的均匀分布整数随机数生成器

`random_device` 是一个基于硬件的均匀分布随机数生成器，在熵池耗尽前可以高速生成随机数。该类在 C++11 定义，需要 `random` 头文件。由于熵池耗尽后性能急剧下降，所以建议用此方法生成 `mt19937` 等伪随机数的种子，而不是直接生成。

`random_device` 是非确定的均匀随机数生成器，尽管若不支持非确定随机数生成，则允许实现用伪随机数引擎实现。目前笔者尚未接到报告称 NOIP 评测机不支持基于硬件的均匀分布随机数生成。但出于保守考虑，建议使用该算法生成随机数种子。

参考代码如下。

```
1 #include <iostream>
2 #include <map>
3 #include <random>
4 #include <string>
5
6 int main() {
7     std::random_device rd;
8     std::map<int, int> hist;
9     std::uniform_int_distribution<int> dist(0, 9);
10    for (int n = 0; n < 20000; ++n) {
11        ++hist[dist(rd)]; // 注意：仅用于演示：一旦熵池耗尽，
12                           // 许多 random_device 实现的性能就急剧下滑
13                           // 对于实践使用，random_device 通常仅用于
14                           // 播种类似 mt19937 的伪随机数生成器
```

```
15     }
16     for (auto p : hist) {
17         std::cout << p.first << " : " << std::string(p.second / 100, '*') <<
18         '\n';
19     }
}
```

可能的输出如下。

```
1 0 : ****
2 1 : ****
3 2 : ****
4 3 : ****
5 4 : ****
6 5 : ****
7 6 : ****
8 7 : ****
9 8 : ****
10 9 : ****
```

## 随机数分布

这里介绍的是要求生成的随机数按照一定的概率出现，如等概率，[伯努利分布](#)，[二项分布](#)，[几何分布](#)，[标准正态（高斯）分布](#)。

具体类名请参见[伪随机数生成—随机数分布](#)的列表。

### 实现

下面的程序模拟了一个六面体骰子。

```
1 #include <iostream>
2 #include <random>
3
4 int main() {
5     std::random_device rd;    // 将用于为随机数引擎获得种子
6     std::mt19937 gen(rd());  // 以播种标准 mersenne_twister_engine
7     std::uniform_int_distribution<> dis(1, 6);
8
9     for (int n = 0; n < 10; ++n)
10        // 用 dis 变换 gen 所生成的随机 unsigned int 到 [1, 6] 中的 int
11        std::cout << dis(gen) << ' ';
12        std::cout << '\n';
13 }
```

## 其他实现方法

有的时候我们需要实现自己的随机数生成器。下面是一些常用的随机数生成方法。

### 线性同余随机数生成器

利用下式来生成随机数序列  $\{R_i\}$ :

$$R_{i+1} = (A \times R_i + B) \bmod P$$

其中  $A, B, P$  均为常数。

该方法实现难度低，但生成的随机序列周期长度较短（周期最大为  $P$ ，但大多数情况下都会比  $P$  短）。

### 参考实现

```
1 #include <iostream>
2 using namespace std;
3
4 struct myrand {
5     int A, B, P, x;
6
7     myrand(int A, int B, int P) {
8         this->A = A;
9         this->B = B;
10        this->P = P;
11    }
12
13    // 生成随机序列的下一个随机数
14    int next() { return x = (A * x + B) % P; }
15};
16
17 myrand rnd(3, 5, 97); // 初始化一个随机数生成器
18
19 int main() {
20     int x = rnd.next();
21     cout << x << endl;
22     return 0;
23 }
```

## 时滞斐波那契随机数生成器

利用下式来生成随机数序列  $\{R_i\}$  (其中  $0 < j < k$ ):

$$R_i \equiv R_{i-j} * R_{i-k} \bmod P$$

这里的  $P$  通常取 2 的幂 (常用  $2^{32}$  或  $2^{64}$ )， $*$  表示二元运算符，可以使用加法，减法，乘法，异或。

该方法较传统的线性同余随机数生成器而言，拥有更长的周期，但随机性受初始条件影响较大。



## 参考实现

```
1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 struct myrand {
6     vector<unsigned> vec;
7     int l, j, k, cur;
8
9     myrand(int l, int j, int k) {
10         this->l = l;
11         this->j = j;
12         this->k = k;
13         cur = 0;
14         for (int i = 0; i < l; i++) {
15             vec.push_back(rand()); // 先用其他方法生成随机序列中的前几个
16             元素
17         }
18     }
19
20     unsigned next() {
21         vec[cur] = vec[(cur - j + l) % l] * vec[(cur - k + l) % l];
22         // 这里用 unsigned 类型是为了实现自动对 2^32 取模
23         return vec[cur++];
24     }
25 };
26
27 myrand rnd(11, 4, 7);
28
29 int main() {
30     unsigned x = rnd.next();
31     cout << x << endl;
32     return 0;
33 }
```

## 参考资料与注释

1. [Don't use rand\(\): a guide to random number generators in C++ ↪](#)
2. [伪随机数生成 - cppreference.com ↪](#)
3. [Mersenne Twister algorithm ↪](#)
4. 版本号为 GCC 9.2.0 ↪ ↪

 本页面最近更新：2024/10/9 22:38:42，[更新历史](#)

 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

 本页面贡献者：[Ir1d](#), [TianyiQ](#), [StudyingFather](#), [partychicken](#), [Henry-ZHR](#), [Marcythm](#), [ouuan](#), [Tiphereth-A](#), [Xeonacid](#), [Arielfoever](#), [CCXXXI](#), [Enter-tainer](#), [ksyx](#), [R-G-Mocoratioen](#), [Vivian Heleneto](#), [woruo27](#)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用



# 404 Not Found

🔧 本页面最近更新：无更新，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：(自动生成)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用





# 404 Not Found

🔧 本页面最近更新：无更新，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：(自动生成)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用



# 动态规划基础

本页面主要介绍了动态规划的基本思想，以及动态规划中状态及状态转移方程的设计思路，帮助各位初学者对动态规划有一个初步的了解。

本部分的其他页面，将介绍各种类型问题中动态规划模型的建立方法，以及一些动态规划的优化技巧。

## 引入



### [IOI1994] 数字三角形

给定一个  $r$  行的数字三角形 ( $r \leq 1000$ )，需要找到一条从最高点到底部任意处结束的路径，使路径经过数字的和最大。每一步可以走到当前点左下方的点或右下方的点。

1		7	
2		3	8
3		8	1
4		2	7
5	4	5	2
		6	4

在上面这个例子中，最优路径是  $7 \rightarrow 3 \rightarrow 8 \rightarrow 7 \rightarrow 5$ 。

最简单粗暴的思路是尝试所有的路径。因为路径条数是  $O(2^r)$  级别的，这样的做法无法接受。

注意到这样一个事实，一条最优的路径，它的每一步决策都是最优的。

以例题里提到的最优路径为例，只考虑前四步  $7 \rightarrow 3 \rightarrow 8 \rightarrow 7$ ，不存在一条从最顶端到 4 行第 2 个数的权值更大的路径。

而对于每一个点，它的下一步决策只有两种：往左下角或者往右下角（如果存在）。因此只需要记录当前点的最大权值，用这个最大权值执行下一步决策，来更新后续点的最大权值。

这样做还有一个好处：我们成功缩小了问题的规模，将一个问题分成了多个规模更小的问题。要想得到从顶端到第  $r$  行的最优方案，只需要知道从顶端到第  $r - 1$  行的最优方案的信息就可以了。

这时候还存在一个问题：子问题间重叠的部分会有很多，同一个子问题可能会被重复访问多次，效率还是不高。解决这个问题的方法是把每个子问题的解存储下来，通过记忆化的方式限制访问顺序，确保每个子问题只被访问一次。

上面就是动态规划的一些基本思路。下面将会更系统地介绍动态规划的思想。

# 动态规划原理

能用动态规划解决的问题，需要满足三个条件：最优子结构，无后效性和子问题重叠。

## 最优子结构

具有最优子结构也可能是适合用贪心的方法求解。

注意要确保我们考察了最优解中用到的所有子问题。

1. 证明问题最优解的第一个组成部分是做出一个选择；
2. 对于一个给定问题，在其可能的第一步选择中，假定你已经知道哪种选择才会得到最优解。  
你现在并不关心这种选择具体是如何得到的，只是假定已经知道了这种选择；
3. 给定可获得的最优解的选择后，确定这次选择会产生哪些子问题，以及如何最好地刻画子问题空间；
4. 证明作为构成原问题最优解的组成部分，每个子问题的解就是它本身的最优解。方法是反证法，考虑加入某个子问题的解不是其自身的最优解，那么就可以从原问题的解中用该子问题的最优解替换掉当前的非最优解，从而得到原问题的一个更优的解，从而与原问题最优解的假设矛盾。

要保持子问题空间尽量简单，只在必要时扩展。

最优子结构的不同体现在两个方面：

1. 原问题的最优解中涉及多少个子问题；
2. 确定最优解使用哪些子问题时，需要考察多少种选择。

子问题图中每个定点对应一个子问题，而需要考察的选择对应关联至子问题顶点的边。

## 无后效性

已经求解的子问题，不会再受到后续决策的影响。

## 子问题重叠

如果有大量的重叠子问题，我们可以用空间将这些子问题的解存储下来，避免重复求解相同的子问题，从而提升效率。

## 基本思路

对于一个能用动态规划解决的问题，一般采用如下思路解决：

1. 将原问题划分为若干 **阶段**，每个阶段对应若干个子问题，提取这些子问题的特征（称之为**状态**）；

2. 寻找每一个状态的可能 **决策**, 或者说是各状态间的相互转移方式 (用数学的语言描述就是**状态转移方程**)。

3. 按顺序求解每一个阶段的问题。

如果用图论的思想理解, 我们建立一个 **有向无环图**, 每个状态对应图上一个节点, 决策对应节点间的连边。这样问题就转变为了一个在 DAG 上寻找最长 (短) 路的问题 (参见: [DAG 上的 DP](#))。

## 最长公共子序列



### 最长公共子序列问题



给定一个长度为  $n$  的序列  $A$  和一个长度为  $m$  的序列  $B$  ( $n, m \leq 5000$ ), 求出一个最长的序列, 使得该序列既是  $A$  的子序列, 也是  $B$  的子序列。

子序列的定义可以参考 [子序列](#)。一个简要的例子: 字符串 `abcde` 与字符串 `acde` 的公共子序列有 `a`、`c`、`d`、`e`、`ac`、`ad`、`ae`、`cd`、`ce`、`de`、`ade`、`ace`、`cde`、`acde`, 最长公共子序列的长度是 4。

设  $f(i, j)$  表示只考虑  $A$  的前  $i$  个元素,  $B$  的前  $j$  个元素时的最长公共子序列的长度, 求这时的最长公共子序列的长度就是 **子问题**。 $f(i, j)$  就是我们所说的 **状态**, 则  $f(n, m)$  是最终要达到的状态, 即为所求结果。

对于每个  $f(i, j)$ , 存在三种决策: 如果  $A_i = B_j$ , 则可以将它接到公共子序列的末尾; 另外两种决策分别是跳过  $A_i$  或者  $B_j$ 。状态转移方程如下:

$$f(i, j) = \begin{cases} f(i - 1, j - 1) + 1 & A_i = B_j \\ \max(f(i - 1, j), f(i, j - 1)) & A_i \neq B_j \end{cases}$$

可参考 [SourceForge 的 LCS 交互网页](#) 来更好地理解 LCS 的实现过程。

该做法的时间复杂度为  $O(nm)$ 。

另外, 本题存在  $O\left(\frac{nm}{w}\right)$  的算法<sup>1</sup>。有兴趣的同学可以自行探索。

```
1 int a[MAXN], b[MAXM], f[MAXN][MAXM];
2
3 int dp() {
4     for (int i = 1; i <= n; i++)
5         for (int j = 1; j <= m; j++)
6             if (a[i] == b[j])
7                 f[i][j] = f[i - 1][j - 1] + 1;
8             else
9                 f[i][j] = std::max(f[i - 1][j], f[i][j - 1]);
10            return f[n][m];
11 }
```

## 最长不下降子序列

### 最长不下降子序列问题

给定一个长度为  $n$  的序列  $A$  ( $n \leq 5000$ )，求出一个最长的  $A$  的子序列，满足该子序列的后一个元素不小于前一个元素。

### 算法一

设  $f(i)$  表示以  $A_i$  为结尾的最长不下降子序列的长度，则所求为  $\max_{1 \leq i \leq n} f(i)$ 。

计算  $f(i)$  时，尝试将  $A_i$  接到其他的最长不下降子序列后面，以更新答案。于是可以写出这样的状态转移方程： $f(i) = \max_{1 \leq j < i, A_j \leq A_i} (f(j) + 1)$ 。

容易发现该算法的时间复杂度为  $O(n^2)$ 。

#### C++

```
1 int a[MAXN], d[MAXN];
2
3 int dp() {
4     d[1] = 1;
5     int ans = 1;
6     for (int i = 2; i <= n; i++) {
7         d[i] = 1;
8         for (int j = 1; j < i; j++)
9             if (a[j] <= a[i]) {
10                 d[i] = max(d[i], d[j] + 1);
11                 ans = max(ans, d[i]);
12             }
13     }
14     return ans;
15 }
```

#### Python

```
1 a = [0] * MAXN
2 d = [0] * MAXN
3
4 def dp():
5     d[1] = 1
6     ans = 1
7     for i in range(2, n + 1):
8         for j in range(1, i):
9             if a[j] <= a[i]:
10                 d[i] = max(d[i], d[j] + 1)
11                 ans = max(ans, d[i])
12
13 return ans
```



## 算法二<sup>2</sup>

当  $n$  的范围扩大到  $n \leq 10^5$  时，第一种做法就不够快了，下面给出了一个  $O(n \log n)$  的做法。

回顾一下之前的状态： $(i, l)$ 。

但这次，我们不是要按照相同的  $i$  处理状态，而是直接判断合法的  $(i, l)$ 。

再看一下之前的转移： $(j, l - 1) \rightarrow (i, l)$ ，就可以判断某个  $(i, l)$  是否合法。

初始时  $(1, 1)$  肯定合法。

那么，只需要找到一个  $l$  最大的合法的  $(i, l)$ ，就可以得到最终最长不下降子序列的长度了。

那么，根据上面的方法，我们就需要维护一个可能的转移列表，并逐个处理转移。

所以可以定义  $a_1 \dots a_n$  为原始序列， $d_i$  为所有的长度为  $i$  的不下降子序列的末尾元素的最小值， $len$  为子序列的长度。

初始化： $d_1 = a_1, len = 1$ 。

现在我们已知最长的不下降子序列长度为 1，那么我们让  $i$  从 2 到  $n$  循环，依次求出前  $i$  个元素的最长不下降子序列的长度，循环的时候我们只需要维护好  $d$  这个数组还有  $len$  就可以了。**关键在于如何维护。**

考虑进来一个元素  $a_i$ ：

1. 元素大于等于  $d_{len}$ ，直接将该元素插入到  $d$  序列的末尾。
2. 元素小于  $d_{len}$ ，找到 **第一个** 大于它的元素，用  $a_i$  替换它。

为什么：

- 对于步骤 1：

由于我们是从前往后扫，所以说当元素大于等于  $d_{len}$  时一定会有一个不下降子序列使得这个不下降子序列的末项后面可以接这个元素。如果  $d$  不接这个元素，可以发现既不符合定义，又不是最优解。

- 对于步骤 2：

同步骤 1，如果插在  $d$  的末尾，那么由于前面的元素大于要插入的元素，所以不符合  $d$  的定义，因此必须先找到 **第一个** 大于它的元素，再用  $a_i$  替换。

步骤 2 如果采用暴力查找，则时间复杂度仍然是  $O(n^2)$  的。但是根据  $d$  数组的定义，又由于本题要求不下降子序列，所以  $d$  一定是 **单调不减** 的，因此可以用二分查找将时间复杂度降至  $O(n \log n)$ 。

参考代码如下：

## C++

```
1 for (int i = 0; i < n; ++i) scanf("%d", a + i);
2 memset(dp, 0x1f, sizeof dp);
3 mx = dp[0];
4 for (int i = 0; i < n; ++i) {
5     *std::upper_bound(dp, dp + n, a[i]) = a[i];
6 }
7 ans = 0;
8 while (dp[ans] != mx) ++ans;
```

## Python

```
1 dp = [0x1f1f1f1f] * MAXN
2 mx = dp[0]
3 for i in range(0, n):
4     bisect.insort_left(dp, a[i], 0, len(dp))
5 ans = 0
6 while dp[ans] != mx:
7     ans += 1
```

### ⚠ 注意

对于最长 **上升** 子序列问题，类似地，可以令  $d_i$  表示所有长度为  $i$  的最长上升子序列的末尾元素的最小值。

需要注意的是，在步骤 2 中，若  $a_i \leq d_{len}$ ，由于最长上升子序列中相邻元素不能相等，需要在  $d$  序列中找到 **第一个不小于**  $a_i$  的元素，用  $a_i$  替换之。

在实现上（以 C++ 为例），需要将 `upper_bound` 函数改为 `lower_bound`。

## 参考资料与注释

1. 位运算求最长公共子序列 - -Wallace- - 博客园 [←](#)
2. 最长不下降子序列 nlogn 算法详解 - lvmememe - 博客园 [←](#)

🔑 本页面最近更新：2025/5/3 19:43:25，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：[lr1d](#), [Chrogeek](#), [ChungZH](#), [ouuan](#), [Xeonacid](#), [CBW2007](#), [ksyx](#), [Marcythm](#), [StudyingFather](#), [tptpp](#), [Enter-tainer](#), [greyqz](#), [HeRaNO](#), [hhc0001](#), [hsfzLZH1](#), [partychicken](#), [Persdre](#), [xhn16729](#), [XiaoSuan250](#), [xyf007](#), [zhb2000](#), [c-forrest](#), [caoji2001](#), [dong628](#), [iamtwz](#), [LincolnYe](#), [Menci](#), [NachtgeistW](#), [ree-chee](#), [shawlleyw](#), [shuzhouliu](#), [Taoran-01](#), [Taoran\\\_01](#), [Tiphereth-A](#), [TrisolarisHD](#), [WAAutoMaton](#), [xhn16729](#), [yusancy](#), [ZhangZhanhaoxiang](#),

[zchen20](#), [zzhx2006](#)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用



# 区间 DP

## 定义

区间类动态规划是线性动态规划的扩展，它在分阶段地划分问题时，与阶段中元素出现的顺序和由前一阶段的哪些元素合并而来有很大的关系。

令状态  $f(i, j)$  表示将下标位置  $i$  到  $j$  的所有元素合并能获得的价值的最大值，那么  $f(i, j) = \max\{f(i, k) + f(k + 1, j) + cost\}$ ,  $cost$  为将这两组元素合并起来的价值。

## 性质

区间 DP 有以下特点：

**合并：**即将两个或多个部分进行整合，当然也可以反过来；

**特征：**能将问题分解为能两两合并的形式；

**求解：**对整个问题设最优值，枚举合并点，将问题分解为左右两个部分，最后合并两个部分的最优值得到原问题的最优值。

## 解释

### 例题



#### 「NOI1995」石子合并



题目大意：在一个环上有  $n$  个数  $a_1, a_2, \dots, a_n$ ，进行  $n - 1$  次合并操作，每次操作将相邻的两堆合并成一堆，能获得新的一堆中的石子数量的和的得分。你需要最大化你的得分。

需要考虑不在环上，而在一条链上的情况。

令  $f(i, j)$  表示将区间  $[i, j]$  内的所有石子合并到一起的最大得分。

写出 **状态转移方程**： $f(i, j) = \max\{f(i, k) + f(k + 1, j) + \sum_{t=i}^j a_t\}$  ( $i \leq k < j$ )

令  $sum_i$  表示  $a$  数组的前缀和，状态转移方程变形为

$f(i, j) = \max\{f(i, k) + f(k + 1, j) + sum_j - sum_{i-1}\}$ 。

## 怎样进行状态转移

由于计算  $f(i, j)$  的值时需要知道所有  $f(i, k)$  和  $f(k + 1, j)$  的值，而这两个中包含的元素的数量都小于  $f(i, j)$ ，所以我们以  $len = j - i + 1$  作为 DP 的阶段。首先从小到大枚举  $len$ ，然后枚举  $i$  的值，根据  $len$  和  $i$  用公式计算出  $j$  的值，然后枚举  $k$ ，时间复杂度为  $O(n^3)$

## 怎样处理环

题目中石子围成一个环，而不是一条链，怎么办呢？

**方法一：**由于石子围成一个环，我们可以枚举分开的位置，将这个环转化成一个链，由于要枚举  $n$  次，最终的时间复杂度为  $O(n^4)$ 。

**方法二：**我们将这条链延长两倍，变成  $2 \times n$  堆，其中第  $i$  堆与第  $n + i$  堆相同，用动态规划求解后，取  $f(1, n), f(2, n + 1), \dots, f(n, 2n - 1)$  中的最优值，即为最后的答案。时间复杂度  $O(n^3)$ 。

## 实现

C++

```
1 for (len = 2; len <= n; len++)  
2     for (i = 1; i <= 2 * n - len; i++) {  
3         int j = len + i - 1;  
4         for (k = i; k < j; k++)  
5             f[i][j] = max(f[i][j], f[i][k] + f[k + 1][j] + sum[j] - sum[i - 1]);  
6     }
```

Python

```
1 for len in range(2, n + 1):  
2     for i in range(1, 2 * n - len + 1):  
3         j = len + i - 1  
4         for k in range(i, j):  
5             f[i][j] = max(f[i][j], f[i][k] + f[k + 1][j] + sum[j] - sum[i - 1])
```

## 几道练习题

[NOIP 2006 能量项链](#)

[NOIP 2007 矩阵取数游戏](#)

[「IOI2000」邮局](#)

 本页面最近更新：2025/8/30 13:34:30，[更新历史](#)

 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

 本页面贡献者：[Ir1d](#), [Henry-ZHR](#), [hsfzLZH1](#), [iamtwz](#), [ouuan](#), [Xeonacid](#), [AFOBJECT](#), [billchenchina](#), [Chlero](#), [EarlyOvO](#), [Enter-tainer](#), [fyulingi](#), [greyqz](#), [HeRaNO](#), [ImpleLee](#), [isdanni](#), [ksyx](#), [Menci](#), [OYoooooo](#), [partychicken](#), [shawlleyw](#), [shenshuaijie](#), [StudyingFather](#), [thredreams](#), [Wang Hongtian](#)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用



# 404 Not Found

🔧 本页面最近更新：无更新，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：(自动生成)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用





# 404 Not Found

🔧 本页面最近更新：无更新，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：(自动生成)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用





# 404 Not Found

🔧 本页面最近更新：无更新，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：(自动生成)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用





# 404 Not Found

🔧 本页面最近更新：无更新，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：(自动生成)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用





# 404 Not Found

🔧 本页面最近更新：无更新，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：(自动生成)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用





# 404 Not Found

🔧 本页面最近更新：无更新，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：(自动生成)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用



# 树形 DP

树形 DP，即在树上进行的 DP。由于树固有的递归性质，树形 DP 一般都是递归进行的。



## 基础

以下面这道题为例，介绍一下树形 DP 的一般过程。



### 例题 洛谷 P1352 没有上司的舞会



某大学有  $n$  个职员，编号为  $1 \sim N$ 。他们之间有从属关系，也就是说他们的关系就像一棵以校长为根的树，父结点就是子结点的直接上司。现在有个周年庆宴会，宴会每邀请来一个职员都会增加一定的快乐指数  $a_i$ ，但是呢，如果某个职员的直接上司来参加舞会了，那么这个职员就无论如何也不肯来参加舞会了。所以，请你编程计算，邀请哪些职员可以使快乐指数最大，求最大的快乐指数。

我们设  $f(i, 0/1)$  代表以  $i$  为根的子树的最优解（第二维的值为 0 代表  $i$  不参加舞会的情况，1 代表  $i$  参加舞会的情况）。

对于每个状态，都存在两种决策（其中下面的  $x$  都是  $i$  的儿子）：

- 上司不参加舞会时，下属可以参加，也可以不参加，此时有
$$f(i, 0) = \sum \max\{f(x, 1), f(x, 0)\};$$
- 上司参加舞会时，下属都不会参加，此时有
$$f(i, 1) = \sum f(x, 0) + a_i.$$

我们可以通过 DFS，在返回上一层时更新当前结点的最优解。

```
1 #include <algorithm>
2 #include <iostream>
3 using namespace std;
4
5 struct edge {
6     int v, next;
7 } e[6005];
8
9 int head[6005], n, cnt, f[6005][2], ans, is_h[6005], vis[6005];
10
11 void addedge(int u, int v) { // 建图
12     e[++cnt].v = v;
13     e[cnt].next = head[u];
14     head[u] = cnt;
15 }
16
17 void calc(int k) {
```

```

18     vis[k] = 1;
19     for (int i = head[k]; i; i = e[i].next) { // 枚举该结点的每个子结点
20         if (vis[e[i].v]) continue;
21         calc(e[i].v);
22         f[k][1] += f[e[i].v][0];
23         f[k][0] += max(f[e[i].v][0], f[e[i].v][1]); // 转移方程
24     }
25     return;
26 }
27
28 int main() {
29     cin.tie(nullptr)->sync_with_stdio(false);
30     cin >> n;
31     for (int i = 1; i <= n; i++) cin >> f[i][1];
32     for (int i = 1; i < n; i++) {
33         int l, k;
34         cin >> l >> k;
35         is_h[l] = 1;
36         addedge(k, l);
37     }
38     for (int i = 1; i <= n; i++)
39         if (!is_h[i]) { // 从根结点开始DFS
40             calc(i);
41             cout << max(f[i][1], f[i][0]);
42             return 0;
43         }
44     }
}

```

通常，树形 DP 状态一般都为当前节点的最优解。先 DFS 遍历子树的所有最优解，然后向上传递给子树的父节点来转移，最终根节点的值即为所求的最优解。

## 习题

- [HDU 2196 Computer](#)
- [POJ 1463 Strategic game](#)
- [\[POI2014\]FAR-FarmCraft](#)

## 树上背包

树上的背包问题，简单来说就是背包问题与树形 DP 的结合。



### 例题 洛谷 P2014 CTSC1997 选课



现在有  $n$  门课程，第  $i$  门课程的学分为  $a_i$ ，每门课程有零门或一门先修课，有先修课的课程需要先学完其先修课，才能学习该课程。

一位学生要学习  $m$  门课程，求其能获得的最多学分数。

$n, m \leq 300$

每门课最多只有一门先修课的特点，与有根树中一个点最多只有一个父亲结点的特点类似。

因此可以想到根据这一性质建树，从而所有课程组成了一个森林的结构。为了方便起见，我们可以新增一门 0 学分的课程（设这个课程的编号为 0），作为所有无先修课课程的先修课，这样我们就将森林变成了一棵以 0 号课程为根的树。

我们设  $f(u, i, j)$  表示以  $u$  号点为根的子树中，已经遍历了  $u$  号点的前  $i$  棵子树，选了  $j$  门课程的最大学分。

转移的过程结合了树形 DP 和 背包 DP 的特点，我们枚举  $u$  点的每个子结点  $v$ ，同时枚举以  $v$  为根的子树选了几门课程，将子树的结果合并到  $u$  上。

记点  $x$  的儿子个数为  $s_x$ ，以  $x$  为根的子树大小为  $\text{size}_x$ ，可以写出下面的状态转移方程：

$$f(u, i, j) = \max_{v, k \leq j, k \leq \text{size}_v} f(u, i - 1, j - k) + f(v, s_v, k)$$

注意上面状态转移方程中的几个限制条件，这些限制条件确保了一些无意义的状态不会被访问到。

$f$  的第二维可以很轻松地用滚动数组的方式省略掉，注意这时需要倒序枚举  $j$  的值。

可以证明，该做法的时间复杂度为  $O(nm)^1$ 。

## 参考代码

```
1 #include <algorithm>
2 #include <iostream>
3 #include <vector>
4 using namespace std;
5 int f[305][305], s[305], n, m;
6 vector<int> e[305];
7
8 int dfs(int u) {
9     int p = 1;
10    f[u][1] = s[u];
11    for (auto v : e[u]) {
12        int siz = dfs(v);
13        // 注意下面两重循环的上界和下界
14        // 只考虑已经合并过的子树，以及选的课程数超过 m+1 的状态没有意义
15        for (int i = min(p, m + 1); i; i--)
16            for (int j = 1; j <= siz && i + j <= m + 1; j++)
17                f[u][i + j] = max(f[u][i + j], f[u][i] + f[v][j]); // 转
18        移方程
19        p += siz;
20    }
21    return p;
22 }
23
24 int main() {
25     cin.tie(nullptr)->sync_with_stdio(false);
26     cin >> n >> m;
27     for (int i = 1; i <= n; i++) {
28         int k;
29         cin >> k >> s[i];
30         e[k].push_back(i);
31     }
32     dfs(0);
33     cout << f[0][m + 1];
34     return 0;
}
```

## 习题

- 「CTSC1997」选课
- 「JSOI2018」潜入行动
- 「SDOI2017」苹果树
- 「Codeforces Round 875 Div. 1」 Problem D. Mex Tree

## 换根 DP

树形 DP 中的换根 DP 问题又被称为二次扫描，通常不会指定根结点，并且根结点的变化会对一些值，例如子结点深度和、点权和等产生影响。

通常需要两次 DFS，第一次 DFS 预处理诸如深度，点权和之类的信息，在第二次 DFS 开始运行换根动态规划。

接下来以一些例题来带大家熟悉这个内容。

### 例题 [POI2008]STA-Station

给定一个  $n$  个点的树，请求出一个结点，使得以这个结点为根时，所有结点的深度之和最大。

不妨令  $u$  为当前结点， $v$  为当前结点的子结点。首先需要用  $s_i$  来表示以  $i$  为根的子树中的结点个数，并且有  $s_u = 1 + \sum s_v$ 。显然需要一次 DFS 来计算所有的  $s_i$ ，这次的 DFS 就是预处理，我们得到了以某个结点为根时其子树中的结点总数。

考虑状态转移，这里就是体现 "换根" 的地方了。令  $f_u$  为以  $u$  为根时，所有结点的深度之和。

$f_v \leftarrow f_u$  可以体现换根，即以  $u$  为根转移到以  $v$  为根。显然在换根的转移过程中，以  $v$  为根或以  $u$  为根会导致其子树中的结点的深度产生改变。具体表现为：

- 所有在  $v$  的子树上的结点深度都减少了一，那么总深度和就减少了  $s_v$ ；
- 所有不在  $v$  的子树上的结点深度都增加了一，那么总深度和就增加了  $n - s_v$ ；

根据这两个条件就可以推出状态转移方程  $f_v = f_u - s_v + n - s_v = f_u + n - 2 \times s_v$ 。

于是在第二次 DFS 遍历整棵树并状态转移  $f_v = f_u + n - 2 \times s_v$ ，那么就能求出以每个结点为根时的深度和了。最后只需要遍历一次所有根结点深度和就可以求出答案。



## 参考代码

```
1 #include <iostream>
2 using namespace std;
3
4 int head[1000010 << 1], tot;
5 long long n, sz[1000010], dep[1000010];
6 long long f[1000010];
7
8 struct node {
9     int to, next;
10 } e[1000010 << 1];
11
12 void add(int u, int v) { // 建图
13     e[++tot] = {v, head[u]};
14     head[u] = tot;
15 }
16
17 void dfs(int u, int fa) { // 预处理dfs
18     sz[u] = 1;
19     dep[u] = dep[fa] + 1;
20     for (int i = head[u]; i; i = e[i].next) {
21         int v = e[i].to;
22         if (v != fa) {
23             dfs(v, u);
24             sz[u] += sz[v];
25         }
26     }
27 }
28
29 void get_ans(int u, int fa) { // 第二次dfs换根dp
30     for (int i = head[u]; i; i = e[i].next) {
31         int v = e[i].to;
32         if (v != fa) {
33             f[v] = f[u] - sz[v] * 2 + n;
34             get_ans(v, u);
35         }
36     }
37 }
38
39 int main() {
40     cin.tie(nullptr)->sync_with_stdio(false);
41     cin >> n;
42     int u, v;
43     for (int i = 1; i <= n - 1; i++) {
44         cin >> u >> v;
45         add(u, v);
46         add(v, u);
47     }
48     dfs(1, 1);
49     for (int i = 1; i <= n; i++) f[1] += dep[i];
```

```
50     get_ans(1, 1);
51     long long int ans = -1;
52     int id;
53     for (int i = 1; i <= n; i++) { // 统计答案
54         if (f[i] > ans) {
55             ans = f[i];
56             id = i;
57         }
58     }
59     cout << id << '\n';
60     return 0;
61 }
```

## 习题

- Atcoder Educational DP Contest, Problem V, Subtree
- Educational Codeforces Round 67, Problem E, Tree Painting
- POJ 3585 Accumulation Degree
- [USACO10MAR]Great Cow Gathering G
- CodeForce 708C Centroids

## 参考资料与注释

- 
1. 子树合并背包类型的 dp 的复杂度证明 - LYD729 的 CSDN 博客 ↪



本页面最近更新: 2025/7/13 17:32:21, [更新历史](#)



发现错误? 想一起完善? [在 GitHub 上编辑此页!](#)



本页面贡献者: [StudyingFather](#), [H-J-Granger](#), [Ir1d](#), [NachtgeistW](#), [countercurrent-time](#), [Early0v0](#), [Enter-tainer](#), [ShaoChenHeng](#), [sshwy](#), [aaron20100919](#), [AngelKitty](#), [CCXXXI](#), [cjsoft](#), [diauweb](#), [ezoixx130](#), [GekkaSaori](#), [greyqz](#), [Henry-ZHR](#), [Konano](#), [LovelyBuggies](#), [lychees](#), [Makkiy](#), [mgt](#), [minghu6](#), [ouuan](#), [P-Y-Y](#), [PotassiumWings](#), [SamZhangQingChuan](#), [Suyun514](#), [Tiphereth-A](#), [weiyong1024](#), [amakerlife](#), [billchenchina](#), [GavinZhengOI](#), [Gesrua](#), [isdanni](#), [kenlig](#), [ksyx](#), [kxccc](#), [Marcythm](#), [Peanut-Tang](#), [qz-cqy](#), [ShizubaAki](#), [SukkaW](#), [thredreams](#), [widsnoy](#), [Xeonacid](#)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供, 附加条款亦可能应用



# 概率 DP

## 引入

概率 DP 用于解决概率问题与期望问题，建议先对 [概率 & 期望](#) 的内容有一定了解。一般情况下，解决概率问题需要顺序循环，而解决期望问题使用逆序循环，如果定义的状态转移方程存在后效性问题，还需要用到 [高斯消元](#) 来优化。概率 DP 也会结合其他知识进行考察，例如 [状态压缩](#)，树上进行 DP 转移等。

## DP 求概率

这类题目采用顺推，也就是从初始状态推向结果。同一般的 DP 类似的，难点依然对状态转移方程的刻画，只是这类题目经过了概率论知识的包装。



### 例题 [Codeforces 148 D Bag of mice](#)



题目大意：袋子里有  $w$  只白鼠和  $b$  只黑鼠，公主和龙轮流从袋子里抓老鼠。谁先抓到白色老鼠谁就赢，如果袋子里没有老鼠了并且没有谁抓到白色老鼠，那么算龙赢。公主每次抓一只老鼠，龙每次抓完一只老鼠之后会有一只老鼠跑出来。每次抓的老鼠和跑出来的老鼠都是随机的。公主先抓。问公主赢的概率。

## 过程

设  $f_{i,j}$  为轮到公主时袋子里有  $i$  只白鼠， $j$  只黑鼠，公主赢的概率。初始化边界， $f_{0,j} = 0$  因为没有白鼠了算龙赢， $f_{i,0} = 1$  因为抓一只就是白鼠，公主赢。考虑  $f_{i,j}$  的转移：

- 公主抓到一只白鼠，公主赢了。概率为  $\frac{i}{i+j}$ ；
- 公主抓到一只黑鼠，龙抓到一只白鼠，龙赢了。概率为  $\frac{j}{i+j} \cdot \frac{i}{i+j-1}$ ；
- 公主抓到一只黑鼠，龙抓到一只黑鼠，跑出来一只黑鼠，转移到  $f_{i,j-3}$ 。概率为  $\frac{j}{i+j} \cdot \frac{j-1}{i+j-1} \cdot \frac{j-2}{i+j-2}$ ；
- 公主抓到一只黑鼠，龙抓到一只黑鼠，跑出来一只白鼠，转移到  $f_{i-1,j-2}$ 。概率为  $\frac{j}{i+j} \cdot \frac{j-1}{i+j-1} \cdot \frac{i}{i+j-2}$ ；

考虑公主赢的概率，第二种情况不参与计算。并且要保证后两种情况合法，所以还要判断  $i, j$  的大小，满足第三种情况至少要有 3 只黑鼠，满足第四种情况要有 1 只白鼠和 2 只黑鼠。

## 实现

## 参考实现

```
1 #include <cstring>
2 #include <iomanip>
3 #include <iostream>
4 using namespace std;
5
6 using ll = long long;
7 int w, b;
8 double dp[1010][1010];
9
10 int main() {
11     cin.tie(nullptr)->sync_with_stdio(false);
12     cin >> w >> b;
13     memset(dp, 0, sizeof(dp));
14     for (int i = 1; i <= w; i++) dp[i][0] = 1; // 初始化
15     for (int i = 1; i <= b; i++) dp[0][i] = 0;
16     for (int i = 1; i <= w; i++) {
17         for (int j = 1; j <= b; j++) { // 以下为题面概率转移
18             dp[i][j] += (double)i / (i + j);
19             if (j >= 3) {
20                 dp[i][j] += (double)j / (i + j) * (j - 1) / (i + j - 1) *
21 (j - 2) /
22                     (i + j - 2) * dp[i][j - 3];
23             }
24             if (i >= 1 && j >= 2) {
25                 dp[i][j] += (double)j / (i + j) * (j - 1) / (i + j - 1) *
26 i /
27                     (i + j - 2) * dp[i - 1][j - 2];
28             }
29         }
30     }
31     cout << fixed << setprecision(9) << dp[w][b] << '\n';
32     return 0;
33 }
```

## 习题

- [CodeForces 148 D Bag of mice](#)
- [POJ3071 Football](#)
- [CodeForces 768 D Jon and Orbs](#)

## DP 求期望

### 例一



## 例题 POJ2096 Collecting Bugs



题目大意：一个软件有  $s$  个子系统，会产生  $n$  种 bug。某人一天发现一个 bug，这个 bug 属于某种 bug 分类，也属于某个子系统。每个 bug 属于某个子系统的概率是  $\frac{1}{s}$ ，属于某种 bug 分类的概率是  $\frac{1}{n}$ 。求发现  $n$  种 bug，且  $s$  个子系统都找到 bug 的期望天数。

### 过程

令  $f_{i,j}$  为已经找到  $i$  种 bug 分类， $j$  个子系统的 bug，达到目标状态的期望天数。这里的目标状态是找到  $n$  种 bug 分类， $s$  个子系统的 bug。那么就有  $f_{n,s} = 0$ ，因为已经达到了目标状态，不需要用更多的天数去发现 bug 了，于是就以目标状态为起点开始递推，答案是  $f_{0,0}$ 。

考虑  $f_{i,j}$  的状态转移：

- $f_{i,j}$ ，发现一个 bug 属于已经发现的  $i$  种 bug 分类， $j$  个子系统，概率为  $p_1 = \frac{i}{n} \cdot \frac{j}{s}$
- $f_{i,j+1}$ ，发现一个 bug 属于已经发现的  $i$  种 bug 分类，不属于已经发现的子系统，概率为  $p_2 = \frac{i}{n} \cdot (1 - \frac{j}{s})$
- $f_{i+1,j}$ ，发现一个 bug 不属于已经发现 bug 分类，属于  $j$  个子系统，概率为  $p_3 = (1 - \frac{i}{n}) \cdot \frac{j}{s}$
- $f_{i+1,j+1}$ ，发现一个 bug 不属于已经发现 bug 分类，不属于已经发现的子系统，概率为  $p_4 = (1 - \frac{i}{n}) \cdot (1 - \frac{j}{s})$

再根据期望的线性性质，就可以得到状态转移方程：

$$f_{i,j} = p_1 \cdot f_{i,j} + p_2 \cdot f_{i,j+1} + p_3 \cdot f_{i+1,j} + p_4 \cdot f_{i+1,j+1} + 1 \\ = \frac{p_2 \cdot f_{i,j+1} + p_3 \cdot f_{i+1,j} + p_4 \cdot f_{i+1,j+1} + 1}{1 - p_1}$$

### 实现

## 参考实现

```
1 #include <iomanip>
2 #include <iostream>
3 using namespace std;
4 int n, s;
5 double dp[1010][1010];
6
7 int main() {
8     cin.tie(nullptr)->sync_with_stdio(false);
9     cin >> n >> s;
10    dp[n][s] = 0;
11    for (int i = n; i >= 0; i--) {
12        for (int j = s; j >= 0; j--) {
13            if (i == n && s == j) continue;
14            dp[i][j] = (dp[i][j + 1] * i * (s - j) + dp[i + 1][j] * (n
15            - i) * j +
16                        dp[i + 1][j + 1] * (n - i) * (s - j) + n * s) /
17                        (n * s - i * j); // 概率转移
18        }
19    }
20    cout << fixed << setprecision(4) << dp[0][0] << '\n';
21    return 0;
}
```

## 例二

### 例题 「NOIP2016」换教室

题目大意：牛牛要上  $n$  个时间段的课，第  $i$  个时间段在  $c_i$  号教室，可以申请换到  $d_i$  号教室，申请成功的概率为  $p_i$ ，至多可以申请  $m$  节课进行交换。第  $i$  个时间段的课上完后要走到第  $i+1$  个时间段的教室，给出一张图  $v$  个教室  $e$  条路，移动会消耗体力，申请哪几门课程可以使他在教室间移动耗费的体力值的总和的期望值最小，也就是求出最小的期望路程和。

### 过程

对于这个无向连通图，先用 Floyd 求出最短路，为后续的状态转移带来便利。以移动一步为一个阶段（从第  $i$  个时间段到达第  $i+1$  个时间段就是移动了一步），那么每一步就有  $p_i$  的概率到  $d_i$ ，不过在所有的  $d_i$  中只能选  $m$  个，有  $1 - p_i$  的概率到  $c_i$ ，求出在  $n$  个阶段走完后的最小期望路程和。定义  $f_{i,j,0/1}$  为在第  $i$  个时间段，连同这一个时间段已经用了  $j$  次换教室的机会，在这个时间段换（1）或者不换（0）教室的最小期望路程和，那么答案就是  $\min\{f_{n,i,0}, f_{n,i,1}\}, i \in [0, m]$ 。注意边界  $f_{1,0,0} = f_{1,1,1} = 0$ 。

考虑  $f_{i,j,0/1}$  的状态转移：

- 如果这一阶段不换，即  $f_{i,j,0}$  可能是由上一次不换的状态转移来的，那么就是  $f_{i-1,j,0} + w_{c_{i-1},c_i}$ ，也有可能是由上一次交换的状态转移来的，这里结合条件概率和全概率的知识分析可以得到  $f_{i-1,j,1} + w_{d_{i-1},c_i} \cdot p_{i-1} + w_{c_{i-1},c_i} \cdot (1 - p_{i-1})$ ，状态转移方程就有

$$f_{i,j,0} = \min(f_{i-1,j,0} + w_{c_{i-1},c_i}, f_{i-1,j,1} + w_{d_{i-1},c_i} \cdot p_{i-1} + w_{c_{i-1},c_i} \cdot (1 - p_{i-1}))$$

- 如果这一阶段交换，即  $f_{i,j,1}$ 。类似地，可能由上一次不换的状态转移来，也可能由上一次换的状态转移来。那么遇到不换的就乘上  $(1 - p_i)$ ，遇到交换的就乘上  $p_i$ ，将所有会出现的情况都枚举一遍进行计算就好了。这里不再赘述各种转移情况，相信通过上一种阶段例子，这里的状态转移应该能够很容易写出来。

## 实现

## 参考实现

```
1 #include <algorithm>
2 #include <iomanip>
3 #include <iostream>
4
5 using namespace std;
6
7 constexpr int MAXN = 2010;
8 int n, m, v, e;
9 int f[MAXN][MAXN], c[MAXN], d[MAXN];
10 double dp[MAXN][MAXN][2], p[MAXN];
11
12 int main() {
13     cin.tie(nullptr)->sync_with_stdio(false);
14     cin >> n >> m >> v >> e;
15     for (int i = 1; i <= n; i++) cin >> c[i];
16     for (int i = 1; i <= n; i++) cin >> d[i];
17     for (int i = 1; i <= n; i++) cin >> p[i];
18     for (int i = 1; i <= v; i++)
19         for (int j = 1; j < i; j++) f[i][j] = f[j][i] = 1e9;
20
21     int u, V, w;
22     for (int i = 1; i <= e; i++) {
23         cin >> u >> V >> w;
24         f[u][V] = f[V][u] = min(w, f[u][V]);
25     }
26
27     for (int k = 1; k <= v; k++)
28         for (int i = 1; i <= n; i++) // 前面的，按照前面的题解进行一个
29             for (int j = 1; j < i; j++)
30                 if (f[i][k] + f[k][j] < f[i][j]) f[i][j] = f[j][i] = f[i]
31 [k] + f[k][j];
32
33     for (int i = 1; i <= n; i++)
34         for (int j = 0; j <= m; j++) dp[i][j][0] = dp[i][j][1] = 1e9;
35
36     dp[1][0][0] = dp[1][1][1] = 0;
37     for (int i = 2; i <= n; i++) // 有后效性方程
38         for (int j = 0; j <= min(i, m); j++) {
39             dp[i][j][0] = min(dp[i - 1][j][0] + f[c[i - 1]][c[i]],
40                               dp[i - 1][j][1] + f[c[i - 1]][c[i]] * (1
41 - p[i - 1]) +
42                               f[d[i - 1]][c[i]] * p[i - 1]);
43             if (j != 0) {
44                 dp[i][j][1] = min(dp[i - 1][j - 1][0] + f[c[i - 1]][d[i]]
45 * p[i] +
46                               f[c[i - 1]][c[i]] * (1 - p[i]),
47                 dp[i - 1][j - 1][1] +
48                               f[c[i - 1]][c[i]] * (1 - p[i - 1]));
49             }
50         }
51 }
```

```

50 * (1 - p[i]) +
51 f[c[i - 1]][d[i]] * (1 - p[i - 1])
52 * p[i] +
53 f[d[i - 1]][c[i]] * (1 - p[i]) *
54 p[i - 1] +
55 f[d[i - 1]][d[i]] * p[i - 1] *
56 p[i]);
57 }
}

double ans = 1e9;
for (int i = 0; i <= m; i++) ans = min(dp[n][i][0], min(dp[n]
[i][1], ans));
cout << fixed << setprecision(2) << ans;

return 0;
}

```

比较这两个问题可以发现，DP 求期望题目在对具体是求一个值或是最优化问题上会对方程得到转移方式有一些影响，但无论是 DP 求概率还是 DP 求期望，总是离不开概率知识和列出、化简计算公式的步骤，在写状态转移方程时需要思考的细节也类似。

## 习题

- [POJ2096 Collecting Bugs](#)
- [HDU3853 LOOPS](#)
- [HDU4035 Maze](#)
- [「NOIP2016」换教室](#)
- [「SCOI2008」奖励关](#)

## 有后效性 DP

### CodeForces 24 D Broken robot ▼

题目大意：给出一个  $n \times m$  的矩阵区域，一个机器人初始在第  $x$  行第  $y$  列，每一步机器人会等概率地选择停在原地，左移一步，右移一步，下移一步，如果机器人在边界则不会往区域外移动，问机器人到达最后一行的期望步数。

## 过程

在  $m = 1$  时每次有  $\frac{1}{2}$  的概率不动，有  $\frac{1}{2}$  的概率向下移动一格，答案为  $2 \cdot (n - x)$ 。设  $f_{i,j}$  为机器人从第  $i$  行第  $j$  列出发到达第  $n$  行的期望步数，最终状态为  $f_{n,j} = 0$ 。由于机器人会等

概率地选择停在原地，左移一步，右移一步，下移一步，考虑  $f_{i,j}$  的状态转移：

- $f_{i,1} = \frac{1}{3} \cdot (f_{i+1,1} + f_{i,2} + f_{i,1}) + 1$
- $f_{i,j} = \frac{1}{4} \cdot (f_{i,j} + f_{i,j-1} + f_{i,j+1} + f_{i+1,j}) + 1$
- $f_{i,m} = \frac{1}{3} \cdot (f_{i,m} + f_{i,m-1} + f_{i+1,m}) + 1$

在行之间由于只能向下移动，是满足无后效性的。在列之间可以左右移动，在移动过程中可能产生环，不满足无后效性。将方程变换后可以得到：

- $2f_{i,1} - f_{i,2} = 3 + f_{i+1,1}$
- $3f_{i,j} - f_{i,j-1} - f_{i,j+1} = 4 + f_{i+1,j}$
- $2f_{i,m} - f_{i,m-1} = 3 + f_{i+1,m}$

由于是逆序的递推，所以每一个  $f_{i+1,j}$  是已知的。由于有  $m$  列，所以右边相当于是一个  $m$  行的列向量，那么左边就是  $m$  行  $m$  列的矩阵。使用增广矩阵，就变成了  $m$  行  $m+1$  列的矩阵，然后进行 [高斯消元](#) 即可解出答案。

## 实现

## 参考实现

```
1 #include <cstdio>
2 #include <cstring>
3 using namespace std;
4
5 constexpr int MAXN = 1e3 + 10;
6
7 double a[MAXN][MAXN], f[MAXN];
8 int n, m;
9
10 void solve(int x) {
11     memset(a, 0, sizeof a);
12     for (int i = 1; i <= m; i++) {
13         if (i == 1) {
14             a[i][i] = 2;
15             a[i][i + 1] = -1;
16             a[i][m + 1] = 3 + f[i];
17             continue;
18         } else if (i == m) {
19             a[i][i] = 2;
20             a[i][i - 1] = -1;
21             a[i][m + 1] = 3 + f[i];
22             continue;
23         }
24         a[i][i] = 3;
25         a[i][i + 1] = -1;
26         a[i][i - 1] = -1;
27         a[i][m + 1] = 4 + f[i];
28     }
29
30     for (int i = 1; i < m; i++) {
31         double p = a[i + 1][i] / a[i][i];
32         a[i + 1][i] = 0;
33         a[i + 1][i + 1] -= a[i][i + 1] * p;
34         a[i + 1][m + 1] -= a[i][m + 1] * p;
35     }
36
37     f[m] = a[m][m + 1] / a[m][m];
38     for (int i = m - 1; i >= 1; i--)
39         f[i] = (a[i][m + 1] - f[i + 1] * a[i][i + 1]) / a[i][i];
40 }
41
42 int main() {
43     scanf("%d %d", &n, &m);
44     int st, ed;
45     scanf("%d %d", &st, &ed);
46     if (m == 1) {
47         printf("%.10f\n", 2.0 * (n - st));
48         return 0;
49     }
```

```
50     for (int i = n - 1; i >= st; i--) {
51         solve(i);
52     }
53     printf("%.10f\n", f[ed]);
54     return 0;
55 }
```

## 习题

- [CodeForce 24 D Broken robot](#)
- [HDU 4418 Time Travel](#)
- [「HNOI2013」游走](#)

## 参考文献

[kuangbin 概率 DP 总结](#)

🔧 本页面最近更新：2025/5/3 19:43:25，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：[Tiphereth-A](#), [ShaoChenHeng](#), [Enter-tainer](#), [ksyx](#), [StudyingFather](#), [c-forrest](#), [H-J-Granger](#), [iamtwz](#), [imp2002](#), [Ir1d](#), [kenlig](#), [LeBronGod](#), [Marcythm](#), [MegaOwler](#), [NachtgeistW](#), [ouuan](#), [Patchouliys](#), [Soohti](#), [TianKong-y](#)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用



# 状压 DP

## 简介

状压 DP 是动态规划的一种，通过将状态集合转化为整数记录在 DP 状态中来实现状态转移的目的。

为了达到更低的时间复杂度，通常需要寻找更低状态数的状态。大部分题目中会利用二元状态，用  $n$  位二进制数表示  $n$  个独立二元状态的情况。

使用状态压缩通常涉及位运算，关于基础位运算详见 [位运算](#) 页面。

## 例题 1



### 「SCOI2005」互不侵犯



在  $N \times N$  的棋盘里面放  $K$  个国王 ( $1 \leq N \leq 9, 1 \leq K \leq N \times N$ )，使他们互不攻击，共有多少种摆放方案。

国王能攻击到它上下左右，以及左上左下右上右下八个方向上附近的各一个格子，共 8 个格子。

## 解释

设  $f(i, j, l)$  表示前  $i$  行，第  $i$  行的状态为  $j$ ，且棋盘上已经放置  $l$  个国王时的合法方案数。

对于编号为  $j$  的状态，我们用二进制整数  $sit(j)$  表示国王的放置情况， $sit(j)$  的某个二进制位为 0 表示对应位置不放国王，为 1 表示在对应位置上放置国王；用  $sta(j)$  表示该状态的国王个数，即二进制数  $sit(j)$  中 1 的个数。例如，如下图所示的状态可用二进制数 100101 来表示（棋盘左边对应二进制低位），则有  $sit(j) = 100101_{(2)} = 37, sta(j) = 3$ 。



设当前行的状态为  $j$ ，上一行的状态为  $x$ ，可以得到下面的状态转移方程：

$$f(i, j, l) = \sum f(i - 1, x, l - sta(j))。$$

设上一行的状态编号为  $x$ , 在保证当前行和上一行不冲突的前提下, 枚举所有可能的  $x$  进行转移, 转移方程:

$$f(i, j, l) = \sum f(i - 1, x, l - sta(j))$$

## 实现

### 参考代码

```
1 #include <algorithm>
2 #include <iostream>
3 using namespace std;
4 long long sta[2005], sit[2005], f[15][2005][105];
5 int n, k, cnt;
6
7 void dfs(int x, int num, int cur) {
8     if (cur >= n) { // 有新的合法状态
9         sit[++cnt] = x;
10        sta[cnt] = num;
11        return;
12    }
13    dfs(x, num, cur + 1); // cur位置不放国王
14    dfs(x + (1 << cur), num + 1,
15         cur + 2); // cur位置放国王, 与它相邻的位置不能再放国王
16 }
17
18 bool compatible(int j, int x) {
19     if (sit[j] & sit[x]) return false;
20     if ((sit[j] << 1) & sit[x]) return false;
21     if (sit[j] & (sit[x] << 1)) return false;
22     return true;
23 }
24
25 int main() {
26     cin >> n >> k;
27     dfs(0, 0, 0); // 先预处理一行的所有合法状态
28     for (int j = 1; j <= cnt; j++) f[1][j][sta[j]] = 1;
29     for (int i = 2; i <= n; i++)
30         for (int j = 1; j <= cnt; j++) {
31             for (int x = 1; x <= cnt; x++) {
32                 if (!compatible(j, x)) continue; // 排除不合法转移
33                 for (int l = sta[j]; l <= k; l++) f[i][j][l] += f[i - 1]
34 [x][l - sta[j]];
35             }
36             long long ans = 0;
37             for (int i = 1; i <= cnt; i++) ans += f[n][i][k]; // 累加答案
38             cout << ans << endl;
39         }
40 }
```

## 例题 2



[POI2004] PRZ

有  $n$  个人需要过桥，第  $i$  的人的重量为  $w_i$ ，过桥用时为  $t_i$ . 这些人过桥时会分成若干组，只有在某一组的所有人全部过桥后，其余的组才能过桥。桥最大承重为  $W$ ，问这些人全部过桥的最短时间。

$$100 \leq W \leq 400, 1 \leq n \leq 16, 1 \leq t_i \leq 50, 10 \leq w_i \leq 100.$$

### 解释

我们用  $S$  表示所有人物构成集合的一个子集，设  $t(S)$  表示  $S$  中人的最长过桥时间， $w(S)$  表示  $S$  中所有人的总重量， $f(S)$  表示  $S$  中所有人全部过桥的最短时间，则：

$$\begin{cases} f(\emptyset) = 0, \\ f(S) = \min_{T \subseteq S; w(T) \leq W} \{t(T) + f(S \setminus T)\}. \end{cases}$$

需要注意的是这里不能直接枚举集合再判断是否为子集，而应使用 [子集枚举](#)，从而使时间复杂度为  $O(3^n)$ .

### 实现

## 参考代码

```
1 #include <iostream>
2 #include <limits>
3 #include <vector>
4 using namespace std;
5
6 int main() {
7     ios::sync_with_stdio(false);
8     cin.tie(nullptr);
9     int W, n;
10    cin >> W >> n;
11    const int S = (1 << n) - 1;
12    vector<int> ts(S + 1), ws(S + 1);
13    for (int j = 0, t, w; j < n; ++j) {
14        cin >> t >> w;
15        for (int i = 0; i <= S; ++i)
16            if (i & (1 << j)) {
17                ts[i] = max(ts[i], t);
18                ws[i] += w;
19            }
20    }
21    vector<int> dp(S + 1, numeric_limits<int>::max() / 2);
22    for (int i = 0; i <= S; ++i) {
23        if (ws[i] <= W) dp[i] = ts[i];
24        for (int j = i; j; j = i & (j - 1))
25            if (ws[i ^ j] <= W) dp[i] = min(dp[i], dp[j] + ts[i ^ j]);
26    }
27    cout << dp[S] << '\n';
28    return 0;
29 }
```

## 习题

- 「NOI2001」炮兵阵地
- 「USACO06NOV」玉米田 Corn Fields
- 「九省联考 2018」一双木棋

🔑 本页面最近更新：2025/8/8 20:46:23，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：[StudyingFather](#), [Ir1d](#), [H-J-Granger](#), [NachtgeistW](#), [countercurrent-time](#), [Enter-tainer](#), [ouuan](#), [Marcyhm](#), [sshwy](#), [AngelKitty](#), [CCXXXI](#), [cjsoft](#), [diauweb](#), [Early0v0](#), [ezoixx130](#), [GekkaSaori](#), [Henry-ZHR](#), [HeRaNO](#), [Konano](#), [LovelyBuggies](#), [Makkiy](#), [mgt](#), [minghu6](#), [P-Y-Y](#), [PotassiumWings](#), [SamZhangQingChuan](#), [Suyun514](#), [weiyong1024](#), [chieh2lu2](#), [Chrogeek](#),

GavinZhengOI, Gesrua, hsfzLZH1, iamtwz, kenlig, ksyx, kxccc, Link-cute, lychees, Peanut-Tang, REYwmp, shinzanmono, SukkaW, TianKong-y, Tiphereth-A, Xeonacid, YuJunDongGit, zhb2000

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用





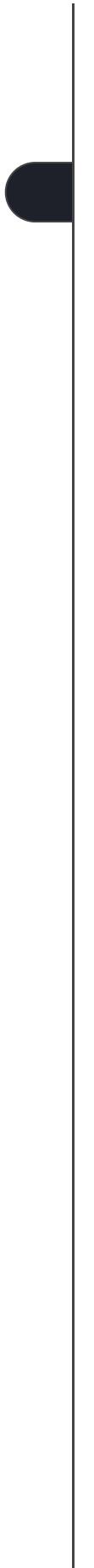
# 404 Not Found

🔧 本页面最近更新：无更新，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：(自动生成)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用





# 404 Not Found

🔧 本页面最近更新：无更新，[更新历史](#)

✍ 发现错误？想一起完善？[在 GitHub 上编辑此页！](#)

👤 本页面贡献者：(自动生成)

© 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用



# 背包 DP

前置知识：[动态规划部分简介](#)。



## 引入

在具体讲解何为「背包 dp」前，先来看如下的例题：



### 「USACO07 DEC」 Charm Bracelet



题意概要：有  $n$  个物品和一个容量为  $W$  的背包，每个物品有重量  $w_i$  和价值  $v_i$  两种属性，要求选若干物品放入背包使背包中物品的总价值最大且背包中物品的总重量不超过背包的容量。

在上述例题中，由于每个物体只有两种可能的状态（取与不取），对应二进制中的 0 和 1，这类问题便被称为「0-1 背包问题」。

## 0-1 背包

### 解释

例题中已知条件有第  $i$  个物品的重量  $w_i$ ，价值  $v_i$ ，以及背包的总容量  $W$ 。

设 DP 状态  $f_{i,j}$  为在只能放前  $i$  个物品的情况下，容量为  $j$  的背包所能达到的最大总价值。

考虑转移。假设当前已经处理好了前  $i - 1$  个物品的所有状态，那么对于第  $i$  个物品，当其不放入背包时，背包的剩余容量不变，背包中物品的总价值也不变，故这种情况的最大价值为  $f_{i-1,j}$ ；当其放入背包时，背包的剩余容量会减小  $w_i$ ，背包中物品的总价值会增大  $v_i$ ，故这种情况的最大价值为  $f_{i-1,j-w_i} + v_i$ 。

由此可以得出状态转移方程：

$$f_{i,j} = \max(f_{i-1,j}, f_{i-1,j-w_i} + v_i)$$

这里如果直接采用二维数组对状态进行记录，会出现 MLE。可以考虑改用滚动数组的形式来优化。

由于对  $f_i$  有影响的只有  $f_{i-1}$ ，可以去掉第一维，直接用  $f_i$  来表示处理到当前物品时背包容量为  $i$  的最大价值，得出以下方程：

$$f_j = \max(f_j, f_{j-w_i} + v_i)$$

务必牢记并理解这个转移方程，因为大部分背包问题的转移方程都是在此基础上推导出来的。

## 实现

还有一点需要注意的是，很容易写出这样的 错误核心代码：

C++

```
1 for (int i = 1; i <= n; i++)
2     for (int l = 0; l <= W - w[i]; l++)
3         f[l + w[i]] = max(f[l] + v[i], f[l + w[i]]);
4 // 由 f[i][l + w[i]] = max(max(f[i - 1][l + w[i]], f[i - 1][l] + w[i]),
5 // f[i][l + w[i]]); 简化而来
```

Python

```
1 for i in range(1, n + 1):
2     for l in range(0, W - w[i] + 1):
3         f[l + w[i]] = max(f[l] + v[i], f[l + w[i]])
4 # 由 f[i][l + w[i]] = max(max(f[i - 1][l + w[i]], f[i - 1][l] + w[i]),
5 # f[i][l + w[i]]); 简化而来
```

这段代码哪里错了呢？枚举顺序错了。

仔细观察代码可以发现：对于当前处理的物品  $i$  和当前状态  $f_{i,j}$ ，在  $j \geq w_i$  时， $f_{i,j}$  是会被  $f_{i,j-w_i}$  所影响的。这就相当于物品  $i$  可以多次被放入背包，与题意不符。（事实上，这正是完全背包问题的解法）

为了避免这种情况发生，我们可以改变枚举的顺序，从  $W$  枚举到  $w_i$ ，这样就不会出现上述的错误，因为  $f_{i,j}$  总是在  $f_{i,j-w_i}$  前被更新。

因此实际核心代码为

C++

```
1 for (int i = 1; i <= n; i++)
2     for (int l = W; l >= w[i]; l--) f[l] = max(f[l], f[l - w[i]] + v[i]);
```

Python

```
1 for i in range(1, n + 1):
2     for l in range(W, w[i] - 1, -1):
3         f[l] = max(f[l], f[l - w[i]] + v[i])
```

## 例题代码

```
1 #include <iostream>
2 using namespace std;
3 constexpr int MAXN = 13010;
4 int n, W, w[MAXN], v[MAXN], f[MAXN];
5
6 int main() {
7     cin >> n >> W;
8     for (int i = 1; i <= n; i++) cin >> w[i] >> v[i]; // 读入数据
9     for (int i = 1; i <= n; i++)
10         for (int l = W; l >= w[i]; l--)
11             if (f[l - w[i]] + v[i] > f[l]) f[l] = f[l - w[i]] + v[i];
12     // 状态方程
13     cout << f[W];
14     return 0;
15 }
```

## 完全背包

### 解释

完全背包模型与 0-1 背包类似，与 0-1 背包的区别仅在于一个物品可以选取无限次，而非仅能选取一次。

我们可以借鉴 0-1 背包的思路，进行状态定义：设  $f_{i,j}$  为只能选前  $i$  个物品时，容量为  $j$  的背包可以达到的最大价值。

需要注意的是，虽然定义与 0-1 背包类似，但是其状态转移方程与 0-1 背包并不相同。

### 过程

可以考虑一个朴素的做法：对于第  $i$  件物品，枚举其选了多少个来转移。这样做的时间复杂度是  $O(n^3)$  的。

状态转移方程如下：

$$f_{i,j} = \max_{k=0}^{+\infty} (f_{i-1,j-k \times w_i} + v_i \times k)$$

考虑做一个简单的优化。可以发现，对于  $f_{i,j}$ ，只要通过  $f_{i,j-w_i}$  转移就可以了。因此状态转移方程为：

$$f_{i,j} = \max(f_{i-1,j}, f_{i,j-w_i} + v_i)$$

理由是当我们这样转移时， $f_{i,j-w_i}$  已经由  $f_{i,j-2 \times w_i}$  更新过，那么  $f_{i,j-w_i}$  就是充分考虑了第  $i$  件物品所选次数后得到的最优结果。换言之，我们通过局部最优子结构的性质重复使用了之前的枚举过程，优化了枚举的复杂度。

与 0-1 背包相同，我们可以将第一维去掉来优化空间复杂度。如果理解了 0-1 背包的优化方式，就不难明白压缩后的循环是正向的（也就是上文中提到的错误优化）。



### 「Luogu P1616」 疯狂的采药



题意概要：有  $n$  种物品和一个容量为  $W$  的背包，每种物品有重量  $w_i$  和价值  $v_i$  两种属性，要求选出若干个物品放入背包使背包中物品的总价值最大且背包中物品的总重量不超过背包的容量。



### 例题代码



```
1 #include <iostream>
2 using namespace std;
3 constexpr int MAXN = 1e4 + 5;
4 constexpr int MAXW = 1e7 + 5;
5 int n, W, w[MAXN], v[MAXN];
6 long long f[MAXW];
7
8 int main() {
9     cin >> W >> n;
10    for (int i = 1; i <= n; i++) cin >> w[i] >> v[i];
11    for (int i = 1; i <= n; i++)
12        for (int l = w[i]; l <= W; l++)
13            if (f[l - w[i]] + v[i] > f[l]) f[l] = f[l - w[i]] + v[i];
14    // 核心状态方程
15    cout << f[W];
16    return 0;
}
```

## 多重背包

多重背包也是 0-1 背包的一个变式。与 0-1 背包的区别在于每种物品有  $k_i$  个，而非一个。

一个很朴素的想法就是：把「每种物品选  $k_i$  次」等价转换为「有  $k_i$  个相同的物品，每个物品选一次」。这样就转换成了一个 0-1 背包模型，套用上文所述的方法就可已解决。状态转移方程如下：

$$f_{i,j} = \max_{k=0}^{k_i} (f_{i-1, j-k \times w_i} + v_i \times k)$$

时间复杂度  $O(W \sum_{i=1}^n k_i)$ 。

## 核心代码

```
1 for (int i = 1; i <= n; i++) {  
2     for (int weight = W; weight >= w[i]; weight--) {  
3         // 多遍历一层物品数量  
4         for (int k = 1; k * w[i] <= weight && k <= cnt[i]; k++) {  
5             dp[weight] = max(dp[weight], dp[weight - k * w[i]] + k *  
6             v[i]);  
7         }  
8     }  
}
```

## 二进制分组优化

考虑优化。我们仍考虑把多重背包转化成 0-1 背包模型来求解。

### 解释

显然，复杂度中的  $O(nW)$  部分无法再优化了，我们只能从  $O(\sum k_i)$  处入手。为了表述方便，我们用  $A_{i,j}$  代表第  $i$  种物品拆分出的第  $j$  个物品。

在朴素的做法中， $\forall j \leq k_i$ ， $A_{i,j}$  均表示相同物品。那么我们效率低的原因主要在于我们进行了大量重复性的工作。举例来说，我们考虑了「同时选  $A_{i,1}, A_{i,2}$ 」与「同时选  $A_{i,2}, A_{i,3}$ 」这两个完全等效的情况。这样的重复性工作我们进行了许多次。那么优化拆分方式就成为了解决问题的突破口。

### 过程

我们可以通过「二进制分组」的方式使拆分方式更加优美。

具体地说就是令  $A_{i,j}$  ( $j \in [0, \lfloor \log_2(k_i + 1) \rfloor - 1]$ ) 分别表示由  $2^j$  个单个物品「捆绑」而成的大物品。特殊地，若  $k_i + 1$  不是 2 的整数次幂，则需要在最后添加一个由  $k_i - 2^{\lfloor \log_2(k_i + 1) \rfloor - 1}$  个单个物品「捆绑」而成的大物品用于补足。

举几个例子：

- $6 = 1 + 2 + 3$
- $8 = 1 + 2 + 4 + 1$
- $18 = 1 + 2 + 4 + 8 + 3$
- $31 = 1 + 2 + 4 + 8 + 16$

显然，通过上述拆分方式，可以表示任意  $\leq k_i$  个物品的等效选择方式。将每种物品按照上述方式拆分后，使用 0-1 背包的方法解决即可。

时间复杂度  $O(W \sum_{i=1}^n \log_2 k_i)$

## 实现

二进制分组代码

C++

```
1 index = 0;
2 for (int i = 1; i <= m; i++) {
3     int c = 1, p, h, k;
4     cin >> p >> h >> k;
5     while (k > c) {
6         k -= c;
7         list[++index].w = c * p;
8         list[index].v = c * h;
9         c *= 2;
10    }
11    list[++index].w = p * k;
12    list[index].v = h * k;
13 }
```

Python

```
1 index = 0
2 for i in range(1, m + 1):
3     c = 1
4     p, h, k = map(int, input().split())
5     while k > c:
6         k -= c
7         index += 1
8         list[index].w = c * p
9         list[index].v = c * h
10        c *= 2
11        index += 1
12        list[index].w = p * k
13        list[index].v = h * k
```

## 单调队列优化

见 [单调队列/单调栈优化](#)。

习题：[「Luogu P1776」宝物筛选\\_NOI 导刊 2010 提高（02）](#)

## 混合背包

混合背包就是将前面三种的背包问题混合起来，有的只能取一次，有的能取无限次，有的只能取  $k$  次。

这种题目看起来很吓人，可是只要领悟了前面几种背包的中心思想，并将其合并在一起就可以了。下面给出伪代码：

```
1 for (循环物品种类) {  
2     if (是 0 - 1 背包)  
3         套用 0 - 1 背包代码;  
4     else if (是完全背包)  
5         套用完全背包代码;  
6     else if (是多重背包)  
7         套用多重背包代码;  
8 }
```

## 例题



### 「Luogu P1833」樱花



有  $n$  种樱花树和长度为  $T$  的时间，有的樱花树只能看一遍，有的樱花树最多看  $A_i$  遍，有的樱花树可以看无数遍。每棵樱花树都有一个美学值  $C_i$ ，求在  $T$  的时间内看哪些樱花树能使美学值最高。



### 核心代码



```
1 for (int i = 1; i <= n; i++) {  
2     if (cnt[i] == 0) { // 如果数量没有限制使用完全背包的核心代码  
3         for (int weight = w[i]; weight <= W; weight++) {  
4             dp[weight] = max(dp[weight], dp[weight - w[i]] + v[i]);  
5         }  
6     } else { // 物品有限使用多重背包的核心代码，它也可以处理0-1背包问题  
7         for (int weight = W; weight >= w[i]; weight--) {  
8             for (int k = 1; k * w[i] <= weight && k <= cnt[i]; k++) {  
9                 dp[weight] = max(dp[weight], dp[weight - k * w[i]] + k *  
10                    v[i]);  
11            }  
12        }  
13    }  
}
```

习题：[HDU 5410 CRB and His Birthday](#)

## 二维费用背包



### 「Luogu P1855」榨取 kkksc03



有  $n$  个任务需要完成，完成第  $i$  个任务需要花费  $t_i$  分钟，产生  $c_i$  元的开支。

现在有  $T$  分钟时间， $W$  元钱来处理这些任务，求最多能完成多少任务。

这道题是很明显的 0-1 背包问题，可是不同的是选一个物品会消耗两种价值（经费、时间），只需在状态中增加一维存放第二种价值即可。

这时候就要注意，再开一维存放物品编号就不合适了，因为容易MLE。

## 实现

### C++

```
1 for (int k = 1; k <= n; k++)
2     for (int i = m; i >= mi; i--)      // 对经费进行一层枚举
3         for (int j = t; j >= ti; j--) // 对时间进行一层枚举
4             dp[i][j] = max(dp[i][j], dp[i - mi][j - ti] + 1);
```

### Python

```
1 for k in range(1, n + 1):
2     for i in range(m, mi - 1, -1): # 对经费进行一层枚举
3         for j in range(t, ti - 1, -1): # 对时间进行一层枚举
4             dp[i][j] = max(dp[i][j], dp[i - mi][j - ti] + 1)
```

## 分组背包



### 「Luogu P1757」通天之分组背包



有  $n$  件物品和一个大小为  $m$  的背包，第  $i$  个物品的价值为  $w_i$ ，体积为  $v_i$ 。同时，每个物品属于一个组，同组内最多只能选择一个物品。求背包能装载物品的最大总价值。

这种题怎么想呢？其实是从「在所有物品中选择一件」变成了「从当前组中选择一件」，于是就对每一组进行一次 0-1 背包就可以了。

再说一说如何进行存储。我们可以将  $t_{k,i}$  表示第  $k$  组的第  $i$  件物品的编号是多少，再用  $cnt_k$  表示第  $k$  组物品有多少个。

## 实现

### C++

```
1  for (int k = 1; k <= ts; k++)           // 循环每一组
2      for (int i = m; i >= 0; i--)         // 循环背包容量
3          for (int j = 1; j <= cnt[k]; j++) // 循环该组的每一个物品
4              if (i >= w[t[k][j]])           // 背包容量充足
5                  dp[i] = max(dp[i],
6                               dp[i - w[t[k][j]]] + c[t[k][j]]); // 像0-1背包一样状态转移
```

移

### Python

```
1  for k in range(1, ts + 1): # 循环每一组
2      for i in range(m, -1, -1): # 循环背包容量
3          for j in range(1, cnt[k] + 1): # 循环该组的每一个物品
4              if i >= w[t[k][j]]: # 背包容量充足
5                  dp[i] = max(
6                      dp[i], dp[i - w[t[k][j]]] + c[t[k][j]])
7          ) # 像0-1背包一样状态转移
```

这里要注意：**一定不能搞错循环顺序**，这样才能保证正确性。

## 有依赖的背包



### 「Luogu P1064」金明的预算方案



金明有  $n$  元钱，想要买  $m$  个物品，第  $i$  件物品的价格为  $v_i$ ，重要度为  $p_i$ 。有些物品是从属于某个主件物品的附件，要买这个物品，必须购买它的主件。

目标是让所有购买的物品的  $v_i \times p_i$  之和最大。

考虑分类讨论。对于一个主件和它的若干附件，有以下几种可能：只买主件，买主件 + 某些附件。因为这几种可能性只能选一种，所以可以将这看成分组背包。

如果是多叉树的集合，则要先算子节点的集合，最后算父节点的集合。

## 泛化物品的背包

这种背包，没有固定的费用和价值，它的价值是随着分配给它的费用而定。在背包容量为  $V$  的背包问题中，当分配给它的费用为  $v_i$  时，能得到的价值就是  $h(v_i)$ 。这时，将固定的价值换成函数的引用即可。

## 杂项

## 小优化

根据贪心原理，当费用相同时，只需保留价值最高的；当价值一定时，只需保留费用最低的；当有两件物品  $i, j$  且  $i$  的价值大于  $j$  的价值并且  $i$  的费用小于  $j$  的费用时，只需保留  $i$ 。

## 背包问题变种

### 输出方案

输出方案其实就是记录下来背包中的某一个状态是怎么推出来的。我们可以用  $g_{i,v}$  表示第  $i$  件物品占用空间为  $v$  的时候是否选择了此物品。然后在转移时记录是选用了哪一种策略（选或不选）。输出时的伪代码：

```
1 int v = V; // 记录当前的存储空间
2
3 // 因为最后一件物品存储的是最终状态，所以从最后一件物品进行循环
4 for (从最后一件循环至第一件) {
5     if (g[i][v]) {
6         选了第 i 项物品;
7         v -= 第 i 项物品的重量;
8     } else {
9         未选第 i 项物品;
10    }
11 }
```

### 求方案数

对于给定的一个背包容量、物品费用、其他关系等的问题，求装到一定容量的方案总数。

这种问题就是把求最大值换成求和即可。

例如 0-1 背包问题的转移方程就变成了：

$$dp_i = \sum (dp_i, dp_{i-c_i})$$

初始条件： $dp_0 = 1$

因为当容量为 0 时也有一个方案，即什么都不装。

### 求最优方案总数

要求最优方案总数，我们要对 0-1 背包里的  $dp$  数组的定义稍作修改，DP 状态  $f_{i,j}$  为在只能放前  $i$  个物品的情况下，容量为  $j$  的背包「正好装满」所能达到的最大总价值。

这样修改之后，每一种 DP 状态都可以用一个  $g_{i,j}$  来表示方案数。

$f_{i,j}$  表示只考虑前  $i$  个物品时背包体积「正好」是  $j$  时的最大价值。

$g_{i,j}$  表示只考虑前  $i$  个物品时背包体积「正好」是  $j$  时的方案数。

转移方程：

如果  $f_{i,j} = f_{i-1,j}$  且  $f_{i,j} \neq f_{i-1,j-v} + w$  说明我们此时不选择把物品放入背包更优，方案数由  $g_{i-1,j}$  转移过来，

如果  $f_{i,j} \neq f_{i-1,j}$  且  $f_{i,j} = f_{i-1,j-v} + w$  说明我们此时选择把物品放入背包更优，方案数由  $g_{i-1,j-v}$  转移过来，

如果  $f_{i,j} = f_{i-1,j}$  且  $f_{i,j} = f_{i-1,j-v} + w$  说明放入或不放入都能取得最优解，方案数由  $g_{i-1,j}$  和  $g_{i-1,j-v}$  转移过来。

初始条件：

```
1 memset(f, 0x3f3f, sizeof(f)); // 避免没有装满而进行了转移
2 f[0] = 0;
3 g[0] = 1; // 什么都不装是一种方案
```

因为背包体积最大值有可能装不满，所以最优解不一定是  $f_m$ 。

最后我们通过找到最优解的价值，把  $g_j$  数组里取到最优解的所有方案数相加即可。

## 实现

```
1 for (int i = 0; i < N; i++) {
2     for (int j = V; j >= v[i]; j--) {
3         int tmp = std::max(dp[j], dp[j - v[i]] + w[i]);
4         int c = 0;
5         if (tmp == dp[j]) c += cnt[j]; // 如果从
6         dp[j] 转移
7         if (tmp == dp[j - v[i]] + w[i]) c += cnt[j - v[i]]; // 如果从
8         dp[j-v[i]] 转移
9         dp[j] = tmp;
10        cnt[j] = c;
11    }
12 }
13 int max = 0; // 寻找最优解
14 for (int i = 0; i <= V; i++) {
15     max = std::max(max, dp[i]);
16 }
17 int res = 0;
18 for (int i = 0; i <= V; i++) {
19     if (dp[i] == max) {
20         res += cnt[i]; // 求和最优解方案数
21     }
22 }
```

背包的第 k 优解

普通的 0-1 背包是要求最优解，在普通的背包 DP 方法上稍作改动，增加一维用于记录当前状态下的前  $k$  优解，即可得到求 0-1 背包第  $k$  优解的算法。具体来讲： $dp_{i,j,k}$  记录了前  $i$  个物品中，选择的物品总体积为  $j$  时，能够得到的第  $k$  大的价值和。这个状态可以理解为将普通 0-1 背包只用记录一个数据的  $dp_{i,j}$  扩展为记录一个有序的优解序列。转移时，普通背包最优解的求法是  $dp_{i,j} = \max(dp_{i-1,j}, dp_{i-1,j-v_i} + w_i)$ ，现在我们则是要合并  $dp_{i-1,j}$ ,  $dp_{i-1,j-v_i} + w_i$  这两个大小为  $1$  的递减序列，并保留合并后前  $k$  大的价值记在  $dp_{i,j}$  里，这一步利用双指针法，复杂度是  $O(k)$  的，整体时间复杂度为  $O(nmk)$ 。空间上，此方法与普通背包一样可以压缩掉第一维，复杂度是  $O(mk)$  的。

### 例题 HDU 2639 Bone Collector II

求 0-1 背包的严格第  $k$  优解。 $n \leq 100, v \leq 1000, k \leq 30$

### 实现

```
1  memset(dp, 0, sizeof(dp));
2  int i, j, p, x, y, z;
3  scanf("%d%d%d", &n, &m, &K);
4  for (i = 0; i < n; i++) scanf("%d", &w[i]);
5  for (i = 0; i < n; i++) scanf("%d", &c[i]);
6  for (i = 0; i < n; i++) {
7      for (j = m; j >= c[i]; j--) {
8          for (p = 1; p <= K; p++) {
9              a[p] = dp[j - c[i]][p] + w[i];
10             b[p] = dp[j][p];
11         }
12         a[p] = b[p] = -1;
13         x = y = z = 1;
14         while (z <= K && (a[x] != -1 || b[y] != -1)) {
15             if (a[x] > b[y])
16                 dp[j][z] = a[x++];
17             else
18                 dp[j][z] = b[y++];
19             if (dp[j][z] != dp[j][z - 1]) z++;
20         }
21     }
22 }
23 printf("%d\n", dp[m][K]);
```

## 参考资料与注释

- 背包问题九讲 - 崔添翼。

-  本页面最近更新：2025/8/9 14:37:03，[更新历史](#)
-  发现错误？想一起完善？[在 GitHub 上编辑此页！](#)
-  本页面贡献者：[lr1d](#), [sshy](#), [StudyingFather](#), [Marcythm](#), [partychicken](#), [H-J-Granger](#), [NachtgeistW](#), [countercurrent-time](#), [Enter-tainer](#), [weiranfu](#), [greyqz](#), [iamtwz](#), [Konano](#), [ksyx](#), [ouuan](#), [paigeman](#), [Tiphereth-A](#), [wolfdan666](#), [AngelKitty](#), [CCXXXI](#), [cjsoft](#), [diauweb](#), [Early0ver](#), [ezoixx130](#), [GekkaSaori](#), [GoodCoder666](#), [Henry-ZHR](#), [HeRaNO](#), [Link-cute](#), [LovelyBuggies](#), [LuoshuiTianyi](#), [Makkiy](#), [mgt](#), [minghu6](#), [odeinjul](#), [oldoldtea](#), [P-Y-Y](#), [PotassiumWings](#), [SamZhangQingChuan](#), [shenshuaijie](#), [Suyun514](#), [weiyong1024](#), [Xeonacid](#), [xyf007](#), [Alisahhh](#), [Alphnia](#), [CBW2007](#), [dhbloo](#), [fps5283](#), [GavinZhengOI](#), [Gesrua](#), [hsfzLZH1](#), [hydingsy](#), [kenlig](#), [kxccc](#), [lychees](#), [Menci](#), [Peanut-Tang](#), [Planarialce](#), [sbofgayschool](#), [shawlleyw](#), [Siyuan](#), [SukkaW](#), [tLLWtG](#), [WAAutoMaton](#), [x4Cx58x54](#), [xk2013](#), [zhb2000](#), [zhufengning](#)
- © 本页面的全部内容在 [CC BY-SA 4.0](#) 和 [SATA](#) 协议之条款下提供，附加条款亦可能应用