南京工業大學
NANJING TECH UNIVERSITY

# ICPC Template Manual

**作者: 贺梦杰**

October 21, 2019

# Contents

# Chapter 1

# 基础

## 1.1 测试

# Chapter 2

# 动态规划

# 2.1 背包 DP

## 2.1.1 0-1 背包洛谷 P1048 采药

### 2.1.1.1 题目描述

有 $n(1 \leq n \leq 100)$ 株药草，每一株有一个采摘时长 $a_i$ 和价值 $b_i$ $(1 \leq a_i, b_i \leq 100)$，给你 $m(1 \leq m \leq 1000)$ 时间，问最大价值是多少?

### 2.1.1.2 代码

```cpp
#include <bits/stdc++.h>

using namespace std;
const int N = 1e5 + 10;

int n, m;            // 物品数量，最大容量
int f[N];            // f[i]表示用i容量能得到的最大价值
int c[110], v[110];  // 每一个物品的消耗和价值

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);

    int i, j;
    cin >> m >> n;
    for (i = 1; i <= n; i++)
        cin >> c[i] >> v[i];

    // 对于每个物品i，恰好使用j容量时，不取或取a[i]
    // 注意j要倒着更新
    for (i = 1; i <= n; i++)
        for (j = m; j >= c[i]; j--)
            f[j] = max(f[j], f[j - c[i]] + v[i]);

    int ans = 0;
    for (i = 1; i <= m; i++)
        ans = max(ans, f[i]);

    cout << ans << endl;

    return 0;
}
```

## 2.1.2 完全背包洛谷 P1616 疯狂的采药

### 2.1.2.1 题目描述

与上一题描述差不多，但此题和原题的不同点：
1. 每种草药可以无限制地疯狂采摘。
2. 药的种类眼花缭乱，采药时间好长好长啊！
另外，物品数 $n(1 \leq n \leq 10000)$，最大容量 $m(1 \leq m \leq 100000)$，物品消耗和价值 $(1 \leq a_i, b_i \leq 10000)$，并且 $n * m \leq 10^7$

### 2.1.2.2 代码 1，直接枚举每种取的数量

大概率 TLE，但还是要放在这，因为有时候可能数据范围不大，但却有别的限制

```cpp
#include <bits/stdc++.h>

using namespace std;
const int N = 1e4 + 10, M = 1e5 + 10;

int n, m;        // 物品数量，最大容量
int f[M];        // f[i]表示用i容量能得到的最大价值
int c[N], v[N];  // 每一个物品的消耗和价值
```

```

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);

    int i, j, k;
    cin >> m >> n;
    for (i = 1; i <= n; i++)
        cin >> c[i] >> v[i];

    // 对于每个物品i，恰好使用j容量时，取k个a[i]
    // 注意j要倒着更新
    for (i = 1; i <= n; i++)
        for (j = m; j >= c[i]; j--)
            for (k = 0; k <= j / c[i]; k++)
                f[j] = max(f[j], f[j - k * c[i]] + k * v[i]);

    int ans = 0;
    for (i = 1; i <= m; i++)
        ans = max(ans, f[i]);

    cout << ans << endl;

    return 0;
}
```

#### 2.1.2.3 代码 2，巧妙地用无限取这个条件

由于可以无限取，所以其实不用关心先后更新

```
// 对于每个物品i，恰好使用j容量时，取k个a[i]
// 注意j的更新顺序，非常妙
for (i = 1; i <= n; i++)
    for (j = c[i]; j <= m; j++)
        f[j] = max(f[j], f[j - c[i]] + v[i]);
}
```

### 2.1.3 分组背包

#### 2.1.3.1 问题描述

有 N 件物品和一个容量为 V 的背包。第 i 件物品的费用是 c[i]，价值是 w[i]。这些物品被划分为若干组，每组中的物品互相冲突，最多选一件。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大。

#### 2.1.3.2 解决方案

状态空间：f[k][v] 表示前 k 组物品花费费用 v 能取得的最大权值
状态转移方程：
f[k][v]=maxf[k-1][v],f[k-1][v-c[i]]+w[i]| 物品 i 属于第 k 组
使用一维数组的伪代码如下：

```
for 所有的组k
    for v=V..0
        for 所有的i属于组k
            f[v]=max{f[v],f[v-c[i]]+w[i]}
```

### 2.1.4 多重背包

#### 2.1.4.1 问题描述

有 N 种物品和一个容量是 V 的背包。
第 i 种物品最多有 si 件，每件体积是 vi，价值是 wi。
求解将哪些物品装入背包，可使物品体积总和不超过背包容量，且价值总和最大。
输出最大价值。

#### 2.1.4.2 解法 1：直接枚举

与完全背包的暴力解法相似，只要加上 k<=s[i] 这个条件

#### 2.1.4.3 解法 2：二进制优化

```cpp
#include <bits/stdc++.h>

using namespace std;
const int N = 16 * 1e3 + 10, M = 2e3 + 10;

int n, m;               // 物品数量，最大容量
int f[M];               // f[i]表示用i容量能得到的最大价值
int s[N], v[N], w[N];   // 每一个物品的个数、消耗和价值

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);

    int i, j, cnt;
    cin >> n >> m;
    for (i = 1; i <= n; i++)
        cin >> v[i] >> w[i] >> s[i];

    // 二进制优化
    cnt = n;
    for (i = 1; i <= n; i++) {
        int t = s[i], d = 1;
        while (t >= d)
            v[++cnt] = v[i] * d, w[cnt] = w[i] * d, t -= d, d <<= 1;
        if (t)
            v[++cnt] = v[i] * t, w[cnt] = w[i] * t;
    }

    // 0-1背包DP
    // 注意i要从n+1开始，因为前n个是未拆分的，如果算上那么每个物品的数量就被错误的乘2了
    for (i = n + 1; i <= cnt; i++)
        for (j = m; j >= v[i]; j--)
            f[j] = max(f[j], f[j - v[i]] + w[i]);

    int ans = 0;
    for (i = 1; i <= m; i++)
        ans = max(ans, f[i]);

    cout << ans << endl;

    return 0;
}
```

#### 2.1.4.4 解法 3：单调队列优化

```cpp
#include <bits/stdc++.h>

using namespace std;
const int N = 1e3 + 10, M = 2e4 + 10;

int n, m;               // 物品数量，最大容量
int f[M];               // f[i]表示用i容量能得到的最大价值
int s[N], v[N], w[N];   // 每一个物品的个数、消耗和价值

struct CycQue {
    const static int MAXSZ = M;
    typedef pair<int, int> Ele;
    Ele q[MAXSZ];
    int head, tail, sz;
```

```
15      CycQue()
16          : head(1), tail(0), sz(0) {}
17      void clear() {
18          head = 1, tail = 0, sz = 0;
19      }
20      bool empty() {
21          return !sz;
22      }
23      void push_back(Ele x) {
24          // 队列满了
25          if (sz >= MAXSZ - 1)
26              sz = sz / (sz - sz);  // 除0，抛出异常
27          tail = (tail + 1) % MAXSZ, q[tail] = x, ++sz;
28      }
29      void push_front(Ele x) {
30          if (sz >= MAXSZ - 1)
31              sz = sz / (sz - sz);
32          head = (head - 1 + MAXSZ) % MAXSZ, q[head] = x, ++sz;
33      }
34      void pop_front() {
35          if (!sz)
36              sz = sz / (sz - sz);
37          head = (head + 1) % MAXSZ, --sz;
38      }
39      void pop_back() {
40          if (!sz)
41              sz = sz / (sz - sz);
42          tail = (tail - 1 + MAXSZ) % MAXSZ, --sz;
43      }
44      Ele front() {
45          if (!sz)
46              sz = sz / (sz - sz);
47          return q[head];
48      }
49      Ele back() {
50          if (!sz)
51              sz = sz / (sz - sz);
52          return q[tail];
53      }
54  } q;
55
56  int main() {
57      ios::sync_with_stdio(0);
58      cin.tie(0);
59
60      int i, j, b;
61      scanf("%d%d", &n, &m);
62      for (i = 1; i <= n; i++)
63          scanf("%d%d%d", &v[i], &w[i], &s[i]);
64
65      // 枚举物品
66      for (i = 1; i <= n; i++) {
67          // 枚举余数
68          for (b = 0; b < v[i]; b++) {
69              q.clear();
70              // 枚举当前取的个数
71              for (j = 0; j <= (m - b) / v[i]; j++) {
72                  int t = f[b + j * v[i]] - j * w[i];
73                  // 入队
74                  while (!q.empty() && q.back().second <= t)
75                      q.pop_back();
76                  q.push_back({j, t});
77                  while (j - q.front().first > s[i])
78                      q.pop_front();
79                  f[b + j * v[i]] = q.front().second + j * w[i];
80              }
```

```
81          }
82      }
83
84      int ans = 0;
85      for (i = 1; i <= m; i++)
86          ans = max(ans, f[i]);
87
88      printf("%d\n", ans);
89
90      return 0;
91  }
```

### 2.1.5 杭电多校 2019 第一场 C HDU6580 Milk

#### 2.1.5.1 题目描述

有一个 n*m 的矩阵，左右可以随便走，但只能在每一行的中点往下走，每走一格花费时间 1. 现在这个矩阵里放了 k 瓶牛奶，第 i 个牛奶喝下去需要 ti 时间起点是（1，1）对于每个 i [1,k], 问喝掉 k 瓶牛奶花费的最小时间

#### 2.1.5.2 解决方案

首先对瓶子的横坐标离散化处理，一行一行地更新答案。每到新的一行，先预处理出向左（右）走喝了 $i$ 瓶水后，回到原点以及不回到原点所需要的最短时间，分别记为 $l, r, l\_back, r\_back$。然后将左右两边合并，求出喝了 $i$ 瓶水后，回到或不回到原点所需时间，记为 $g, g\_back$。设 $f[i]$ 为从 $(1,1)$ 出发，喝了 $i$ 瓶水后回到当前行中点所需的最短时间，则这时就可以根据 $g$ 来和之前的 $f[i]$ 来更新答案，并用 $g\_back$ 来更新 $f$ 的值。时间复杂度为 $O(k^2)$

### 2.1.6 ICPC 上海 2019 网络预选赛 J-Stone game

#### 2.1.6.1 题目描述

有 n 个石头，每个石头都有一个重量为 $a_i$。现要取出一部分，我们设它们为集合 A，则剩余部分为集合 B，要求重量 $A \geq B$，并且当移去 A 中的任意一个石头 $t$ 时，要求 $(A-t) \leq B$。问取法有多少种，最后输出模 $1e9+7$ 其中 $n \leq 300, a_i l \leq 500$，样例数 $T \leq 100$

#### 2.1.6.2 解决方案

容易发现，当划分出 A 之后，只要移除 A 中**最轻的石头 t** 是满足题意的这种方案就是可行的。
于是我们可以先将**数组排序，然后枚举 t**，对每一个 t 分别考虑。最暴力的想法是考虑 t 后面的每一个 $a_i$ 取或不取，但是这样复杂度上天。
再仔细想想，就会发现所有 $a_i$ 加起来也就 $1.5e5$，也就是可以用**背包 dp**，$f(i,j)$ 表示 $t=i$ 时总重量为 j 的方案数。复杂度是？不考虑 T，枚举 t 为 $O(N)$，对于每个 t，考虑其后的每个数取或不取为 $O(N*1.5e5)$，总复杂度为 $O(N^2*1.5*10^5)$，又上天了。
但是当我们**倒过来想**，**将数组从大到小排序**，然后从前向后枚举 t，就会发现复杂度降到了 $O(N*1.5*10^5)$。每次 $++t$，都相当于整体加上 $a_t$（$a_t$ 必取），然后再考虑不取 $a_{t-1}$ 的情况（$a_{t-1}$ 在上一个 t 中是必选的）。

## 2.2 位置 DP

这类 dp 一般都会以数字出现位置作为一个维度。

### 2.2.1 杭电多校 2019 第一场 A HDU6578 Blank

#### 2.2.1.1 题目描述

有 n ( 100) 个格子，向其中填入 0、1、2、3 这 4 个数，但是有 m ( 100) 个限制
限制 l r x： 表示 l r 的格子内不同的数的个数为 x
问一共有多少种填入方案?

#### 2.2.1.2 解决方案

**初步分析：**构建 dp[i][j][k][t][cur]，表示到 cur 位置为止，0,1,2,3 四个数最后出现在位置 i,j,k,t 的情况数。
dp[cur][j][k][t][cur]=dp[cur][j][k][t][cur]+dp[i][j][k][t][cur−1]
dp[i][cur][k][t][cur]=dp[i][cur][k][t][cur]+dp[i][j][k][t][cur−1]
dp[i][j][cur][t][cur]=dp[i][j][cur][t][cur]+dp[i][j][k][t][cur−1]
dp[i][j][k][cur][cur]=dp[i][j][k][cur][cur]+dp[i][j][k][t][cur−1]
**(注意以上求法是对每个待求 dp 分别求，等号后面第二项需要迭代)**
这样当 cur==r，只要依据四个数出现位置是否大于等于 l 就可以判断有几个不同的数，若不满足限制则令 dp 等于 0。

**优化 1:** 我们发现上面状态中一定会有一个 cur，即有一维度是不会变的，但是这一维度的位置一直在变。
那么就可以构建 dp[i][j][k][cur] ，其中 i < j < k < cur （若为 0 可等于）：只知道 i, j, k, cur 是不同的数最后出现的位置，且 i < j < k < cur （并不需要知道 i,j,k,cur 对应的填入数是哪个，因为这并不影响对限制的判断）
dp[j][k][cur−1][cur]=dp[j][k][cur−1][cur]+dp[i][j][k][cur−1]
dp[i][k][cur−1][cur]=dp[i][k][cur−1][cur]+dp[i][j][k][cur−1]
dp[i][j][cur−1][cur]=dp[i][j][cur−1][cur]+dp[i][j][k][cur−1]
dp[i][j][k][cur]=dp[i][j][k][cur]+dp[i][j][k][cur−1]
**(注意以上求法是对每一个已知 dp 去累加至待求 dp，因此不需要迭代**
分别表示将最后一次出现位置为 i, j, k ,cur-1 的数填入第 cur 个格子中

**优化 2:** 我们发现上面状态中最后一个维度要么为 cur 要么为 cur-1，即当前状态仅与上一个状态有关，所以可以用滚动数组优化空间。
构建 dp[i][j][k][2]，得到状态转移方程：
dp[j][k][cur−1][now]=dp[j][k][cur−1][now]+dp[i][j][k][pre]
dp[i][k][cur−1][now]=dp[i][k][cur−1][now]+dp[i][j][k][pre]
dp[i][j][cur−1][now]=dp[i][j][cur−1][now]+dp[i][j][k][pre]
dp[i][j][k][now]=dp[i][j][k][now]+dp[i][j][k][pre]
注意：因为这里是累加，所以用滚动数组时，dp[i][j][k][pre] 用完以后要清空!

### 2.2.2 杭电多校 2019 第一场 J HDU6587 Kingdom

#### 2.2.2.1 题目描述

给出 n(<=100) 个点的树的前序遍历和中序遍历，这两个序列中某些编号未知 (用 0 来表示)。问有多少种合法的树。

#### 2.2.2.2 解决方案

位置 DP，二叉树前序和中序遍历的关系

最关键的不是 dp 的状态和方程，而是限制，**要求左子树和右子树的节点必须在对应的区间内（即不交叉，不出界）**
**记录 a 中节点在 b 中出现的位置，b 中节点在 a 中出现的位置。**
**在每进入一个新的区间时，都检查一遍，是否 a 的区间中所有节点都在 b 的对应区间中，同理反过来再做一次**

## 2.3 树形动态规划

### 2.3.1 加分二叉树

#### 2.3.1.1 问题描述

设一个 n 个节点的二叉树 tree 的中序遍历为（l,2,3,…,n），其中数字 1,2,3,…,n 为节点编号。每个节点都有一个分数（均为正整数），记第 i 个节点的分数为 di，tree 及它的每个子树都有一个加分，任一棵子树 subtree（也包含 tree 本身）的加分计算方法如下：
subtree 的左子树的加分 × subtree 的右子树的加分+ subtree 的根的分数
若某个子树为空，规定其加分为 1，叶子的加分就是叶节点本身的分数。不考虑它的空子树。
试求一棵符合中序遍历为（1,2,3,…,n）且加分最高的二叉树 tree。要求输出；
（1）tree 的最高加分
（2）tree 的前序遍历

#### 2.3.1.2 输入格式

第 1 行：一个整数 n（n < 30），为节点个数。
第 2 行：n 个用空格隔开的整数，为每个节点的分数（分数< 100）。

#### 2.3.1.3 输出格式

第 1 行：一个整数，为最高加分（结果不会超过 4,000,000,000）。
第 2 行：n 个用空格隔开的整数，为该树的前序遍历。

#### 2.3.1.4 思路

这道题看上去是树形 DP，但仔细思考，我们发现很难依据中序遍历建出树。但中序遍历有其独特之处，即一旦根被确定，则左右子树也被确定。
所以我们应该用区间 DP 来解决这道题。
$f(i,j)$: 选 i 到 j 的节点作为一颗子树最大的得分

$$f(i,j) = \max_{i \le k \le j} \{f(i,k-1) * f(k+1,j) + f(k,k)\}$$

由题意，当 $i > j$，$f(i,j) = 1$ 为空节点。
前序遍历就是重新用 dfs 走一遍：每次找到使 $f(i,j)$ 最大的 $k$，然后以 $k$ 为分界向左右两边递归。

### 2.3.2 洛谷 P2015 二叉苹果树

#### 2.3.2.1 问题描述

有一棵苹果树，如果树枝有分叉，一定是分 2 叉（就是说没有只有 1 个儿子的结点）这棵树共有 N 个结点（叶子点或者树枝分叉点），编号为 1-N, 树根编号一定是 1。我们用一根树枝两端连接的结点的编号来描述一根树枝的位置。现在这颗树枝条太多了，需要剪枝。但是一些树枝上长有苹果。
给定需要保留的树枝数量，求出最多能留住多少苹果。

#### 2.3.2.2 输入格式

第 1 行 2 个数，N 和 Q(1<=Q<= N,1<N<=100)。
N 表示树的结点数，Q 表示要保留的树枝数量。接下来 N-1 行描述树枝的信息。
每行 3 个整数，前两个是它连接的结点的编号。第 3 个数是这根树枝上苹果的数量。
每根树枝上的苹果不超过 30000 个。

#### 2.3.2.3 输出格式

剩余苹果的最大数量。

#### 2.3.2.4 思路

$f(i,j)$ 表示以 i 为节点的根保留 k 条边的最大值接下来对每一个节点分类讨论，共三种情况：
1. 全选左子树
2. 全选右子树
3. 左右子树都有一部分

为了方便，我们设左儿子为 ls，右儿子为 rs，连接左儿子的边为 le，连接右儿子的边为 re

$$f(i,j)=\max\{f(ls,j-1)+le,f(rs,j-1)+re,\max_{0\le k\le j-2}\{f(ls,k)+f(rs,j-2-k)\}+le+re\}$$

### 2.3.3 最大利润

#### 2.3.3.1 问题描述

政府邀请了你在火车站开饭店，但不允许同时在两个相连接的火车站开。任意两个火车站有且只有一条路径，每个火车站最多有 50 个和它相连接的火车站。
告诉你每个火车站的利润，问你可以获得的最大利润为多少。
最佳投资方案是在 1，2，5，6 这 4 个火车站开饭店可以获得利润为 90

#### 2.3.3.2 输入格式

第一行输入整数 N(<=100000)，表示有 N 个火车站，分别用 1，2。。。，N 来编号。接下来 N 行，每行一个整数表示每个站点的利润，接下来 N-1 行描述火车站网络，每行两个整数，表示相连接的两个站点。

#### 2.3.3.3 输出格式

输出一个整数表示可以获得的最大利润。

#### 2.3.3.4 思路

这道题虽然是多叉树，但状态仍然是比较简单的。
对于某个结点，如果选择该节点，则该节点的所有儿子都不能选，如果不选该节点，则它的儿子可选可不选。
所以，我们令 $f(i)$ 表示以 i 节点为根的子树中选 i 的最大利润，$h(i)$ 表示以 i 节点为根的子树中不选 i 的最大利润。a[i] 是 i 本身的利润，j 是 i 的儿子

$$f(i)=a[i]+\sum_j h(j)$$

$$h(i)=\sum_j \max\{f(j),h(j)\}$$

### 2.3.4 洛谷 P2014 选课

#### 2.3.4.1 问题描述

学校实行学分制。每门的必修课都有固定的学分，同时还必须获得相应的选修课程学分。学校开设了 N（N<300）门的选修课程，每个学生可选课程的数量 M 是给定的。学生选修了这 M 门课并考核通过就能获得相应的学分。
在选修课程中，有些课程可以直接选修，有些课程需要一定的基础知识，必须在选了其它的一些课程的基础上才能选修。例如《Frontpage》必须在选修了《Windows 操作基础》之后才能选修。我们称《Windows 操作基础》是《Frontpage》的先修课。每门课的直接先修课最多只有一门。两门课也可能存在相同的先修课。每门课都有一个课号，依次为 1，2，3，…。
你的任务是为自己确定一个选课方案，使得你能得到的学分最多，并且必须满足先修课优先的原则。假定课程之间不存在时间上的冲突。

#### 2.3.4.2 输入格式

输入文件的第一行包括两个整数 N、M（中间用一个空格隔开），其中 $1\le N\le 300, 1\le M\le N$。以下 N 行每行代表一门课。课号依次为 1，2，…，N。每行有两个数（用一个空格隔开），第一个数为这门课先修课的课号（若不存在先修课则该项为 0），第二个数为这门课的学分。学分是不超过 10 的正整数。

#### 2.3.4.3 输出格式

只有一个数：实际所选课程的学分总数。

#### 2.3.4.4 思路一

分析一下，似乎也不是很难，对于某个节点，只有选择它，才能选择它的儿子们。
令 $f(x,i)$ 表示在以 x 结点为根的子树中选择 i 个点的最大学分。$f(x,1)=s[x]$，s[x] 是课程 x 本身的学分。
假设结点 x 有 k 个儿子，标号为 $y_1,y_2,..,y_k$，让他们分别选 $i_1,i_2,..,i_k$ 门课程，那么状态转移方程似乎是..

$$f(x,i)=s[x]+\max_{i_1+i_2+..+i_k=i-1}\sum_{1\le j\le k}f(y_j,i_j)$$

好像.. 写不出这样的循环啊，当然，如果用 dfs 强行写也不是不可以，但是时间复杂度必炸。

这时候我们要用一种类似**前缀和**的思维
即，我们每 dfs 完一棵子树，都进行一次完整的 dp 更新。假设当前根结点为 x，我们刚 dfs 完它的一棵子树 y，那么我们进行如下更新（对所有 i）：

$$f(x,i)=\max_{1\le j\le i}\{f(x,j)+f(y,i-j)\}$$

**此时 $f(x,i)$ 表示在以 x 结点为根的子树中已经被 dfs 过的部分中选 i 个点的最大学分，而每次更新就像是把一棵新子树添加到答案中。**
除此以外，还有一点要注意，就是我们要让 **i 递减更新**，因为大的 i 要用到之前小的 i，而小的 i 不要用到大的 i

#### 2.3.4.5 思路二

先将**森林转二叉树（左孩子右兄弟）**。如何转呢？设 D[x]、c[x] 和 b[x] 分别表示 x 结点的父亲、左孩子和右兄弟，对于每个节点，D[x] 是已知的，所以 **b[x]=c[D[x]],c[D[x]]=x**。
再分析，对于某个节点，如果要选其左孩子，则必须选它，而选右孩子则没有限制。
令 $f(x,i)$ 表示在以 x 结点为根的子树中选择 i 个点的最大学分。$f(x,1)=s[x]$，s[x] 是课程 x 本身的学分。
还是沿用上面**前缀和思想**，先由左孩子 (ls) 更新，再由右孩子 (rs) **倒着**更新，两次的方程如下：

$$f(x,i)=f(x,1)+f(ls,i-1)$$

$$f(x,i)=\max_{0\le j\le i}f(x,j)+f(rs,i-j)$$

### 2.3.5 HYSBZ 2427 软件安装

#### 2.3.5.1 题目描述

现在我们的手头有 N 个软件，对于一个软件 i，它要占用 Wi 的磁盘空间，它的价值为 Vi。我们希望从中选择一些软件安装到一台磁盘容量为 M 的计算机上，使得这些软件的价值尽可能大（即 Vi 的和最大）。
但是现在有个问题：软件之间存在依赖关系，即软件 i 只有在安装了软件 j（包括软件 j 的直接或间接依赖）的情况下才能正确工作（软件 i 依赖软件 j）。幸运的是，一个软件最多依赖另外一个软件。如果一个软件不能正常工作，那么他能够发挥的作用为 0。
我们现在知道了软件之间的依赖关系：软件 i 依赖 Di。现在请你设计出一种方案，安装价值尽量大的软件。一个软件只能被安装一次，如果一个软件没有依赖则 Di=0，这是只要这个软件安装了，它就能正常工作。
subsubsection 输入格式 第 1 行：N,M（0<=N<=100,0<=M<=500）第 2 行：W1,W2, …Wi, …,Wn 第 3 行：V1,V2, …Vi, …,Vn 第 4 行：D1,D2, …Di, …,Dn

#### 2.3.5.2 输出格式

一个整数，代表最大价值。

#### 2.3.5.3 思路

**注意，此图可能有环**
仔细读题，题中说一个软件只依赖最多一个软件，所以在一个联通分之内最多只有一个环，并且该环可以指向环以外的节点，而环以外的节点不会指向环。
因此，我们可以**把环当做一个新节点**来处理，这个新节点所占空间和价值都为环内元素加和。
怎么确定环及环内元素呢？我推荐**着色法（有向图为黑白灰，无向图为黑白）**。
有向图中白色为未探索的点，灰色为正在探索的点，黑色为已经探索过的并且确定不成环的点。若在探索过程中遇到灰点，则说明成环。
无向图中白色为未探索的点，黑色为已经探索过和正在探索的点。若在探索过程中遇到黑点，则说明成环。
处理完成后，DP 思路同”洛谷 *P2014* 选课”

## 2.3.6 CF486D Valid Sets

### 2.3.6.1 题目描述

给定一棵树，每个点都有一个权，现在要你选择一个连通块，问有多少种选择方法，使得连通块中的最大点权和最小点权的差值小于等于 d。答案对 1e9+7 取模。

### 2.3.6.2 输入格式

The first line contains two space-separated integers d (0 d 2000) and n (1 n 2000).
The second line contains n space-separated positive integers a1, a2, ..., an(1 ai 2000).
Then the next n - 1 line each contain pair of integers u and v (1 u, v n) denoting that there is an edge between u and v. It is guaranteed that these edges form a tree.

### 2.3.6.3 输出格式

Print the number of valid sets modulo 1000000007.

### 2.3.6.4 思路

我起初的错误想法是，设 $f(x,i)$ 和 $g(x,i)$ 分别为以 x 为根的子树中最大值和最小值为 i 的方案数，企图通过对 g 和 f 的运算得出答案，但这是错误的，主要原因是 f 和 g 无法把范围限定住。
受到前面以子树为对象思考的影响，形成了**思维惯性**，这里我们并不是以子树为对象，在一遍 dfs 里算出所有答案。
而是**枚举**，枚举每一点 x，将 x 作为连通块的最大值，以 x 为根进行 dfs，求方案数。
另外要注意：因为点权可能相同，所以为了避免重复计算，我们需要定序，若权值相同，让编号大的点访问编号小的，而编号小的不能访问大的。

## 2.3.7 CF294E Shaass the Great

### 2.3.7.1 题目描述

给一颗带边权的树，让你选一条边删除，然后在得到的两个子树中各选一个点，用原来被删除的边连起来，重新拼成一棵树。使得这棵树的所有点对的距离总和最小。

### 2.3.7.2 输入格式

The first line of the input contains an integer n denoting the number of cities in the empire, ($2 \leq n \leq 5000$). The next n - 1 lines each contains three integers ai, bi and wi showing that two cities ai and bi are connected using a road of length wi, ($1 \leq a_i, b_i \leq n, a_i \neq b_i, 1 \leq w_i \leq 106$).

### 2.3.7.3 输出格式

所有点对距离总和最优解。

### 2.3.7.4 思路

设我们要删除的边为 e，以 e 为分界树被分割成了左右两个部分 (我们称为左树和右树)，最后我们连接的两个点为 vl 和 vr
则答案为：
左树内点对距离总和 + 右树内点对距离总和
+ 左树中所有点到 vl 的距离总和 * 右树中点的个数 + 右树中所有点到 vr 的距离总和 * 左树中点的个数
+ e 的权重 * 左树中点的个数 * 右树中点的个数

再观察一下，其实在删除 e 的情况下左右树内点对距离总和、左右树中的点的个数、e 的权重都是不变的，变化的只有左右树中所有点到 vl 和 vr 的距离总和
由此我们知道了，在删除 e 的情况下，答案最优的条件即为**找到 vl 和 vr，使得左树中所有点到 vl 的距离总和最小，右树中所有点到 vr 的距离总和最小**
于是我们知道了，要得到答案，就是要求一棵树的三个值：树内点的个数、树上所有点到某一点的距离总和的最小值、树内所有点对距离总和
树内点的个数最简单，不多说了，而树内所有点对的距离和也可以由树上所有点到某一点的距离和算得 (即树上所有点到**每**一点的和除以 2)

下面主要考虑如何在 $O(n)$ 的时间内求出树上所有点到每一点的距离：

**考虑 dfs，结果发现一遍 dfs 无论如何都不可能得到，但是可以知道以某一点 x 为根的子树上所有点到根 x 的距离总和**
**在第一遍 dfs 的基础上，我们再进行一次 dfs，这次 dfs 我们不再是自底向上更新，而是自顶向下更新，更新父节点及其以上所有的点到 x 的距离总和**
**可以理解为，第一次 dfs 我们得到了 x 以下的所有点到 x 的距离总和，第二次 dfs 我们得到了 x 以上的所有点到 x 的距离总和**

对于第二次 dfs，例如我们现在在 a 节点上，即将去往 x 结点，我们需要下传给 x 节点的距离有哪些呢?
1.a 结点上面传下来的距离
2.a 结点上除了 x 分支所有旁路上的距离 (由第一次 dfs 可以算出)
3.a 与 x 的边权

所以最后的算法是：**枚举每一条边**，运用以上算法，求出最小值

## 2.4 最长上升子序列

### 2.4.1 基本实现

```cpp
#include <bits/stdc++.h>

using namespace std;
const int N = 1e3 + 10;

int n, a[N];
int b[N], c[N];  // b[i]是以a[i]为右端点的最长上升子序列的长度，c[i]是长度为i的最长上升子序列的右端点的最小值

inline void solve() {
    int i, p, len = 0;
    for (i = 1; i <= n; i++) {
        p = lower_bound(c + 1, c + 1 + len, a[i]) - c;
        b[i] = p, c[p] = a[i], len = max(len, p);
    }
}

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);

    int i;
    cin >> n;
    for (i = 1; i <= n; i++)
        cin >> a[i];
    solve();
    int ans = 0;
    for (i = 1; i <= n; i++)
        ans = max(ans, b[i]);
    cout << ans << endl;

    return 0;
}
```

### 2.4.2 另一种解法：权值线段树

对于 a[i]，我们可以通过查找以比 a[i] 小的数为结尾的最长长度，然后 +1 即可得到以 a[i] 为结尾的最长长度。
权值线段树维护以某权值为结尾的最大长度。注意需要离散化。
由于是查前缀，所以权值线段树可以换用树状数组

```cpp
#include <bits/stdc++.h>

using namespace std;
const int N = 1e5 + 10;

int n, a[N];
int b[N];    // b[i]是以a[i]为右端点的最长上升子序列的长度
int i2x[N];  // 离散化
int m;       // 离散化的长度
int t[N];    // 树状数组

inline int x2i(int x) {
    return lower_bound(i2x + 1, i2x + 1 + m, x) - i2x;
}

inline int lowbit(int x) {
    return -x & x;
}

inline void upd(int p, int v) {
    for (; p <= m; p += lowbit(p))
```

```
        t[p] = max(t[p], v);
}

inline int qry(int p) {
    int ret = 0;
    for (; p > 0; p -= lowbit(p))
        ret = max(ret, t[p]);
    return ret;
}

inline void solve() {
    int i;
    for (i = 1; i <= n; i++) {
        b[i] = qry(x2i(a[i]) - 1) + 1;
        upd(x2i(a[i]), b[i]);
    }
}

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);

    int i;
    cin >> n;
    for (i = 1; i <= n; i++)
        cin >> a[i], i2x[i] = a[i];
    sort(i2x + 1, i2x + 1 + n);
    m = unique(i2x + 1, i2x + 1 + n) - i2x - 1;

    solve();

    int ans = 0;
    for (i = 1; i <= n; i++)
        ans = max(ans, b[i]);
    cout << ans << endl;

    return 0;
}
```

### 2.4.3 输出最小字典序的下标

原问题是记录了以 a[i] 为结尾的最长上升子序列的长度，而此问题是记录以 a[i] 为**开头**的最长上升子序列的长度

PS：若要求输出最大字典序的下标，只要还改成记录以 a[i] 为**结尾**的最长上升子序列的长度，然后倒过来筛就可以了

```
#include <bits/stdc++.h>

using namespace std;
const int N = 1e5 + 10;

int n, a[N];
int d[N], c[N];  // d[i]是以a[i]为左端点的最长上升子序列的长度，c[i]用于记录从右向左长度为i的最小左端点

// 此处定义的是小于号，注意，不可以有等于。lower_bound配上<=就是upper_bound。
// 所以无论是lower还是upper都不带等号
bool cmp(const int& lhs, const int& rhs) {
    return lhs > rhs;
}

inline void solve() {
    int i, len, mx = 0;
    for (i = n; i >= 1; i--) {
        len = lower_bound(c + 1, c + 1 + mx, a[i], cmp) - c;
        d[i] = len, c[len] = a[i], mx = max(mx, len);
    }
```

```
}

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);

    int i;
    cin >> n;
    for (i = 1; i <= n; i++)
        cin >> a[i];

    solve();

    int mx = 0;
    for (i = 1; i <= n; i++)
        mx = max(mx, d[i]);

    cout << mx << endl;

    int cur = mx, last = 0;
    for (i = 1; i <= n; i++) {
        if (d[i] == cur && a[i] > last) {
            if (cur != mx)
                cout << " ";
            cout << i;  // 输出下标
            --cur, last = a[i];
        }
    }
    cout << endl;

    return 0;
}
```

### 2.4.4 输出值的最小字典序

与上面问题不同的是，这里要求的是值的最小字典序。
例如：6 7 8 9 1 2 3 4，我们要输出的是 1 2 3 4。
对于这个问题，我们只要记录每个数的前驱即可。

```
#include <bits/stdc++.h>

using namespace std;
const int N = 1e5 + 10;

int n, a[N];
int b[N], c[N];      // b[i]是以a[i]为右端点的最长上升子序列的长度，c[i]用于记录长度为i的最小端点
int pos[N], pre[N];  // pos[i]是与c[i]对应的下标，pre[i]是a[i]的前驱的下标
int ans[N];
// a[] b[] pre[] 下标一致；c[] pos[] 下标一致

inline void solve() {
    int i, len, mx = 0;
    for (i = 1; i <= n; i++) {
        len = lower_bound(c + 1, c + 1 + mx, a[i]) - c;
        b[i] = len, pre[i] = pos[len - 1];
        c[len] = a[i], pos[len] = i;
        mx = max(mx, len);
    }
}

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);

    int i, T;
    cin >> T;
```

```
28      while (T--) {
29          cin >> n;
30          for (i = 1; i <= n; i++)
31              cin >> a[i];
32
33          solve();
34
35          int mx = 0;
36          for (i = 1; i <= n; i++)
37              mx = max(mx, b[i]);
38
39          cout << mx << endl;
40
41          int p = pos[mx], len = mx;
42          while (p) {
43              ans[len--] = a[p];
44              p = pre[p];
45          }
46
47          for (i = 1; i <= mx; i++) {
48              if (i > 1)
49                  cout << " ";
50              cout << ans[i];
51          }
52          cout << endl;
53      }
54
55      return 0;
56  }
```

### 2.4.5 最长不降子序列

```
1  p = upper_bound(c + 1, c + 1 + len, a[i]) - c;
```

## 2.5 最长公共子序列

给定 S1 和 S2 串，要求一个序列，同时是 S1 和 S2 的子序列。

### 2.5.1 CCPC 2019 秦皇岛 C.Sakura Reset

#### 2.5.1.1 题目描述

给定 A、B 串及其长度 n、m($1 \le n, m \le 5000$)，要求 A 的子序列 a 和 B 的子序列 b，使得 a 的长度大于 b 的长度或 a、b 长度相等且字符串 a 大于字符串 b。

#### 2.5.1.2 解决方案

整体上，要分类讨论：1.a 的长度大于 b 的长度；2. 长度相等且字符串 a 大于字符串 b

对于分类 1：
即要分别考虑 A 和 B 的各个长度的本质不同的子序列个数。答案就是枚举 A 的每个长度，其个数乘上 B 的小于该长度的自序列个数。
如何统计呢？首先要理解贪心匹配，即如何判断 t 是否是 s 的子序列？根据贪心匹配，对于元素 $t_1$ 我们就是要找它在 s 中出现的**第一个**位置 $p_1$（$t_1 = s_{p_1}$），接下来在 s 的 $p_1$ 后面找 $t_2$ 并令其为 $p_2$，以此类推。
根据上面的规则，对于一个串 X，我们令 $f(i)$ 表示以 $X[i]$ 为结尾的**与之前的不重复**的且**本质不同**的子序列个数。则转移方程

$$f(i) = \sum_{j=last[i]}^{i-1} f(j)$$

其中 $last[i]$ 表示 $X[i]$ 上一次出现的位置，若第一次出现 $last[i] = 0$。
为什么是从 $last[i]$ 开始累加呢？因为以下标小于 $last[i]$ 的元素为结尾的子序列后面再加上当前的 $X[i]$ 得出的新的子序列，和之前加上 $last[i]$ 处的 $X[last[i]]$ 组成的子序列重复了。通过**一维前缀和**可以优化至 $O(n)$。回到现在的问题，我们要求每个长度上的本质不同的子序列个数，只要加上一维长度维度即可 ($f(i, j)$)。时间 $O(n^2)$。

对于分类 2：
要求长度相等，且按字符串的方式比较 a 大于 b。即要求 a 和 b 有一段长度大于等于 0 的相同前缀，在第一个不相等的位置 a 的元素大于 b 的元素，后面只要有长度相等的后缀即可。
令 $g(i, j)$ 表示以 $A[i]$ 结尾的 a 和以 $B[j]$ 结尾的 b **与之前的不重复**的且**本质不同**的公共子序列对数。
若 $A[i] \neq b[i]$，显然 $g(i, j) = 0$；若 $A[i] = B[j]$，

$$g(i, j) = \sum_{u=last[i]}^{i-1} \sum_{v=last[j]}^{j-1} g(u, v)$$

再考虑后缀，令 $h(i, j)$ 为从后向前以 $A[i]$ 结尾的 a 和以 $B[j]$ 结尾的 b **与之前的不重复**的且**本质不同**的长度相等子序列对数。后缀的求法有两种，可以和求 $f$ 向类似，但是在算答案的时候回非常复杂，也可以和求 $g$ 类似，只不过是倒着求，并且不要求 $A[i] = B[j]$。最后枚举 $i$、$j$，若 $A[i] > B[j]$，则

$$ans+ = (\sum_{u=1}^{i-1} \sum_{v=1}^{j-1} g(u, v)) * h(i, j)$$

以上都可以通过**二维前缀和或二维后缀和**优化至 $O(N^2)$。

**注意：上面两个前缀和的意义是不一样的。$f$ 的前缀和是一维的，它只对下标 i 求前缀和，不对长度 j 求；而下面 $g$ 的前缀和以及 $h$ 的后缀和是二维的，即 i,j 都要求。所以还要注意两种在边界上的初始化是不一样的。**

#### 2.5.1.3 代码

```
#include <bits/stdc++.h>

using namespace std;
typedef long long ll;
const ll N = 5e3 + 10, P = 998244353;

ll len[2];          // 两串的长度
ll S[2][N];         // A串和B串
```

```
9  ll f[2][N][N];      // A串和B串的dp, f[0][i][j]:以A[i]为结尾且长度为j的本质不同的子序列个数
10 ll fpre[2][N][N];  // 对f的**i**求前缀和, 并不是二维前缀和
11 ll last[2][N];     // A串和B串, A[i]或B[i]上一次出现的位置
12 ll pos[2][110];    // pos[i], 值为i的元素上一次出现的位置
13 ll g[N][N];        // 二维矩阵, g[i][j]表示以A[i]结尾的子序列和以B[i]结尾的子序列为公共子序列的方案数
14 ll gpre[N][N];     // g的二维前缀和
15 ll h[N][N];        // 二维矩阵, h[i][j]表示 从后向前 以A[i]结尾的子序列和以B[i]结尾的子序列长度相等但本
       质不同的后缀对个数
16 ll hsuf[N][N];     // h的二维后缀和
17
18 inline bool isOk(ll r, ll c) {
19     return r >= 0 && c >= 0;
20 }
21
22 inline ll presum(ll r1, ll c1, ll r2, ll c2) {
23     ll v1 = 0, v2 = 0, v3 = 0, v4 = 0;
24     if (isOk(r2, c2))
25         v1 = gpre[r2][c2];
26     if (isOk(r1 - 1, c2))
27         v2 = gpre[r1 - 1][c2];
28     if (isOk(r2, c1 - 1))
29         v3 = gpre[r2][c1 - 1];
30     if (isOk(r1 - 1, c1 - 1))
31         v4 = gpre[r1 - 1][c1 - 1];
32     return v1 - v2 - v3 + v4;
33 }
34
35 inline ll sufsum(ll r1, ll c1, ll r2, ll c2) {
36     return hsuf[r2][c2] - hsuf[r1 + 1][c2] - hsuf[r2][c1 + 1] + hsuf[r1 + 1][c1 + 1];
37 }
38
39 int main() {
40     ios::sync_with_stdio(0);
41     cin.tie(0);
42
43     ll i, j, k, ans = 0;
44
45     cin >> len[0] >> len[1];
46
47     for (k = 0; k <= 1; k++)
48         for (i = 1; i <= len[k]; i++)
49             cin >> S[k][i];
50
51     // 预处理last
52     for (k = 0; k <= 1; k++)
53         for (i = 1; i <= len[k]; i++)
54             last[k][i] = pos[k][S[k][i]],
55             pos[k][S[k][i]] = i;
56
57     // 对于长度不同的情况-----------------------------------------
58     // 首先求f和fpre
59     f[0][0][0] = f[1][0][0] = fpre[0][0][0] = fpre[1][0][0] = 1;
60     for (k = 0; k <= 1; k++) {
61         // 初始化边界
62         for (i = 1; i <= len[k]; i++)
63             fpre[k][i][0] = 1;
64         for (i = 1; i <= len[k]; i++)
65             for (j = 1; j <= i; j++)
66                 f[k][i][j] = (fpre[k][i - 1][j - 1] - fpre[k][last[k][i]][j - 1] + f[k][last[k][i
       ]][j - 1]) % P,
67                 fpre[k][i][j] = (fpre[k][i - 1][j] + f[k][i][j]) % P;
68     }
69     // 计算长度不同的答案
70     ll t = 0;
71     for (i = 1; i <= len[0]; i++)
72         ans = (ans + fpre[0][len[0]][i] * t % P) % P,
```

```
73          t += fpre[1][len[1]][i];
74
75      // 对于长度相同的情况-----------------------------------------
76      // 首先求g和gpre
77      g[0][0] = gpre[0][0] = 1;
78      // 初始化边界
79      for (i = 1; i <= len[0]; i++)
80          gpre[i][0] = 1;
81      for (i = 1; i <= len[1]; i++)
82          gpre[0][i] = 1;
83      for (i = 1; i <= len[0]; i++)
84          for (j = 1; j <= len[1]; j++)
85              g[i][j] = (S[0][i] == S[1][j] ? presum(last[0][i], last[1][j], i - 1, j - 1) : 0),
86              gpre[i][j] = gpre[i - 1][j] + gpre[i][j - 1] - gpre[i - 1][j - 1] + g[i][j];
87      // 再求从后向前以i为结尾的长度为j的本质不同的子序列个数，求法类似之前求的f，只是倒过来了
88      // 在此之前，先预处理last
89      for (k = 0; k <= 1; k++)
90          for (i = 1; i <= 100; i++)
91              pos[k][i] = len[k] + 1;
92      for (k = 0; k <= 1; k++)
93          for (i = len[k]; i >= 1; i--)
94              last[k][i] = pos[k][S[k][i]],
95              pos[k][S[k][i]] = i;
96      // 开始求
97      hsuf[len[0] + 1][len[1] + 1] = h[len[0] + 1][len[1] + 1] = 1;
98      // 初始化边界
99      for (i = 1; i <= len[1]; i++)
100         hsuf[len[0] + 1][i] = 1;
101     for (i = 1; i <= len[0]; i++)
102         hsuf[i][len[1] + 1] = 1;
103     for (i = len[0]; i >= 1; i--)
104         for (j = len[1]; j >= 1; j--)
105             h[i][j] = sufsum(last[0][i], last[1][j], i + 1, j + 1),
106             hsuf[i][j] = hsuf[i + 1][j] + hsuf[i][j + 1] - hsuf[i + 1][j + 1] + h[i][j];
107     // 计算长度相等时的答案
108     for (i = 1; i <= len[0]; i++)
109         for (j = 1; j <= len[1]; j++)
110             if (S[0][i] > S[1][j])
111                 ans = (ans + gpre[i - 1][j - 1] * h[i][j]) % P;
112
113     cout << ans << endl;
114
115     return 0;
116 }
```

## 2.6 约瑟夫问题

$n$ 个人标号 $0, 1, \ldots, n-1$。逆时针站成一圈，从 0 号开始，每一次从当前的人逆时针数 m 个，然后让这个人出局。问最后剩下的人是谁。
参考：https://fancypei.github.io/JosephusProblem/

### 2.6.1 递推法

时间 $O(n)$，空间 $O(1)$
令 $f(n)$ 表示 n 个人最后的幸存者编号，则递推方程为：

$$f(n) = (f(n-1) + m) \bmod n$$

初始条件 $f(1) = 0$

```
// 共n个人，编号从0到n-1，从编号为0的人开始报数，每报到m的人被抬走(从1开始报数)。返回最后存活的人的编号
inline int solve1(int n, int m) {
    int i, ret = 0;
    for (i = 2; i <= n; i++)
        ret = (ret + m) % i;
    return ret;
}
```

### 2.6.2 另类递归

时间 $O(logN)$，空间 $O(logN)$
一次去掉 $\lfloor \frac{n}{m} \rfloor$ 个数，其中 n 是当前剩余个数，m 不变

```
// 时空O(logN)
inline int solve0(int n, int m) {
    if (n == 1)
        return 0;
    if (n < m)
        return (solve0(n - 1, m) + m) % n;  // 类似于O(N)递推的f(n)=(f(n-1)+m)%n
    int s = solve0(n - n / m, m) - n % m;
    return s < 0 ? s + n : s + s / (m - 1);
}
```

### 2.6.3 终极方法

时间 $O(logN)$，空间 $O(1)$ 这个方法同时还解决了扩展约瑟夫问题，即第 k 个被抬走的人的编号是什么?

```
// 时间O(logN)，空间O(1)
// 共n人，数m个数，第k个被抬走的(k取[0,n-1])，返回0~n-1中的一个数
inline int solve(int n, int m, int k) {
    int i;
    for (i = (k + 1) * m - 1; i >= n; i = (i - n) + (i - n) / (m - 1))
        ;
    return i;
}
```

## 2.7 反向约瑟夫问题

提问：共 n 人，数 m 个数，编号为 p 的人第几个被抬走?

### 2.7.1 解法

最简单的还是模拟，用双端队列即可。
还有一种方法，时间 $O(MlogN)$。
这个方法隐含在上面“终极方法”的推导过程中，我们固定 p 点，看每次 p 报数之前几个人被抬走以及 p 在本次报数中是否被抬走。

```cpp
// n个人，编号为1~n，报m个数(从1到m)，编号为p的人是第几个被抬走的，返回1~n中的一个数
inline int solve(int n, int m, int p) {
    // a是每次p报数之前被抬走的人数
    int a = p / m;
    // 如果整除则代表p在本次报数就是那个被抬走的人
    while (p % m) {
        // 若本次p没被抬走，那么下一次p是第p+n-a个报数的人
        p = p + n - a;
        // 下一次p报数之前被抬走的人数
        a = p / m;
    }
    // 返回被抬走的人数
    return a;
}
```

# Chapter 3

# 字符串 L

## 3.1 Hash

Hash 的核心思想在于，暴力算法中，单次比较的时间太长了，应当如何才能缩短一些呢?
如果要求每次只能比较 $O(1)$ 个字符，应该怎样操作呢?
我们定义一个**把 string 映射成 int** 的函数 $f$，这个 $f$ 称为是 Hash 函数。
我们需要关注的是时间复杂度和 Hash 的准确率。
通常我们采用的是多项式 Hash 的方法，即

$$f(s) = \sum (s[i] * b^i) (\mathrm{mod} M)$$

其中 $b$ 与 $M$ 互质，且 M 越大错误率越小。(单次匹配错误率 $\frac{1}{M}$，n 次匹配的错误率为 $\frac{n}{M}$)

### 3.1.1 基础 Hash 匹配

```cpp
#include <bits/stdc++.h>

using namespace std;
typedef long long ll;
// b和M互质；M可以尽量取大、随机化。对于本问题，无需建立关于M的数组，所以M最大可以达到1<<31大小。
const ll N = 1e3 + 10, b = 131, M = 1 << 20;

char s[2][N];  // s[0]是待匹配串，s[1]是模式串。下标从1开始
ll len[2];     // 两串的长度
ll Exp[N];     // Exp[i]=b^i

// 返回s[l..r]的哈希值 s[l]*Exp[1]+s[l+1]*Exp[2]+..
inline ll Hash(char s[], ll l, ll r) {
    ll i, ret = 0;
    for (i = 1; l + i - 1 <= r; i++)
        ret = (ret + Exp[i] * s[l + i - 1]) % M;
    return ret;
}

vector<ll> ans;  // 匹配子串的开始下标
inline void match() {
    ans.clear();
    ll i;
    ll h0 = Hash(s[0], 1, len[1]);  // 初始时待匹配串对应模式串那部分子串的哈希值
    ll h1 = Hash(s[1], 1, len[1]);  // 初始时模式串的哈希值
    for (i = 1; i <= len[0] - len[1] + 1; i++) {
        // 若两哈希值一致，则认为匹配
        if ((h0 - h1 * Exp[i - 1]) % M == 0)
            ans.push_back(i);
        h0 = (h0 - Exp[i] * s[0][i] + Exp[i + len[1]] * s[0][i + len[1]]) % M;  // 模式串向后移动，对应h0也要改变
    }
}

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);

    ll i, j;

    // 初始化
    Exp[0] = 1;
    for (i = 1; i < N; i++)
        Exp[i] = Exp[i - 1] * b % M;

    cin >> (s[0] + 1) >> (s[1] + 1);
    len[0] = strlen(s[0] + 1), len[1] = strlen(s[1] + 1);

    match();

    return 0;
}
```

## 3.2 KMP

### 3.2.1 前缀函数

给定一个长度为 n 的字符串 s（假定下标从 1 开始），其**前缀函数**被定义为一个长度为 n 的数组 $\pi$，其中 $\pi[i]$ 为既是子串 $s[1\dots i]$ 的前缀同时也是该子串的后缀的最长真前缀（proper prefix）长度。一个字符串的真前缀是其前缀但不等于该字符串自身。根据定义，$\pi[1]=0$。

前缀函数的定义可用数学语言描述如下：

$$\pi[i]=\max_{k=0\dots i-1}\{k : s[1\dots k]=s[i-k+1\dots i]\}$$

举例来说，字符串 abcabcd 的前缀函数为 $[0,0,0,1,2,3,0]$，字符串 aabaaab 的前缀函数为 $[0,1,0,1,2,2,3]$。

#### 3.2.1.1 朴素算法

直接按定义计算前缀函数：

```
// 朴素法求前缀函数O(n^3)，下标从1开始
void prefix_func0(char t[], int n, int pi[]) {
    int i, k;
    for (i = 1; i <= n; i++)     // 对每一个子串
        for (k = 0; k < i; k++)  // 枚举前缀后缀长度，并判断是否相等
            if (!strncmp(t + 1, t + i - k + 1, k))
                pi[i] = k;
}

```

#### 3.2.1.2 第一个优化

第一个重要的事实是相邻的前缀函数值至多增加 1。（如不然，会产生矛盾）
所以当移动到下一个位置时，前缀函数要么增加 1，要么不变或减少。实际上，该事实已经允许我们将复杂度降至 $O(n^2)$。因为每一步中前缀函数至多增加 1，因此在总的运行过程中，前缀函数至多增加 n，同时也至多减小 n。这意味着我们仅需进行 $O(n)$ 次字符串比较，所以总复杂度为 $O(n^2)$。

```
void prefix_func1(char t[], int n, int pi[]) {
    int i, j;
    i = 2, j = 1;
    while (i <= n) {
        if (t[i] == t[j])  // 加1
            pi[i] = pi[i - 1] + 1, ++j;
        else {  // 开始减
            pi[i] = pi[i - 1];
            while (strncmp(t + i - pi[i] + 1, t + 1, pi[i]))
                --pi[i];
            j = pi[i] + 1;
        }
        ++i;
    }
}

```

#### 3.2.1.3 第二个优化

考虑计算位置 i+1 的前缀函数 $\pi$ 的值，如果 $s[i+1]=s[\pi[i]+1]$，显然 $\pi[i+1]=\pi[i]+1$。

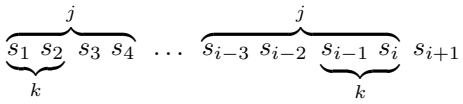
$$\underbrace{\overbrace{s_1\ s_2}^{\pi[i]}\ s_3\ \overbrace{s_4}^{s_4=s_{i+1}}}_{\pi[i+1]=\pi[i]+1}\ \dots\ \underbrace{\overbrace{s_{i-2}\ s_{i-1}}^{\pi[i]}\ s_i\ \overbrace{s_{i+1}}^{s_4=s_{i+1}}}_{\pi[i+1]=\pi[i]+1}$$

如果不是上述情况，即 $s[i+1]\neq s[\pi[i]+1]$，我们需要尝试更短的字符串。为了加速，我们希望直接移动到最长的长度 $j<\pi[i]$，使得在位置 i 的前缀性质仍得以保持，也即 $s[1\dots j]=s[i-j+1\dots i]$：

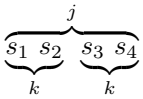
$$\overbrace{\underbrace{s_1\ s_2}_{j}\ s_3\ s_4}^{\pi[i]}\ \dots\ \overbrace{s_{i-3}\ s_{i-2}\ \underbrace{s_{i-1}\ s_i}_{j}}^{\pi[i]}\ s_{i+1}$$

实际上，如果我们找到了这样的 j，我们仅需要再次比较 $s[i+1]$ 和 $s[j+1]$。如果它们相等，则 $\pi[i+1]=j+1$，否则，我们就需要找小于 j 的最大的新的 j 使得前缀性质仍然保持，如此反复，直到 $s[i+1]=s[j+1]$ 或者确实完全找不到（令 $j=-1$）。最后 $\pi[i+1]=j+1$。

所以我们已经有了一个大致框架，现在仅剩的问题是对于满足 $s[1\dots j]=s[i-j+1\dots i]$ 的 j，如何快速找到小于 j 的最大的新的 j，我们令新的 j 为 k，使得 $s[1\dots k]=s[i-k+1\dots i]$ 仍然满足。



由上图，我们要求的是比 j 小的最大的 k，而两边长度为 j 的前后缀本身是相等的，那么新的长为 k 的前后缀则可以只放到最左边长为 j 的前缀中去考虑:



即 $k=\pi[j]$，而 $\pi[j]$ 之前已经求过了。

```
1  void prefix_func2(char t[], int n, int pi[]) {
2      int i, j;
3      pi[0] = -1, pi[1] = 0;  // 确实没有找到任何相等的
4      for (i = 1; i < n; ++i) {
5          j = pi[i];
6          while (j >= 0 && t[j + 1] != t[i + 1])  // 若不相等，找更小的新的j
7              j = pi[j];
8          pi[i + 1] = j + 1;  //最后得出pi[i+1]
9      }
10 }
11
12
```

### 3.2.2 在线 KMP

最基础的字符串匹配。下面的算法不是在线的，但只要稍作修改就可以变成在线的了。
假设当前在 s 串的 i 处，cur 是当前 s 和 t 匹配的最大长度，也就是 t 的长为 cur 的前缀和 s 中以 s[i] 为右端点的子串相等

```
1  // 长为n的待匹配串s，长为m的模式串t，返回t在s中出现的次数
2  int kmp(char s[], int n, char t[], int m, int pi[]) {
3      int cur = 0, i, j, cnt = 0;  // cur是当前pi值
4      for (i = 0; i < n; i++) {
5          j = cur;
6          while (j >= 0 && s[i + 1] != t[j + 1])
7              j = pi[j];
8          cur = j + 1;
9          if (cur == m)
10             ++cnt;
11     }
12     return cnt;
13 }
14
```

### 3.2.3 统计每个前缀出现次数

#### 3.2.3.1 单串统计

统计 s 的每个前缀在 s 中出现的次数。首先我们明确，一个长度为 i 的前缀中会出现长度为 $\pi[i]$ 的前缀，然后长度为 $\pi[i]$ 的前缀又会出现长度为 $\pi[\pi[i]]$ 的前缀，等等。所以我们考虑后缀和的思想，首先统计每一个位置的 $\pi[i]$，然后将 $ans[i]$ 累加给长为 $ans[\pi[i]]$ 的前缀个数，按照长度递减的顺序依次累加下去就可以了。最后再统计原始前缀，即对每个 $ans[i]$ 加一。

```
1  // 统计s[]的每个前缀在s[]中出现的次数
2  inline void count_prefix(char s[], int n, int pi[]) {
3      prefix_func(s, n, pi);  // 计算前缀函数
```

```
    int i;
    for (i = 1; i <= n; i++)
        ++ans[pi[i]];
    // 令真前后缀长度为i，其个数为ans[i]，则长为i的真前后缀的真前后缀长为pi[i]，其原本个数为ans[pi[i]]
    // 现在需要累加上它在更长的真前后缀中出现的次数。有点类似倍增
    for (i = n; i >= 1; i--)
        ans[pi[i]] += ans[i];
    // 这里不能放到开头，否则，一开始ans[n]=1，也就认为s有一个长为n的真前后缀
    for (i = 1; i <= n; i++)
        ++ans[i];
}

```

#### 3.2.3.2 双串统计

给出串 s 和 t，问 t 的每个前缀在 s 中出现的次数。首先运用类似 KMP 的思想，通过'#' 连接 t 和 s，即"t#s"，设为 link，对 link 求前缀函数。接下来我们只关心与 s 有关的前缀函数值，即 $i \geq m+2$ 的 $\pi[i]$。

```
inline void count_prefix(char s[], int n, char t[], int m, int pi[]) {
    // 连接字符串 t..#s..
    strcpy(link + 1, t + 1);
    link[m + 1] = '#';
    strcpy(link + m + 2, s + 1);
    // 计算前缀函数
    prefix_func(link, n + m + 1, pi);

    int i;
    // 只关心#后面的pi值
    for (i = m + 2; i <= n + m + 1; i++)
        ++ans[pi[i]];
    // pi[i]的值不会超过t的长度，即i<=m
    for (i = m; i >= 1; i--)
        ans[pi[i]] += ans[i];
}

```

### 3.2.4 统计一个字符串本质不同的子串的数目

给定一个长度为 n 的字符串 s，我们希望计算其本质不同子串的数目。
假设现在知道了当前 s 本质不同的子串的数目，那么接下来可以考虑在原来的 s 末尾加上一个字符 c，然后统计产生了多少新的子串。
我们枚举 $i$，对每一个 $i$，反转 $s[1 \dots i]$ 令其为 t，然后对 t 求前缀函数，$\pi_{max}$ 即为以 $s[i]$ 结尾的重复子串的数目，那么 $|s| - \pi_{max}$ 即为以 $s[i]$ 结尾的新的子串的数目。

```
// 统计串s中本质不同的子串数目
inline int diff(char s[], int n) {
    int i, ret = 0;
    for (i = 1; i <= n; i++) {
        reverse_copy(s + 1, s + 1 + i, t + 1);  // 翻转s并存入t
        int mx = prefix_func2(t, i, pi);        // 略微修改一下prefix_func，使其可以返回最大的pi值
        ret += i - mx;                          // 统计新的子串数目
    }
    return ret;
}

```

### 3.2.5 字符串压缩

给定一个长度为 n 的字符串 s，我们希望找到其最短的“压缩”表示，也即我们希望寻找一个最短的字符串 t，使得 s 可以被 t 的一份或多份拷贝的拼接表示。
显然，我们只需要找到 t 的长度即可。知道了长度，该问题的答案即为长度为该值的 s 的前缀。
直接说结论，首先求 s 的前缀函数，令 $k = n - \pi[n]$，若 $n \bmod k = 0$，则该长度为 $k$，否则为 $n$。
**注：上述情况为 s 恰好为最短循环节的倍数，如果没有这个限制，那么对于 $s[1 \dots i]$ 最短循环节长度永远是 $k = i - \pi[i]$。**

```
// 计算s的最短压缩表示，返回最短压缩的长度
int compress(char s[], int n, int pi[]) {
    prefix_func(s, n, pi);
    int k = n - pi[n];
    if (n % k)
        return n;
    return k;
}

```

### 3.2.6 根据前缀函数构建一个自动机

让我们重新回到通过一个分割符将两个字符串拼接的新字符串。设模式串为 t，待匹配串为 s，则拼接成 $t+\#+s$。前面我们就知道，我们只需要管 $t+\#$ 的前缀函数值就可以，所以在这里我们的自动机也是如此。
自动机的状态为当前前缀函数的值，而下一个读入自动机的字符则决定状态如何转移。下面我们就是要求状态转移表 $aut$，$aut[i][c]$ 表示当前前缀函数值为 $i$（也即当前处于 $t[i]$ 处）下一个字符为 $c$ 的转移的目标状态。

```
// 计算长度为n的字符串t[]的自动机的转移表，t[]下标从1开始
void compute_automaton0(char t[], int n, int pi[], int aut[][26]) {
    prefix_func(t, n, pi);  // 先求t[]的前缀函数
    int i, j, c;
    // 0是初始状态，匹配长度为0；n是终止状态，完全匹配
    for (i = 0; i <= n; i++) {
        // 转移的因素--下一个字符
        for (c = 0; c < 26; c++) {
            j = i;
            // 通过不断跳转前缀来匹配下一个字符
            while (j >= 0 && c + 'a' != t[j + 1])
                j = pi[j];
            aut[i][c] = j + 1;
        }
    }
}

```

在上面的代码中，由于有 while 循环的存在，所以时间复杂度为 $O(|\Sigma|n^2)$。
事实上，我们可以通过动态规划来优化。我们注意到当下一个字符 $c \neq s[j+1]$ 时，$j$ 会跳转至 $\pi[j]$，而在之前我们已经计算过所有 $aut[\pi[j]][c], \forall c \in \Sigma$，所以我们可以直接利用 $aut[\pi[i]][c]$。时间复杂度 $O(|\Sigma|n)$。

```
// 计算长度为n的字符串t[]的自动机的转移表，t[]下标从1开始
void compute_automaton1(char t[], int n, int pi[], int aut[][26]) {
    prefix_func(t, n, pi);  // 先求t[]的前缀函数
    int i, c;
    // 0是初始状态，匹配长度为0；n是终止状态，完全匹配
    aut[0][t[1] - 'a'] = 1;
    for (i = 1; i <= n; i++) {
        // 转移的因素--下一个字符
        for (c = 0; c < 26; c++) {
            // 利用动态规划的思想来优化，在上面j跳转至pi[j]时，aut[pi[j]][c]对于所有的c都已经被计算过了
            if (c + 'a' == t[i + 1])
                aut[i][c] = i + 1;
            else
                aut[i][c] = aut[pi[i]][c];
        }
    }
}

```

### 3.2.7 Gray 字符串

首先定义 Gray 字符串，令 $g[0]=$ ""（空串），$g[1]=$ "1"，之后 $g[i]=g[i-1]+i+g[i-1]$，加号为字符串拼接。
接下来我们考虑这样一个问题（与 OIWIKI 不同的是数据范围作了修改）：给定长为 $n(n \leq 1000)$ 的字符串 $s(1 \leq s[i] \leq n)$，一个整数 $k(k \leq 1000)$，要求 s 在 g[k] 中出现的次数。

这题是 kmp 自动机 +dp。

显然，g[k] 的长度非常大，我们不可能去构造。但我们可以好好利用 Gray 字符串递归的性质，即 $g[i] = g[i-1] + i + g[i-1]$。
对 s 构建一个**自动机 aut[i][j]**，表示从当前状态 i 通过输入的 j 到达的状态。
假设当前自动机处于状态 i，接下来要处理 g[j]，我们可以分为 3 步：
1. 从状态 i 开始处理 g[j-1]，自动机到达状态 t1
2. 从状态 t1 开始处理 j，自动机到达状态 t2
3. 从状态 t2 开始处理 g[j-1]，自动机到达状态 t3。
其中 t3 即为目标状态。

显然，如果每一步都老老实实做的话，工作量并未减少。仔细观察发现我们可以建立一个 dp 状态，**G[i][j]** 表示自动机从状态 i 开始处理 g[j]，处理完成后自动机所处的状态。那么上面的 3 步就可以变为如下形式：

$$t1 = G[i][j-1]$$

$$t2 = aut[t1][j]$$

$$t3 = G[t2][j-1]$$

由于 $i \le n$ 且 $j \le k$ 复杂度 $O(nk)$。
如何计算答案呢？我们只要在自动机转移的过程中看当前状态是否为 $|s|$ 状态（和 s 完全匹配）即可。但是我们上面都是一跳一大步，可能有的 $|s|$ 状态就给跳过去了。所以我们再用一个 **K[i][j]** 记录自动机从状态 i 开始处理 g[j] 直到处理完成，这个过程中几次达到状态 $|s|$。显然有：

$$K[i][j] = K[i][j-1] + (t2 == |s|) + K[t2][j-1]$$

初始时，aut[i][j] 可全部求出，G[i][0]=i，K[i][0]=0，因为 g[0] 为空串，自动机不会转移，g[0] 也不会包含 s。
显然，最后答案为 K[0][k]。

```cpp
#include <bits/stdc++.h>

using namespace std;
const int N = 1e3 + 10;

int s[N];
int n, k;
int pi[N];
int aut[N][N];  // 自动机，aut[i][j]表示当前状态通过输入j得到的下一个状态
int G[N][N];    // G(i,j)表示自动机从状态i开始处理g[j]，处理完成后自动机的状态
int K[N][N];    // K(i,j)表示自动机从状态i开始处理g[j]，处理完成后s在g[j]中出现的次数

void prefix_func(int t[], int n, int pi[]) {
    int i, j;
    pi[0] = -1, pi[1] = 0;  // 确实没有找到任何相等的
    for (i = 1; i < n; ++i) {
        j = pi[i];
        while (j >= 0 && t[j + 1] != t[i + 1])  // 若不相等，找更小的新的j
            j = pi[j];
        pi[i + 1] = j + 1;  //最后得出pi[i+1]
    }
}

// 计算长度为n的字符串t[]的自动机的转移表，t[]下标从1开始
void compute_automaton1(int t[], int n, int pi[], int aut[][N]) {
    t[n + 1] = -1;          // 结束符，结束符应是字母表中没有的符号
    prefix_func(t, n, pi);  // 先求t[]的前缀函数
    int i, c;
    // 0是初始状态，匹配长度为0；n是终止状态，完全匹配
    aut[0][t[1]] = 1;
    for (i = 1; i <= n; i++) {
        // 转移的因素--下一个字符
        for (c = 1; c <= n; c++) {
            // 利用动态规划的思想来优化，在上面j跳转至pi[j]时，aut[pi[j]][c]对于所有的c都已经被计算过了
            if (c == t[i + 1])
                aut[i][c] = i + 1;
            else
                aut[i][c] = aut[pi[i]][c];
        }
```

```
    }
}

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);

    int i, j;
    cin >> n >> k;
    for (i = 1; i <= n; i++)
        cin >> s[i];

    compute_automaton1(s, n, pi, aut);

    // 初始化，从状态i开始处理g[0]，由于g[0]是空串，所以自动机不会转移，s也不会出现在g[0]中
    for (i = 0; i <= n; i++)
        G[i][0] = i, K[i][0] = 0;
    // 类似于动态规划的递推
    for (j = 1; j <= k; j++) {
        for (i = 0; i <= n; i++) {
            int mid = aut[G[i][j - 1]][j];
            G[i][j] = G[mid][j - 1];
            K[i][j] = K[i][j - 1] + (n == mid) + K[mid][j - 1];
        }
    }

    cout << K[0][k] << endl;

    return 0;
}

```

### 3.2.8 UVA11022 String Factoring

给定一个字符串 $s(|s| \le 80)$，问最小压缩长度。例如，$AAA = (A)^3$ 最小长度为 1，$CABAB = C(AB)^2$ 最小长度为 3，再如 $POPPOP$ 既可以是 $PO(P)^2OP$ 也可以是 $(POP)^2$，但后者长度为 3，前者为 5，所以后者更优。

这题是 kmp 字符串压缩 + 区间 dp。
f[i][j] 表示 s[i..j] 被压缩后的最短长度，pi2[i][j] 表示以 i 为左端点，j 处的前缀函数值。先考虑最简单的区间 dp：

$$f(l,r) = \min_{l \le k < r} \{f(l,m) + f(m+1,r)\}$$

可以发现，这是**不完整**的，例如 ATTATT，假设已知 $f(1,3) = 2$ 和 $f(4,6) = 2$，接下来算 $f(1,6)$ 就会等于 4，而正确答案是 2。也就是说，我们没有考虑 s[l..r] 可能本身就可以被压缩。若 s[l..r] 本身可以被压缩，则一定存在一个最小循环节，这我们可以通过上面的 pi2[l][r] 来求。
若确实存在循环节，设其长度为 k，则 $f(l,r) = f(l, l+k-1)$，之所以这样，是因为循环节本身也是可以被压缩的（类似子问题）。之后再用上面的区间 dp，就是对的了。

```
#include <bits/stdc++.h>

using namespace std;
const int N = 85;

char s[N];
int n;
int pi[N], pi2[N][N];  // pi2[i][j]表示以i为左端点，j处的前缀函数值
int f[N][N];           // f[i][j]表示s[i..j]被压缩后的最短长度

void prefix_func(char t[], int n, int pi[]) {
    int i, j;
    pi[0] = -1, pi[1] = 0;  // 确实没有找到任何相等的
    for (i = 1; i < n; ++i) {
        j = pi[i];
        while (j >= 0 && t[j + 1] != t[i + 1])  // 若不相等，找更小的新的j
            j = pi[j];
```

```
        pi[i + 1] = j + 1;  //最后得出pi[i+1]
    }
}

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);

    int i, j, k, l, r, m;

    while (cin >> (s + 1)) {
        if (s[1] == '*')
            break;
        n = strlen(s + 1);

        // 求pi2[i][j]
        for (i = 1; i <= n; i++)
            prefix_func(s + i - 1, n - i + 1, pi2[i] + i - 1);

        // dp求f(l,r), s[l..r]被压缩后的最短长度
        for (i = 1; i <= n; i++) {                  // 枚举长度
            for (l = 1; l + i - 1 <= n; l++) {  // 枚举左端点
                r = l + i - 1;                      // 右端点
                f[l][r] = i;                        // 初始化为最坏情况
                int len = r - l + 1;                // 当前区间的长度
                // 如果本身完全是循环节
                k = len - pi2[l][r];
                if (len % k == 0)
                    f[l][r] = min(f[l][r], f[l][l + k - 1]);  // 循环节的性质，转化为已求的子问题
                // 区间DP
                for (m = l; m < r; m++)
                    f[l][r] = min(f[l][r], f[l][m] + f[m + 1][r]);
            }
        }

        cout << f[1][n] << endl;
    }

    return 0;
}

```

### 3.2.9 UVA12467 Secret word

给你一个串 $S(1 \le |S| \le 10^6)$，要求最长前缀，这种前缀满足：它的反转是 S 的子串。

前缀的反转是 S 的子串，换句话说，我们将 S 反转，若它的某一个子串为原串 S 的前缀，则该子串的反转就是满足条件的子串之一。
令 $S'$ 为 $S$ 的反转，则 $S$ 为模式串，$S'$ 为待匹配串，对 $S$ 和 $S'$ 用 kmp，匹配过程中的最大 $\pi$ 值即为答案。

### 3.2.10 UVA11019 Matrix Matcher

给定一个 $n*m$ 的大矩阵和一个 $x*y$ 的小矩阵，问小矩阵在大矩阵中出现的次数。

令大矩阵为 s[][]，小矩阵为 t[][]，最暴力的想法：
设 $pi2[i][j]$ 为 t[i] 在 j 处的前缀函数值。
设 $f(i,j,k)$ 为以 t[i] 为模式串，s[j] 为待匹配串，匹配到 s[j][k] 处的前缀函数值。
pi2 可以直接通过 x 次 KMP 求出，f 可以通过枚举 (i,j) 对 t[i] 和 s[j] 用 KMP 求出。
最后枚举 t 在 s 中的左上角，判断每一行匹配到最后的值是否为 y 即可。具体来说，设当前 s 左上角为 (i,j)，对 t 的第 k 行匹配，若 $\forall k \in [1,x], f[k][i+k-1][j+y-1] = y$，则匹配成功，否则失败。
事实上，这题正解应该是 **AC 自动机**。

### 3.2.11 cf808G Anthem of Berland

给出两个串 s 和 t，s 中有一部分是问号（t 是确定的），问号可以是任意字符，问 t 在 s 中出现的最大次数。

最暴力的做法是直接 dfs 搜索，时间爆炸。
由于每一个问号处都可能匹配到 t 的任意位置，所以可以考虑 dp 优化，设状态为 $f(i,j)$，表示 s[i] 处匹配到 t[j] 处时 t 在 s 中出现的最大次数。如何转移呢？当 s[i] 处完全匹配 t 串，此时 $f(i,j)$ 的值就要加 1，否则不变。但是光这样还不够，考虑 j 处的 $\pi[j]$ 值，这意味着 t 的长度为 $\pi[j]$ 的前后缀是相等的，也就是说值 $f(i,j)$ 也可以传递给 $f(i,\pi[j])$ 以及 $f(i,\pi[\pi[j]])$ 等等。
如果对每个 j 还要枚举 $\pi[j]$，时间就会爆炸。所以我们还要借用前缀和的思想，$h(i,c)$ 表示当前 $\pi$ 值为 i 且下一个字符为 c 的最优答案。我们按 j 从大到小处理，在处理 j 时，我们同时给 $h[\pi[j]][t[j+1]]$ 更新，而当前的 $f(i,j)$ 也与 $h(j,s[i+1])$ 有关。

# Chapter 4

# 字符串

## 4.1 字符串 Hash

```
#define ull unsigned long long
#define P 131
ull f[N], p[N];
void Init()
{
    p[0] = 1, f[0] = 0;
    for (int i = 1; i <= n; i++)
    {
        p[i] = p[i - 1] * P;
        f[i] = f[i - 1] * P + str[i];
    }
}
ull Hash(int left, int right)
{
    return f[right] - f[left - 1] * p[right - left + 1];
}
```

### 4.1.1 应用：后缀数组

```
#define ull unsigned long long
#define P 131
const int N = 3e5 + 50;
ull f[N], p[N];
char str[N];
int SA[N], n, Height[N];
void Init()
{
    p[0] = 1, f[0] = 0;
    for (int i = 1; i <= n; i++)
    {
        p[i] = p[i - 1] * P;
        f[i] = f[i - 1] * P + str[i];
    }
}
ull Hash(int left, int right)
{
    return f[right] - f[left - 1] * p[right - left + 1];
}
// k:[0,n) 表示后缀S(k,n-1)
// 最长公共前缀
int LCP(int a, int b)
{
    int left = 0;
    int right = N;
    int mid;
    while (left < right)
    {
        mid = (left + right + 1) >> 1;
        if (a + mid - 1 <= n && b + mid - 1 <= n && Hash(a, a + mid - 1) == Hash(b, b + mid - 1))
            left = mid;
        else
            right = mid - 1;
    }
    return left;
}
bool cmp(int a, int b)
{
    int len = LCP(a, b);
    return str[a + len] < str[b + len];
}
void calc_height()
{
    Height[1] = 0;
```

```
45      for (int i = 2; i <= n; i++)
46          Height[i] = LCP(SA[i], SA[i - 1]);
47  }
48  int main()
49  {
50      scanf("%s", str + 1);
51      n = strlen(str + 1);
52      for (int i = 1; i <= n; i++)
53          SA[i] = i;
54      Init();
55      sort(SA + 1, SA + n + 1, cmp);
56      calc_height();
57  }
```

### 4.1.2 应用：二维 Hash

给定一个 M 行 N 列的 01 矩阵（只包含数字 0 或 1 的矩阵），再执行 Q 次询问，每次询问给出一个 A 行 B 列的 01 矩阵，求该矩阵是否在原矩阵中出现过。

做法：选取两个不同的 P 值分别对行列进行 Hash 处理，应用二维前缀和求取矩阵 Hash 值。

```
1   #define P 131
2   #define Q 13331
3   #define ull unsigned long long
4   void Init()
5   {
6       char ch;
7       for (int i = 1; i <= m; i++)
8           for (int j = 1; j <= n; j++)
9               cin >> ch, Hash[i][j] = Hash[i][j - 1] * P + ch;
10      for (int i = 1; i <= m; i++)
11          for (int j = 1; j <= n; j++)
12              Hash[i][j] += Hash[i - 1][j] * Q;
13  }
14  ull temp = Hash[i][j] - Hash[i - a][j] * q[a] - Hash[i][j - b] * p[b] + Hash[i - a][j - b] * q[a]
        * p[b];
```

### 4.1.3 应用：一类同构判定的问题

参考：杨弋《Hash 在信息学竞赛中的一类应用》

## 4.2 后缀自动机

```cpp
#define ll long long
const int MAXLEN = 1e6 + 50;
struct SAM
{
    int len[MAXLEN << 1], link[MAXLEN << 1], next[MAXLEN << 1][26];
    ll sze[MAXLEN << 1]; ////每个结点所代表的字符串的出现次数
    int sz, last, rt;
    int NewNode(int x = 0)
    {
        len[sz] = x;
        link[sz] = -1;
        memset(next[sz], -1, sizeof(next[sz]));
        return sz++;
    }
    void Init()
    {
        //重置
        sz = last = 0, rt = NewNode();
    }
    void Extend(int c)
    {
        int cur = NewNode(len[last] + 1);
        sze[cur] = 1;
        int p = last;
        while (~p && next[p][c] == -1)
            next[p][c] = cur, p = link[p];
        if (p == -1)
            link[cur] = rt;
        else
        {
            int q = next[p][c];
            if (len[q] == len[p] + 1)
                link[cur] = q;
            else
            {
                int clone = NewNode(len[p] + 1);
                memcpy(next[clone], next[q], sizeof(next[q]));
                link[clone] = link[q], link[q] = link[cur] = clone;
                while (~p && next[p][c] == q)
                    next[p][c] = clone, p = link[p];
            }
        }
        last = cur;
    }
    int id[MAXLEN << 1], c[MAXLEN];
    void Topo()
    {
        //计数排序
        memset(c, 0, sizeof(c));
        for (int i = 0; i < sz; i++)
            c[len[i]]++;
        for (int i = 1; i < MAXLEN; i++)
            c[i] += c[i - 1];
        for (int i = 0; i < sz; i++)
            id[--c[len[i]]] = i;
        for (int i = sz - 1; ~i; i--)
        {
            int u = id[i];
            if (~link[u])
                sze[link[u]] += sze[u];
        }
    }
};
```

## 4.3 Manacher

```
const int MAXLEN = 1e5 + 50;
char ori[MAXLEN], str[MAXLEN * 2];
int d1[MAXLEN * 2], n, m;
void Manacher()
{
    for (int i = 0, l = 0, r = -1; i < m; i++)
    {
        int k = (i > r) ? 1 : min(d1[l + r - i], r - i);
        while (i - k >= 0 && i + k < m && str[i - k] == str[i + k])
            k++;
        d1[i] = k--;
        if (i + k > r)
            l = i - k, r = i + k;
    }
}
int main()
{
    scanf("%s", ori);
    n = strlen(ori);
    str[0] = '#';
    for (int i = 0; i < n; i++)
    {
        str[(i + 1) * 2] = '#';
        str[(i + 1) * 2 - 1] = ori[i];
    }
    m = n * 2 + 1;
    Manacher();
}
```

## 4.4 回文树/回文自动机

```
const int MAXLEN=5e5+50;
struct Palindromic_Tree
{
    int nxt[MAXLEN][26],fail[MAXLEN],len[MAXLEN],s[MAXLEN];
    int cnt[MAXLEN];// 结点表示的本质不同的回文串的个数(调用Count()后)
    int num[MAXLEN];// 结点表示的最长回文串的最右端点为回文串结尾的回文串个数
    int last,sz,n;
    int NewNode(int x)
    {
        memset(nxt[sz],0,sizeof(nxt[sz]));
        cnt[sz]=num[sz]=0,len[sz]=x;
        return sz++;
    }
    void Init()
    {
        sz=0;
        NewNode(0),NewNode(-1);
        last=n=0,s[0]=-1,fail[0]=1;
    }
    int GetFail(int u)
    {
        while(s[n-len[u]-1]!=s[n]) u=fail[u];
        return u;
    }
    void Add(int c)
    {
        //c-='a'
        s[++n]=c;
        int u=GetFail(last);
        if(!nxt[u][c])
        {
```

```
32              int np=NewNode(len[u]+2);
33              fail[np]=nxt[GetFail(fail[u])][c];
34              num[np]=num[fail[np]]+1;
35              nxt[u][c]=np;
36          }
37          last=nxt[u][c];
38          cnt[last]++;
39      }
40      void Count()
41      {
42          for(int i=sz-1;~i;i--)
43              cnt[fail[i]]+=cnt[i];
44      }
45  }
```

## 4.5 AC 自动机

```
1   #include <cstring>
2   #include <iostream>
3   #include <queue>
4   #include <stdio.h>
5   using namespace std;
6   const int MAXLEN = 1e6 + 50;
7   struct Trie
8   {
9       int tr[MAXLEN][26], fail[MAXLEN], End[MAXLEN];
10      int sz, rt;
11      int NewNode()
12      {
13          memset(tr[sz], 0, sizeof(tr[sz]));
14          End[sz] = 0, fail[sz] = 0; //注意重置
15          return sz++;
16      }
17      void Init()
18      {
19          sz = 0, rt = NewNode();
20      }
21      void Insert(const char *s)
22      {
23          int p = rt;
24          for (int i = 0; s[i]; i++)
25          {
26              int c = s[i] - 'a';
27              if (!tr[p][c])
28                  tr[p][c] = NewNode();
29              p = tr[p][c];
30          }
31          End[p]++;
32      }
33      void Build()
34      {
35          queue<int> q;
36          for (int c = 0; c < 26; c++)
37              if (tr[rt][c])
38                  q.push(tr[rt][c]);
39          while (!q.empty())
40          {
41              int u = q.front();
42              q.pop();
43              for (int c = 0; c < 26; c++)
44              {
45                  if (tr[u][c])
46                      fail[tr[u][c]] = tr[fail[u]][c], q.push(tr[u][c]);
47                  else
```

```
48                      tr[u][c] = tr[fail[u]][c];
49              }
50          }
51      }
52      int Query(const char *s)
53      {
54          int u = rt, res = 0;
55          for (int i = 0; s[i]; i++)
56          {
57              u = tr[u][s[i] - 'a']; // 转移
58              for (int j = u; j && End[j] != -1; j = fail[j])
59              {
60                  res += End[j], End[j] = -1;
61              }
62          }
63          return res;
64      }
65  } ac;
66  char str[MAXLEN];
67  int main()
68  {
69      int n;
70      scanf("%d", &n);
71      ac.Init();
72      for (int i = 0; i < n; i++)
73      {
74          scanf("%s", str);
75          ac.Insert(str);
76      }
77      ac.Build();
78      scanf("%s", str);
79      printf("%d\n", ac.Query(str));
80      //system("pause");
81      return 0;
82  }
```

## 4.6 KMP

```
// KMP
void calc_next(){
    next[1] = 0;
    for (int i = 2, j = 0; i <= n; i++) {
        while (j > 0 && a[i] != a[j+1]) j = next[j];
        if (a[i] == a[j+1]) j++;
        next[i] = j;
    }
}
void KMP(){
    for (int i = 1, j = 0; i <= m; i++) {
        while (j > 0 && (j == n || b[i] != a[j+1])) j = next[j];
        if (b[i] == a[j+1]) j++;
        f[i] = j;
        // if (f[i] == n), 此时就是a在b中的某一次出现
    }
}
```

字符串循环元可利用 next 数组求解。

## 4.7 最小表示法

```
// 最小表示法
int n = strlen(s + 1);
for (int i = 1; i <= n; i++) s[n+i] = s[i];
int i = 1, j = 2, k;
while (i <= n && j <= n) {
    for (k = 0; k < n && s[i+k] == s[j+k]; k++);
    if (k == n) break; // s likes "aaaaa"
    if (s[i+k] > s[j+k]) {
        i = i + k + 1;
        if (i == j) i++;
    } else {
        j = j + k + 1;
        if (i == j) j++;
    }
}
ans = min(i, j);
```

# Chapter 5

# 数据结构

## 5.1 并查集

```
#define MAX 1010
struct node
{
    int par;
    //int rank;
    //路径压缩后 rank=1或2 rank失去了意义
    int data;
};
node ns[MAX];
void Init()
{
    for (int i = 1; i < MAX; i++)
    {
        ns[i].par = i;
    }
}
int Find(int i)
{
    if (ns[i].par == i)
    {
        //返回根结点
        return i;
    }
    ns[i].par = Find(ns[i].par);
    //路径压缩
    return ns[i].par;
}
void Union(int i, int j)
{
    int pi = Find(i);
    int pj = Find(j);
    if (pi != pj)
    {
        ns[pi].par = pj;
    }
}
```

## 5.2 树状数组

推荐阅读：*https://www.cnblogs.com/RabbitHu/p/BIT.html*

### 5.2.1 单点修改，区间查询

```
#define N 1000100
long long c[N];
int n,q;
int lowbit(int x)
{
    return x&(-x);
}
void change(int x,int v)
{
    while(x<=n)
    {
        c[x]+=v;
        x+=lowbit(x);
    }
}
long long getsum(int x)
{
    long long ans=0;
    while(x>=1)
    {
        ans+=c[x];
        x-=lowbit(x);
    }
    return ans;
}
```

例题：*https://loj.ac/problem/130*

### 5.2.2 区间修改，单点查询

引入差分数组来解决树状数组的区间更新

```
//初始化
change(i,cur-pre);
//区间修改
change(l,x);
change(r+1,-x);
//单点查询
getsum(x)
```

例题：*https://loj.ac/problem/131*

### 5.2.3 区间修改，区间查询

```
//初始化
change(c1,i,cur-pre);
change(c2,i,i*(cur-pre));
//为什么这么写？ 你需要写一下前缀和的表达式
//区间修改
change(c1,l,x);
change(c2,l,l*x);
change(c1,r+1,-x);
change(c2,r+1,-(r+1)*x);
//区间查询
temp1=l*getsum(c1,l-1)-getsum(c2,l-1);
temp2=(r+1)*getsum(c1,r)-getsum(c2,r);
ans=temp2-temp1
```

例题：*https://loj.ac/problem/132*

## 5.3 二维树状数组

### 5.3.1 单点修改，区间查询

```
#define N 5050
long long tree[N][N];
long long n,m;
long long lowbit(long long x)
{
    return x&(-x);
}
void change(long long x,long long y,long long val)
{
    long long init_y=y;
    //这里注意n,m的限制
    while(x<=n)
    {
        y=init_y;
        while(y<=m)
        {
            tree[x][y]+=val;
            y+=lowbit(y);
        }
        x+=lowbit(x);
    }
}
long long getsum(long long x,long long y)
{
    long long ans=0;
    long long init_y=y;
    while(x>=1)
    {
        y=init_y;
        while(y>=1)
        {
            ans+=tree[x][y];
            y-=lowbit(y);
        }
        x-=lowbit(x);
    }
    //这里画图理解
    return ans;
}
//初始化
change(x,y,k);
//二维前缀和
ans = getsum(c,d)+getsum(a-1,b-1)-getsum(a-1,d)-getsum(c,b-1);
```

例题：*https://loj.ac/problem/133*

### 5.3.2 区间修改，区间查询

```
#define N 2050
long long t1[N][N];
long long t2[N][N];
long long t3[N][N];
long long t4[N][N];
long long n,m;
long long lowbit(long long x)
{
    return x&(-x);
}
long long getsum(long long x,long long y)
{
    long long ans=0;
    long long init_y=y;
```

```
15      long long init_x=x;
16      while(x>=1)
17      {
18          y=init_y;
19          while(y>=1)
20          {
21              ans+=(init_x+1)*(init_y+1)*t1[x][y];
22              ans-=(init_y+1)*t2[x][y];
23              ans-=(init_x+1)*t3[x][y];
24              ans+=t4[x][y];
25              y-=lowbit(y);
26          }
27          x-=lowbit(x);
28      }
29      return ans;
30  }
31  void change(long long x,long long y,long long val)
32  {
33      long long init_x=x;
34      long long init_y=y;
35      while(x<=n)
36      {
37          y=init_y;
38          while(y<=m)
39          {
40              t1[x][y]+=val;
41              t2[x][y]+=init_x*val;
42              t3[x][y]+=init_y*val;
43              t4[x][y]+=init_x*init_y*val;
44              y+=lowbit(y);
45          }
46          x+=lowbit(x);
47      }
48  }
49  //区间修改
50  change(c+1,d+1,x);
51  change(a,b,x);
52  change(a,d+1,-x);
53  change(c+1,b,-x);
54  //区间查询
55  ans=getsum(c,d)+getsum(a-1,b-1)-getsum(c,b-1)-getsum(a-1,d);
```

例题：*https://loj.ac/problem/135*

## 5.4 线段树

### 5.4.1 基础操作

```
const int N = 1e5 + 10;
#define ls(a) (a << 1)
#define rs(a) (a << 1 | 1)
struct node
{
    int val;
    int lazy;
};
node tree[N << 2];
int a[N];
void PushUp(int rt)
{
    tree[rt].val = tree[ls(rt)].val + tree[rs(rt)];
}
void PushDown(int ls, int rs, int rt)
{
    tree[ls(rt)].val += ls * tree[rt].lazy;
    tree[rs(rt)].val += rs * tree[rt].lazy;
    tree[ls(rt)].lazy += tree[rt].lazy;
    tree[rs(rt)].lazy += tree[rt].lazy;
    tree[rt].lazy = 0;
}
void Build(int left, int right, int rt)
{
    if (left == right)
    {
        tree[rt].val = a[left];
        return;
    }
    int mid = (left + right) >> 1;
    Build(left, mid, ls(rt));
    Build(mid + 1, right, rs(rt));
    PushUp(rt);
    //向上更新
}
```

### 5.4.2 单点更新

```
void Update(int left, int right, int rt, int pos, int val)
{
    if (left == right && left == pos)
    {
        tree[rt].val += val;
        return;
    }
    int mid = (left + right) >> 1;
    if (tree[rt].lazy)
    {
        PushDown(mid - left + 1, right - mid, rt);
    }
    if (mid >= pos)
        Update(left, mid, ls(rt), pos, val);
    else if (pos > mid)
        Update(mid + 1, right, rs(rt), pos, val);
    PushUp(rt);
}
```

例题：*https://www.luogu.org/problemnew/show/P3372*

### 5.4.3 区间更新

```
void Update(int left, int right, int rt, int s, int t, int val)
{
    if (left >= s && right <= t)
    {
        tree[rt].val += (right - left + 1) * val;
        tree[rt].lazy += val;
        return;
    }
    int mid = (left + right) >> 1;
    if (tree[rt].lazy)
    {
        PushDown(mid - left + 1, right - mid, rt);
    }
    if (mid < s)
        Update(mid + 1, right, rs(rt), s, t, val);
    else if (mid >= t)
        Update(left, mid, ls(rt), s, t, val);
    else
    {
        Update(left, mid, ls(rt), s, t, val);
        Update(mid + 1, right, rs(rt), s, t, val);
    }
    PushUp(rt);
}
```

### 5.4.4 区间查询

```
void Query(int left, int right, int s, int t, int rt)
{
    if (left >= s && right <= t)
    {
        return tree[rt].val;
    }
    int mid = (left + right) >> 1;
    if (tree[rt].lazy)
        PushDown(mid - left + 1, right - mid, rt);
    long long sum = 0;
    if (mid < s)
        sum += Query(mid + 1, right, rs(rt), s, t, val);
    else if (mid >= t)
        sum += Query(left, mid, ls(rt), s, t, val);
    else
    {
        sum += Query(left, mid, ls(rt), s, t, val);
        sum += Query(mid + 1, right, rs(rt), s, t, val);
    }
    return sum;
}
```

例题：*https://www.luogu.org/problemnew/show/P3373*

## 5.5 主席树

又称“可持久化 (权值) 线段树”，主要用于查询区间第 k 小 (大) 值。效率高于归并树低于划分树。

```cpp
#include <bits/stdc++.h>

using namespace std;
const int N = 1e5 + 5;
struct SegTreeNode {
    int l, r, m;
    int ls, rs;  // 左儿子、右儿子
    int s;       // 结点总数
} tr[N << 5];
// tcnt表示当前空余的节点编号, rt[i]为时间点为i的线段树的根结点
int tcnt, rt[N];

int n, m, a[N], i2x[N], len;

inline int x2i(int x) {
    return lower_bound(i2x + 1, i2x + 1 + len, x) - i2x;
}

// 区间为[l,r), 返回子树的根结点
int build(int l, int r) {
    int x = tcnt++;
    int mid = (l + r) / 2;
    tr[x].l = l, tr[x].r = r, tr[x].m = mid;
    tr[x].s = 0;
    if (r - l == 1)
        return x;
    tr[x].ls = build(l, mid);
    tr[x].rs = build(mid, r);
    return x;
}

// 在pos处插入数, pre为上一个版本的根结点
int insert(int k, int pre) {
    int cur = tcnt++;
    tr[cur] = tr[pre];
    tr[cur].s++;
    if (tr[cur].r - tr[cur].l == 1)
        return cur;
    if (k < tr[cur].m)
        tr[cur].ls = insert(k, tr[cur].ls);
    else
        tr[cur].rs = insert(k, tr[cur].rs);
    return cur;
}

// 查询区间(x,y]中的第k大值
// tr[x] 和 tr[y] 表示时间点不一样的*两棵*的线段树中的节点
int query(int x, int y, int k) {
    // 当前区间的左子树中数的个数 = y时间的左子树中数的个数 - x时间的左子树中数的个数
    int s = tr[tr[y].ls].s - tr[tr[x].ls].s;
    if (tr[x].r - tr[x].l == 1)
        return tr[x].l;  // 此处返回的l不是下标, 而是一个权值, 是离散化后的权值
    if (k <= s)
        return query(tr[x].ls, tr[y].ls, k);
    else
        return query(tr[x].rs, tr[y].rs, k - s);
}

inline void init() {
    int i;
    // 离散化
    for (i = 1; i <= n; i++)
        i2x[i] = a[i];
```

```
64      sort(i2x + 1, i2x + 1 + n);
65      len = unique(i2x + 1, i2x + 1 + n) - i2x - 1;
66      // 将下标当成时间序列，依次“新建”线段树
67      rt[0] = build(1, len + 1);
68      for (i = 1; i <= n; i++)
69          rt[i] = insert(x2i(a[i]), rt[i - 1]);
70  }
71
72  inline void work() {
73      int i, l, r, k;
74      for (i = 1; i <= m; i++) {
75          scanf("%d%d%d", &l, &r, &k);
76          // 此处的查询，应该查的是两个时间点的两棵线段树，返回的是离散化后的权值，需要i2x恢复
77          printf("%d\n", i2x[query(rt[l - 1], rt[r], k)]);
78      }
79  }
80
81  int main() {
82      int i;
83      scanf("%d%d", &n, &m);
84      for (i = 1; i <= n; i++)
85          scanf("%d", &a[i]);
86      init();
87      work();
88
89      return 0;
90  }
```

## 5.6 带 Lazy 标记的线段树

以下是区间修改 + 区间最大值查询。
若是区间修改 + **区间和查询**，则在 pushdown 时需要将 lazy 标记**乘上区间长度**加到结点上。

```cpp
#include <bits/stdc++.h>

using namespace std;
const int N = 1e5 + 10;

int n;  // 长度

struct Node {
    int l, r, m;
    int mx;   // [l,r)中的最大值
    int tag;  // lazy标记
} t[4 * N];   // 数组大小不要忘记 * 4

// 将lazy标记下推
inline void pushdown(int x) {
    Node& cur = t[x];
    if (cur.r - cur.l == 1)
        return;
    Node &lch = t[x * 2], &rch = t[x * 2 + 1];
    lch.mx += cur.tag, rch.mx += cur.tag;
    lch.tag += cur.tag, rch.tag += cur.tag;
    cur.tag = 0;
}

// 由x的儿子更新x结点，此时应确保x的儿子为最新
inline void pushup(int x) {
    Node& cur = t[x];
    if (cur.r - cur.l == 1)
        return;
    Node &lch = t[x * 2], &rch = t[x * 2 + 1];
    cur.mx = max(lch.mx, rch.mx);
}

// 建树。注意初始时叶子是否为0，若不是，需要pushup
void build(int l, int r, int x) {
    Node& cur = t[x];
    cur.l = l, cur.r = r, cur.m = (l + r) / 2;
    cur.mx = 0, cur.tag = 0;  // 初始化最大值、lazy标记
    if (r - l == 1)
        return;
    build(l, cur.m, x * 2);
    build(cur.m, r, x * 2 + 1);
    pushup(x);  // 若初始值非0，则这句一定要加
}

// [l,r)每个元素加v。注意打标记时应更新被打标记结点的mx，并且应立即pushdown,pushup，过程中也需要pushdown
    ,pushup
void update(int l, int r, int x, int v) {
    Node& cur = t[x];
    if (cur.l == l && cur.r == r) {
        cur.tag += v, cur.mx += v;  // 打标记的时候一定是同时更新max的
        pushdown(x), pushup(x);
        return;
    }
    pushdown(x);
    if (r <= cur.m)
        update(l, r, x * 2, v);
    else if (l >= cur.m)
        update(l, r, x * 2 + 1, v);
    else
        update(l, cur.m, x * 2, v), update(cur.m, r, x * 2 + 1, v);
    pushup(x);
```

```
62  }
63
64  // 查询[l,r)的最大值。过程中注意pushdown
65  int query(int l, int r, int x) {
66      Node& cur = t[x];
67      pushdown(x);
68      if (cur.l == l && cur.r == r)
69          return cur.mx;
70      int mx = 0;
71      if (r <= cur.m)
72          mx = query(l, r, x * 2);
73      else if (l >= cur.m)
74          mx = query(l, r, x * 2 + 1);
75      else
76          mx = max(query(l, cur.m, x * 2), query(cur.m, r, x * 2 + 1));
77      return mx;
78  }
79
80  int main() {
81      ios::sync_with_stdio(0);
82      cin.tie(0);
83
84      int m, i, l, r, v;
85      cin >> n >> m;
86      build(1, n + 1, 1);  // 不要忘记build初始化，[l,r+1)
87      for (i = 1; i <= m; i++) {
88          cin >> v >> l >> r;
89          if (!v)
90              cout << query(l, r + 1, 1) << endl;
91          else
92              update(l, r + 1, 1, 1);
93      }
94
95      return 0;
96  }
```

## 5.7 归并树

### 5.7.1 简介

归并排序，自底向上越来越有序。而归并树则是将归并排序的中间结果保留了下来。
为什么要这么做呢？因为这样当询问区间时，总是可以将询问区间二分成某一次或几次归并排序的中间结果。

### 5.7.2 区间第 k 小值

#### 5.7.2.1 问题简述

给定 n 个数，m 次查询，每次查询 [l,r] 内从小到大第 k 个数，输出这个数。

#### 5.7.2.2 解决方案

对这 n 个数先排序，然后二分尝试，将二分得到的中间值代入归并树中，看是否排名为 k，**对所有排名小于等于 k 的取最大值**。
建树时间复杂度 O(NlogN)，查询时间复杂度 O(logNlogN)。

```
#include <bits/stdc++.h>

using namespace std;
typedef long long ll;
const int N = 1e5 + 10, INF = 1e9 + 8;

int a[N], n, m;

struct Node {
    int l, r, m;
    vector<int> a;
} t[N * 3];

// 由a[]数组初始化归并树
void build(int l, int r, int x) {
    Node& cur = t[x];
    cur.l = l, cur.r = r, cur.m = (l + r) / 2;
    cur.a.clear();
    if (r - l == 1) {
        cur.a.push_back(a[cur.l]);
        return;
    }
    build(l, cur.m, x * 2);
    build(cur.m, r, x * 2 + 1);
    Node &lch = t[x * 2], &rch = t[x * 2 + 1];
    vector<int>::iterator it1, it2;
    it1 = lch.a.begin(), it2 = rch.a.begin();
    // 合并左右子树
    while (it1 != lch.a.end() && it2 != rch.a.end()) {
        if (*it1 <= *it2)
            cur.a.push_back(*it1), it1++;
        else
            cur.a.push_back(*it2), it2++;
    }
    while (it1 != lch.a.end())
        cur.a.push_back(*it1), it1++;
    while (it2 != rch.a.end())
        cur.a.push_back(*it2), it2++;
}

// 查询在区间[l,r)内比v小的数的个数，稍加修改便可查询大于等于k的最小值
int query(int l, int r, int v, int x) {
    Node& cur = t[x];
    if (cur.l == l && cur.r == r)
        return lower_bound(cur.a.begin(), cur.a.end(), v) - cur.a.begin();
    if (r <= cur.m)
        return query(l, r, v, x * 2);
    else if (l >= cur.m)
```

```
49          return query(l, r, v, x * 2 + 1);
50      else
51          return query(l, cur.m, v, x * 2) + query(cur.m, r, v, x * 2 + 1);
52  }
53
54  // 查询区间[l,r)区间内的从小到大第k个值
55  int QR(int ql, int qr, int k) {
56      int l = 1, r = n + 1, m, ans = -INF;
57      while (l < r) {
58          m = (l + r) / 2;
59          int rank = query(ql, qr, a[m], 1) + 1;
60          if (rank <= k)  // 可能有相等的值，所以需要<=
61              ans = max(ans, a[m]);
62          if (rank <= k)
63              l = m + 1;
64          else
65              r = m;
66      }
67      return ans;
68  }
69
70  int main() {
71      int i, l, r, k;
72      while (scanf("%d%d", &n, &m) != EOF) {
73          for (i = 1; i <= n; i++)
74              scanf("%d", &a[i]);
75          build(1, n + 1, 1);
76          sort(a + 1, a + 1 + n);
77          for (i = 1; i <= m; i++) {
78              scanf("%d%d%d", &l, &r, &k);
79              printf("%d\n", QR(l, r + 1, k));
80          }
81      }
82
83      return 0;
84  }
```

#### 5.7.2.3 思考

如何取区间内不重复的第 k 大?
结点中用两个 vector，其中一个是原来的不变，另一个用于存放不重复的数，查询的时候在后者中查。

### 5.7.3 区间内大于等于 v 的最小值

#### 5.7.3.1 问题简述

给定 n 个数，m 次查询，每次查询 [l,r] 内比 v 大的最小值，输出这个最小值。

#### 5.7.3.2 解决方案

归并树天生适合的问题。对于对应区间，直接返回 lower_bound 找到的第一个数注意，如果没找到返回一个无限大，对于其余祖先节点，返回左右子树中找到的最小值。

```
1   int query(int l, int r, int v, int x) {
2       Node& cur = t[x];
3       if (cur.l == l && cur.r == r) {
4           vector<int>::iterator it;
5           it = lower_bound(cur.a.begin(), cur.a.end(), v);
6           if (it == cur.a.end())
7               return n + 1;
8           return *it;
9       }
10      if (r <= cur.m)
11          return query(l, r, v, x * 2);
12      else if (l >= cur.m)
13          return query(l, r, v, x * 2 + 1);
```

```
14      else
15          return min(query(l, cur.m, v, x * 2), query(cur.m, r, v, x * 2 + 1));
16  }
```

## 5.8 划分树

### 5.8.1 简介

划分树是模拟快速排序的。快速排序，自顶向下越来越有序。划分树是在进行快速排序的过程中记录当前子段中每个位置的数是否进入左子树，并用前缀和的思想统计它们。之后再查询时便可依据进入左子树的个数计算第 k 个数是在左子树还是右子树。

### 5.8.2 区间第 k 小值

#### 5.8.2.1 问题简述

给定 n 个数，m 次查询，每次查询 [l,r] 内从小到大第 k 个数，输出这个数。

#### 5.8.2.2 解决方案

```
#include <bits/stdc++.h>

using namespace std;
typedef long long ll;
const int N = 1e5 + 10;

int a[N], sorted[N], n;
int seg[21][N], toleft[21][N];

// 建树
void build(int l, int r, int dep) {
    if (l == r) {
        seg[dep + 1][l] = seg[dep][l];
        return;
    }
    int i, m = (l + r) / 2, lp = l, rp = m + 1, cnt = 0, t = 0;
    // cnt: 该节点内比sorted[m]小的元素个数
    for (i = l; i <= r; i++)
        cnt += seg[dep][i] < sorted[m];

    for (i = l; i <= r; i++) {
        if (i == l)
            toleft[dep][i] = 0;
        else
            toleft[dep][i] = toleft[dep][i - 1];

        if (seg[dep][i] < sorted[m])
            seg[dep + 1][lp++] = seg[dep][i], toleft[dep][i]++;
        else if (seg[dep][i] > sorted[m])
            seg[dep + 1][rp++] = seg[dep][i];
        else {                          // ==
            if (t < m - l + 1 - cnt)  // m-l+1-cnt: 左边最多可以放多少个sorted[m]
                seg[dep + 1][lp++] = seg[dep][i], toleft[dep][i]++, t++;
            else
                seg[dep + 1][rp++] = seg[dep][i];
        }
    }

    build(l, m, dep + 1);
    build(m + 1, r, dep + 1);
}

// 询问区间[l,r]内从小到大的第k个值，当前区间为[sl,sr](初始时为[1,n])，当前深度为dep(初始时为1)
int query(int l, int r, int k, int sl, int sr, int dep) {
    if (r == l)
        return seg[dep][l];
    int m = (sl + sr) / 2;
    int tlsl = (l - 1 >= sl ? toleft[dep][l - 1] : 0), tlsr = toleft[dep][sr];
    int tlr = toleft[dep][r];
    if (k <= tlr - tlsl)
```

```
51          return query(sl + tlsl, m - (tlsr - tlr), k, sl, m, dep + 1);
52      else
53          return query(m + 1 + l - sl - tlsl, sr - (sr - r - (tlsr - tlr)), k - (tlr - tlsl), m +
        1, sr, dep + 1);
54  }
55
56  int main() {
57      int m, i, l, r, k;
58      scanf("%d%d", &n, &m);
59
60      // 以下4行是初始化
61      for (i = 1; i <= n; i++)
62          scanf("%d", &a[i]), sorted[i] = seg[1][i] = a[i];
63      sort(sorted + 1, sorted + 1 + n);
64      build(1, n, 1);
65
66      for (i = 1; i <= m; i++) {
67          scanf("%d%d%d", &l, &r, &k);
68          int ans = query(l, r, k, 1, n, 1);
69          printf("%d\n", ans);
70      }
71
72      return 0;
73  }
```

## 5.9 左偏树

### 5.9.1 模板

```
const int N = 1e3 + 10;
struct Node {
    int k, d, fa, ch[2];  // 键，距离，父亲，左儿子，右儿子
} t[N];

// 取右子树的标号
int& rs(int x) {
    return t[x].ch[t[t[x].ch[1]].d < t[t[x].ch[0]].d];
}

// 用于删除非根节点后向上更新、
// 建议单独用，因为需要修改父节点
void pushup(int x) {
    if (!x)
        return;
    if (t[x].d != t[rs(x)].d + 1) {
        t[x].d = t[rs(x)].d + 1;
        pushup(t[x].fa);
    }
}

// 整个堆加上、减去一个数或乘上一个整数(不改变相对大小)，类似于lazy标记
void pushdown(int x) {
}

// 合并x和y
int merge(int x, int y) {
    // 若一个堆为空，则返回另一个堆
    if (!x || !y)
        return x | y;
    // 取较小的作为根
    if (t[x].k > t[y].k)
        swap(x, y);
    // 下传标记，这么写的条件是必须保证堆顶元素时刻都是最新的
    pushdown(x);
    // 递归合并右儿子和另一个堆 // 若不满足左偏树性质则交换两儿子 // 更新右子树的父亲，只有右子树有父亲
    t[rs(x) = merge(rs(x), y)].fa = x;
    // 更新dist
    t[x].d = t[rs(x)].d + 1;
    return x;
}
```

### 5.9.2 模板题 P3377 【模板】左偏树（可并堆）

#### 5.9.2.1 题目描述

如题，一开始有 N 个小根堆，每个堆包含且仅包含一个数。接下来需要支持两种操作：
操作 1：1 x y 将第 x 个数和第 y 个数所在的小根堆合并（若第 x 或第 y 个数已经被删除或第 x 和第 y 个数在用一个堆内，则无视此操作）
操作 2：2 x 输出第 x 个数所在的堆最小数，并将其删除（若第 x 个数已经被删除，则输出-1 并无视删除操作）。
当堆里有多个最小值时，优先删除原序列的靠前的。

#### 5.9.2.2 涉及知识点

1. 左偏树的基本操作（合并、删除）
**2. 并查集查询结点所在的堆的根**

需要注意的是：
合并前要检查是否已经在同一堆中。
左偏树的深度可能达到 $O(n)$，因此找一个点所在的堆顶要用并查集维护，不能直接暴力跳父亲。（虽然很多题数据水，暴力跳父亲可以过……）（用并查集维护根时要保证原根指向新根，新根指向自己。）

#### 5.9.2.3 代码

```cpp
#include <bits/stdc++.h>

using namespace std;

const int N = 1e5 + 10;

bitset<N> f;  // 用于标记某个元素是否被删除

// 关键字
struct Key {
    int a, b;
    bool operator<(const Key& rhs) const {
        if (a != rhs.a)
            return a < rhs.a;
        return b < rhs.b;
    }
};

// 左偏树节点
struct Node {
    Key k;
    int d, fa, ch[2];
} t[N];

int& rs(int x) {
    return t[x].ch[t[t[x].ch[1]].d < t[t[x].ch[0]].d];
}

int merge(int x, int y) {
    if (!x || !y)
        return x | y;
    if (t[y].k < t[x].k)
        swap(x, y);
    t[rs(x) = merge(rs(x), y)].fa = x;
    t[x].d = t[rs(x)].d + 1;
    return x;
}

struct UF {
    int fa, t;
} uf[N];  // 并查集

int find(int x) {
    if (x == uf[x].fa)
        return x;
    return uf[x].fa = find(uf[x].fa);
}

void ufUnion(int x, int y) {
    x = find(x), y = find(y);
    uf[y].fa = x;
}

int main() {
    ios::sync_with_stdio(0);
    cin.tie(0);

    int n, m, i, x, y;
    cin >> n >> m;
    for (i = 1; i <= n; i++) {
        cin >> x;
        uf[i].fa = i, uf[i].t = i;
        t[i].d = 1, t[i].k = Key({x, i});
    }
```

```
65      for (i = 1; i <= m; i++) {
66          cin >> x;
67          if (x == 1) {
68              cin >> x >> y;
69              if (f.test(x) || f.test(y) || uf[find(x)].t == uf[find(y)].t)
70                  continue;
71              int root = merge(uf[find(x)].t, uf[find(y)].t);
72              ufUnion(x, y);
73              uf[find(x)].t = root;
74          } else {
75              cin >> x;
76              if (f.test(x)) {
77                  cout << -1 << endl;
78                  continue;
79              }
80              cout << t[uf[find(x)].t].k.a << endl;
81              f[t[uf[find(x)].t].k.b] = 1;
82              uf[find(x)].t = merge(t[uf[find(x)].t].ch[0], t[uf[find(x)].t].ch[1]);
83          }
84      }
85
86      return 0;
87  }
```

### 5.9.3 洛谷 P1552 [APIO2012] 派遣

#### 5.9.3.1 题目描述

题目太长，简述。你有预算 m 元，给定 n 个人，具有上下级关系，构成一棵树，每个人有两个参数–花费、领导力。让你选择先选一个点作为领导者，然后在以领导者为根的子树中任意选择一些点 (要求花费不超过 m)，得到的价值为领导者的领导力 * 选定的人数。问最大价值为多少?

#### 5.9.3.2 涉及知识点

树上问题
可并堆的合并
**用堆维护背包**

#### 5.9.3.3 思路

大根堆中存储每个点的花费。递归地，对于一个点 x，我们合并 x 的所有儿子节点的堆，并计算其总和，如果总和大于 m，不断弹出堆顶元素并更新总和，直到总和小于等于 m。有点像带反悔的贪心。

### 5.9.4 洛谷 P3261 [JLOI2015] 城池攻占

#### 5.9.4.1 题目描述

你要用 m 个骑士攻占 n 个城池。n 个城池 (1 到 n) 构成了一棵有根树，1 号城池为根，其余城池父节点为 fi。m 个骑士 (1 到 m)，其中第 i 个骑士的初始战斗力为 si，第一个攻击的城池为 ci。
每个城池有一个防御值 hi，如果一个骑士的战斗力大于等于城池的生命值，那么骑士就可以占领这座城池；否则占领失败，骑士将在这座城池牺牲。占领一个城池以后，骑士的战斗力将发生变化，然后继续攻击管辖这座城池的城池，直到占领 1 号城池，或牺牲为止。
除 1 号城池外，每个城池 i 会给出一个战斗力变化参数 ai;vi。若 ai =0，攻占城池 i 以后骑士战斗力会增加 vi；若 ai =1，攻占城池 i 以后，战斗力会乘以 vi。注意每个骑士是单独计算的。也就是说一个骑士攻击一座城池，不管结果如何，均不会影响其他骑士攻击这座城池的结果。
现在的问题是，对于每个城池，输出有多少个骑士在这里牺牲；对于每个骑士，输出他攻占的城池数量。

#### 5.9.4.2 涉及知识点

树上问题
可并堆的合并
**堆的整体操作 (打标记，pushdown)**

#### 5.9.4.3 注意

**打标记之后，应该立即更新被打标记的点，然后 pushdown，pushdown 总是由父节点发起帮助儿子提前更新。这样，如果某个点有标记，则表示该点本身是最新的，但他应该为儿子更新。也即上一层的标记是为下一层准备的。**

#### 5.9.4.4 打标记、下传代码

```
// x被打标记，立即更新x的值，并且将标记存放在此(准备之后让pushdown给下一层更新)
inline void mark(ll x, ll a, ll b) {
    if (!x)
        return;
    t[x].k.s = t[x].k.s * b + a;            // 更新自身
    t[x].a *= b, t[x].b *= b, t[x].a += a;  // 寄存标记
}

// 下传标记，本质上就是再给儿子们mark()一下，然后清空自身标记
inline void pushdown(ll x) {
    if (!x)
        return;
    mark(t[x].ch[0], t[x].a, t[x].b);
    mark(t[x].ch[1], t[x].a, t[x].b);
    t[x].a = 0, t[x].b = 1;
}
```

### 5.9.5 洛谷 P3273 [SCOI2011] 棘手的操作

#### 5.9.5.1 题目描述

有 N 个节点，标号从 1 到 N，这 N 个节点一开始相互不连通。第 i 个节点的初始权值为 a[i]，接下来有如下一些操作:
U x y: 加一条边，连接第 x 个节点和第 y 个节点
A1 x v: 将第 x 个节点的权值增加 v
A2 x v: 将第 x 个节点所在的连通块的所有节点的权值都增加 v
A3 v: 将所有节点的权值都增加 v
F1 x: 输出第 x 个节点当前的权值
F2 x: 输出第 x 个节点所在的连通块中，权值最大的节点的权值
F3: 输出所有节点中，权值最大的节点的权值

#### 5.9.5.2 涉及知识点

左偏树
并查集
multiset
整体标记
**启发式合并**

#### 5.9.5.3 思路

这题题如其名，非常棘手。
首先，找一个节点所在堆的堆顶要用并查集，而不能暴力向上跳。
再考虑单点查询，若用普通的方法打标记，就得查询点到根路径上的标记之和，最坏情况下可以达到的复杂度。
如果只有堆顶有标记，就可以快速地查询了，但如何做到呢?
**可以用类似启发式合并的方式，每次合并的时候把较小的那个堆标记暴力下传到每个节点，然后把较大的堆的标记作为合并后的堆的标记。由于合并后有另一个堆的标记，所以较小的堆下传标记时要下传其标记减去另一个堆的标记。由于每个节点每被合并一次所在堆的大小至少乘二，所以每个节点最多被下放次标记，暴力下放标记的总复杂度就是 $O(n)$。**
再考虑单点加，先删除，再更新，最后插入即可。
然后是全局最大值，可以用一个平衡树/支持删除任意节点的堆（如左偏树）/multiset 来维护每个堆的堆顶。

所以，每个操作分别如下:
1. 暴力下传点数较小的堆的标记，合并两个堆，更新 size、tag，在 multiset 中删去合并后不在堆顶的那个原堆

顶。
2. 删除节点，更新值，插入回来，更新 multiset。需要分删除节点是否为根来讨论一下。
3. 堆顶打标记，更新 multiset。
4. 打全局标记。
5. 查询值 + 堆顶标记 + 全局标记。
6. 查询根的值 + 堆顶标记 + 全局标记。
7. 查询 multiset 最大值 + 全局标记。

### 5.9.6 洛谷 P4331 Sequence 数字序列

#### 5.9.6.1 题目描述

这是一道论文题
给定一个整数序列 $a_1, a_2, ..., a_n$，求出一个递增序列 $b_1 < b_2 < ... < b_n$，使得序列 $a_i$ 和 $b_i$ 的各项之差的绝对值之和 $|a_1 - b_1| + |a_2 - b_2| + ... + |a_n - b_n|$ 最小。

#### 5.9.6.2 涉及知识点

堆的合并 (因为没有整体标记，所以这里可以用 ___gnu_pbds::priority_queue)
**堆维护区间中位数**
**递增序列转非递减序列（减下标法）**

#### 5.9.6.3 思路

递增序列转非递减序列：把 a[i] 减去 i，易知 b[i] 也减去 i 后答案不变，本来 b 要求是递增序列，这样就转化成了不下降序列，方便操作。
堆维护区间中位数：大根堆，每次合并之后，如果堆内元素个数大于区间的一半，则一直 pop 直到等于一半，堆顶元素即为中位数。(这么做的前提是中位数大的区间内的最小值小于等于另一个区间内仅比那个区间中位数大的数)

## 5.10 线段树练习

### 5.10.1 区间最大连续子段和

```cpp
#include <bits/stdc++.h>

using namespace std;
const int N = 5e4 + 10;  // 数组大小，记得改

struct Node {
    int l, r, m;
    int s, f, fl, fr;  // 区间的和，区间最大子段和，包含左端点的最大子段和，包含右端点的最大子段和
} s[N * 4];

// 构建线段树
void build(int l, int r, int i) {
    Node& fa = s[i];
    fa.l = l, fa.r = r, fa.m = (l + r) / 2;
    fa.s = fa.f = fa.fl = fa.fr = 0;
    if (r - l == 1)
        return;
    build(l, fa.m, i * 2);
    build(fa.m, r, i * 2 + 1);
}

// 自底向上更新
void pushup(int i) {
    Node &fa = s[i], &lson = s[i * 2], &rson = s[i * 2 + 1];  // 父亲 左儿子 右儿子
    fa.s = lson.s + rson.s;
    fa.f = max(max(lson.f, rson.f), lson.fr + rson.fl);
    fa.fl = max(lson.fl, lson.s + rson.fl);
    fa.fr = max(rson.fr, rson.s + lson.fr);
}

// 单点更新
void update(int x, int p, int i) {
    Node& fa = s[i];
    if (fa.l == p && fa.r - fa.l == 1) {
        fa.s = fa.f = fa.fl = fa.fr = x;
        return;
    }
    if (p < fa.m)
        update(x, p, i * 2);
    else
        update(x, p, i * 2 + 1);
    // 向上更新
    pushup(i);
}

// 作为查询的返回值
struct Ret {
    int f, s, fl, fr;
};

// 查询
Ret query(int l, int r, int i) {
    Node& fa = s[i];
    if (fa.l == l && fa.r == r)
        return {fa.f, fa.s, fa.fl, fa.fr};
    if (r <= fa.m)
        return query(l, r, i * 2);
    else if (l >= fa.m)
        return query(l, r, i * 2 + 1);
    else {
        Ret lret = query(l, fa.m, i * 2);
        Ret rret = query(fa.m, r, i * 2 + 1);
```

```
63          return {max(max(lret.f, rret.f), lret.fr + rret.fl), lret.s + rret.s, max(lret.fl, lret.s
        + rret.fl), max(rret.fr, rret.s + lret.fr)};
64      }
65  }
66
67  int main() {
68      ios::sync_with_stdio(0);
69      cin.tie(0);
70
71      int n, m, i, x, l, r;
72      cin >> n;
73      build(1, n + 1, 1);  // 不要忘了初始化
74      for (i = 1; i <= n; i++) {
75          cin >> x;
76          update(x, i, 1);  // 修改元素
77      }
78      cin >> m;  // 询问
79      while (m--) {
80          cin >> l >> r;
81          cout << query(l, r + 1, 1).f << endl;  // 询问[l,r]，实际查询[l,r+1)，统一用左闭右开区间
82      }
83
84      return 0;
85  }
```

### 5.10.2 最长单峰序列的下标的最小字典序和最大字典序

简化问题，只考虑最小字典序。那么与 LIS 类似，能选就选就可以得到最小字典序。

选哪些？**被包含在最长单峰序列中的元素**

如何知道是否被包含？
类比 LIS，如果 a[i] 被包含，那么**最长长度-a[i] 之前选的个数 = 以 a[i] 为开头的最长长度**。
然后考虑两种情况：1.a[i] 在峰的左侧，已选个数 + 以 a[i] 为开头的单峰序列长度；2.a[i] 为峰或在峰的右侧，已选个数 + 以 a[i] 为开头的递减序列长度

于是我们就需要维护两个量：
1. **以 a[i] 为开头的单峰序列长度**
2. **以 a[i] 为开头的递减序列长度**

递减的就直接 LIS 很简单。对于单峰：
倒过来维护，设以 a[i] 为开头的单峰序列最长长度为 peak[i]，
**peak[i]=max(以比 a[i] 大的数为开头的单峰最大长度 +1, 以 a[i] 为开头的递减序列最大长度)**
而维护以比 a[i] 大的数为开头的单峰最大长度可以用**权值线段树**

```
1   /*
2   求 最长 单峰 序列 的 下标 的 最小字典序 和 最大字典序
3
4   首先用LIS求出以a[i]开头的严格递减序列的最长长度，
5   再用线段树求以a[i]开头的单峰序列的最长长度(倒着求)
6   求最小字典序即要求：
7   1. a[i] > pre && L - cnt == peak[i]
8   2. a[i] < pre && L - cnt == d[i]
9   */
10
11  #include <bits/stdc++.h>
12
13  using namespace std;
14  const int N = 3e5 + 10;
15  int a[N], n;
16  int i2x[N];   // 离散化
17  int m;        // 离散化后的长度
18  int c[N];     // LIS的dp数组
19  int d[N];     // d[i]:以a[i]开头递减序列最长长度
20  int peak[N];  // peak[i]:以a[i]开头的单峰序列最长长度
```

```
int ans[N];

inline int x2i(int x) {
    return lower_bound(i2x + 1, i2x + 1 + m, x) - i2x;
}

// 权值线段树，维护
struct Node {
    int l, r, m;
    int mx;  // [l,r)中的最大值
} t[4 * N];  // 数组大小不要忘记 * 4

inline void pushup(int x) {
    Node& cur = t[x];
    if (cur.r - cur.l == 1)
        return;
    Node &lch = t[x * 2], &rch = t[x * 2 + 1];
    cur.mx = max(lch.mx, rch.mx);
}

void build(int l, int r, int x) {
    Node& cur = t[x];
    cur.l = l, cur.r = r, cur.m = (l + r) / 2;
    cur.mx = 0;
    if (r - l == 1)
        return;
    build(l, cur.m, x * 2);
    build(cur.m, r, x * 2 + 1);
    pushup(x);
}

void update(int l, int r, int x, int v) {
    Node& cur = t[x];
    if (cur.l == l && cur.r == r) {
        cur.mx = v;
        return;
    }
    if (r <= cur.m)
        update(l, r, x * 2, v);
    else if (l >= cur.m)
        update(l, r, x * 2 + 1, v);
    else
        update(l, cur.m, x * 2, v), update(cur.m, r, x * 2 + 1, v);
    pushup(x);
}

int query(int l, int r, int x) {
    Node& cur = t[x];
    if (cur.l == l && cur.r == r)
        return cur.mx;
    int mx = 0;
    if (r <= cur.m)
        mx = query(l, r, x * 2);
    else if (l >= cur.m)
        mx = query(l, r, x * 2 + 1);
    else
        mx = max(query(l, cur.m, x * 2), query(cur.m, r, x * 2 + 1));
    return mx;
}

inline void LIS() {
    int i, j, mx = 0;
    for (i = 1; i <= n; i++) {
        j = lower_bound(c + 1, c + 1 + mx, a[i]) - c;
        d[i] = j, c[j] = a[i], mx = max(mx, j);
    }
```

```
87  }
88
89  inline void work() {
90      int i;
91
92      // 求以a[i]为开头的最长下降子序列的长度
93      reverse(a + 1, a + 1 + n);
94      LIS();
95      reverse(a + 1, a + 1 + n);
96      reverse(d + 1, d + 1 + n);
97
98      // 通过权值线段树维护以a[i]为开头的单峰序列的最大长度
99      build(1, m + 1, 1);
100     for (i = n; i >= 1; i--) {
101         int x = 0;
102         if (a[i] + 1 < m + 1)
103             x = query(a[i] + 1, m + 1, 1);   // 查询以比a[i]大的数为开头的单峰序列的最大长度
104         peak[i] = max(x + 1, d[i]);          // a[i]可以在峰的左侧(原单峰加上a[i]), 也可以就是峰或峰
    的右侧(单调减)
105         update(a[i], a[i] + 1, 1, peak[i]);  // 将以a[i]为开头的单峰添加进权值线段树
106     }
107 }
108
109 inline int getAns() {
110     int L = query(1, m + 1, 1);  // 单峰最大长度
111     int cnt = 0, pre = 0, i;     // cnt是已经求得的个数, pre是上一个的值
112     // 求单峰的上升部分
113     for (i = 1; i <= n; i++) {
114         // 如果a[i]被包含在最长里
115         if (a[i] > pre && L - cnt == peak[i])
116             ans[++cnt] = i, pre = a[i];
117         // 如果可以转为下降, 就停止上升
118         else if (a[i] < pre && L - cnt == d[i])
119             break;
120     }
121     // 单峰的下降部分
122     for (; i <= n; i++)
123         if (a[i] < pre && L - cnt == d[i])
124             ans[++cnt] = i, pre = a[i];
125     return cnt;
126 }
127
128 int main() {
129     int i;
130     while (scanf("%d", &n) != EOF) {
131         for (i = 1; i <= n; i++) {
132             scanf("%d", &a[i]);
133             i2x[i] = a[i];
134             ans[i] = c[i] = d[i] = peak[i] = 0;
135         }
136
137         sort(i2x + 1, i2x + 1 + n);
138         m = unique(i2x + 1, i2x + 1 + n) - i2x - 1;
139         for (i = 1; i <= n; i++)
140             a[i] = x2i(a[i]);
141
142         work();
143         int cnt = getAns();
144         for (i = 1; i <= cnt; i++) {
145             if (i > 1)
146                 printf(" ");
147             printf("%d", ans[i]);
148         }
149         printf("\n");
150
151         for (i = 1; i <= n; i++)
```

```
            ans[i] = c[i] = d[i] = peak[i] = 0;

        reverse(a + 1, a + 1 + n);
        work();
        cnt = getAns();
        for (i = cnt; i >= 1; i--) {
            if (i < cnt)
                printf(" ");
            printf("%d", n - ans[i] + 1);
        }
        printf("\n");
    }

    return 0;
}
```

### 5.10.3 HDU6602 Longest Subarray

#### 5.10.3.1 题目描述

给你一个数组，数的范围是 $[1, C]$，给定 $K$，让你找一个最长的区间使得区间内任意一个出现的数在该区间内的数量都大于 $K$。

#### 5.10.3.2 解决方案

主要思路：**尺取法，枚举右端点，用线段树维护左端点的可行区间**
本题求一个最大的子区间，满足区间内的数字要么出现次数大于等于 k 次，要么没出现过。给定区间内的数字范围是 1 到 c。
如果 r 为右边界，对于一种数字 x，满足条件的左边界 l 的范围是 r 左边第一个 x 出现的位置 +1(即这段区间内没有出现过 x，如果 x 在 1 到 r 内都没有出现过，那么 1 到 r 自然都是 l 的合法范围)，以及 1 到从右往左数数第 k 个 x 出现的位置 (即这段区间内的 x 出现次数大于等于 k)。所以我们只要找到同时是 c 种数字的合法左边界的位置中最小的，然后枚举所有的 i 作为右边界即可得出答案。
由此，我们可以令线段树的叶子节点中的值表示可以作为多少种数字的左端点。然后对每一次查询，我们都尽量向左查最大值为 c 的节点，返回下标。

```
/*
尺取法
移动右端点，用线段树维护左端点的可行区间

思维；线段树区间修改，区间最大值，lazy标记
*/

#include <bits/stdc++.h>

using namespace std;
const int N = 2e5 + 10;

int n;  // 长度
int c, k, a[N];

struct Node {
    int l, r, m;
    int mx;   // [l,r)中的最大值
    int tag;  // lazy标记
} t[4 * N];   // 数组大小不要忘记 * 4

inline void pushdown(int x) {
    Node& cur = t[x];
    if (cur.r - cur.l == 1)
        return;
    Node &lch = t[x * 2], &rch = t[x * 2 + 1];
    lch.mx += cur.tag, rch.mx += cur.tag;
    lch.tag += cur.tag, rch.tag += cur.tag;
    cur.tag = 0;
}
```

```
31
32  inline void pushup(int x) {
33      Node& cur = t[x];
34      if (cur.r - cur.l == 1)
35          return;
36      Node &lch = t[x * 2], &rch = t[x * 2 + 1];
37      cur.mx = max(lch.mx, rch.mx);
38  }
39
40  void build(int l, int r, int x) {
41      Node& cur = t[x];
42      cur.l = l, cur.r = r, cur.m = (l + r) / 2;
43      cur.mx = 0, cur.tag = 0;  // 初始化最大值、lazy标记
44      if (r - l == 1)
45          return;
46      build(l, cur.m, x * 2);
47      build(cur.m, r, x * 2 + 1);
48      pushup(x);  // 若初始值非0, 则这句一定要加
49  }
50
51  void update(int l, int r, int x, int v) {
52      Node& cur = t[x];
53      if (cur.l == l && cur.r == r) {
54          cur.tag += v, cur.mx += v;  // 打标记的时候一定是同时更新max的
55          return;
56      }
57      pushdown(x);
58      if (r <= cur.m)
59          update(l, r, x * 2, v);
60      else if (l >= cur.m)
61          update(l, r, x * 2 + 1, v);
62      else
63          update(l, cur.m, x * 2, v), update(cur.m, r, x * 2 + 1, v);
64      pushup(x);
65  }
66
67  int query(int l, int r, int x) {
68      Node& cur = t[x];
69      pushdown(x);
70      if (cur.r - cur.l == 1)
71          return cur.l;
72      Node &lch = t[x * 2], &rch = t[x * 2 + 1];
73      int p = 0;
74      if (r <= cur.m)
75          p = query(l, r, x * 2);
76      else if (l >= cur.m)
77          p = query(l, r, x * 2 + 1);
78      else {
79          if (lch.mx == c)
80              p = query(l, cur.m, x * 2);
81          else
82              p = query(cur.m, r, x * 2 + 1);
83      }
84
85      return p;
86  }
87
88  vector<int> pos[N];  // pos[x]是x出现的所有位置
89
90  int main() {
91      int i, l, r, sz, ans = 0;
92
93      while (scanf("%d%d%d", &n, &c, &k) != EOF) {
94          for (i = 1; i <= n; i++)
95              scanf("%d", &a[i]);
96          if (n == 0) {
```

```
97              printf("0\n");
98              continue;
99          }
100         if (k == 0 || k == 1) {
101             printf("%d\n", n);
102             continue;
103         }
104         ans = 0;
105         for (i = 1; i <= c; i++)
106             pos[i].clear();
107
108         build(1, n + 1, 1);
109
110         // 枚举右端点
111         for (i = 1; i <= n; i++) {
112             // 对于[i,i]，它可以作为除去它本身以外的c-1个数的左端点。所以位置i要加上c-1
113             update(i, i + 1, 1, c - 1);
114             // 关于a[i]，在它上一次出现的位置和当前位置i之间的位置，在转移来i之前是可以当做a[i]的左端点
    的(即a[i]出现0次)
115             // 但是现在右端点为a[i]，那么这些位置就不可再作为a[i]的左端点了(除非k<=1，这个作为特殊情况考
    虑)，所以这一段要减1
116             sz = pos[a[i]].size();
117             if (sz) {
118                 l = *pos[a[i]].rbegin();
119                 if (l + 1 < i)
120                     update(l + 1, i - 1 + 1, 1, -1);
121             } else {
122                 if (1 < i)
123                     update(1, i, 1, -1);
124             }
125             // 当前a[i]还可以使之前已经出现k-1次的一段由不可以作为a[i]的左端点变为可以作为a[i]的左端点。
    (如果有的话)
126             if (sz >= k - 1) {
127                 r = pos[a[i]][sz - k + 1], l = 1;
128                 if (sz >= k)
129                     l = pos[a[i]][sz - k] + 1;
130                 if (l <= r)
131                     update(l, r + 1, 1, 1);
132             }
133             pos[a[i]].push_back(i);
134             // 最后查询能被c个数作为左端点的最左位置
135             if (t[1].mx == c)
136                 ans = max(ans, i - query(1, i + 1, 1) + 1);
137         }
138         cout << ans << endl;
139     }
140
141     return 0;
142 }
```

## 5.11 树状数组套权值线段树

树状数组套权值线段树通常用来解决一种二维查询，第一维是区间，第二维是值。
最典型的例子就是带修改的区间 k 小值查询，思路是，把二分答案的操作和查询小于一个值的数的数量两种操作结合起来。
在修改操作进行时，先在线段树上从上往下跳到被修改的点，删除所经过的点所指向的动态开点权值线段树上的原来的值，然后插入新的值。
在查询答案时，先取出该区间覆盖在线段树上的所有点，然后用类似于静态区间 k 小值的方法，将这些点一起向左儿子或向右儿子跳。如果所有这些点左儿子存储的值大于等于 k，则往左跳，否则往右跳。

```
/*
例题：洛谷 P2617 Dynamic Rankings
线段树套动态开点权值线段树
*/

#include <bits/stdc++.h>

using namespace std;
const int N = 2e5 + 10;  // FIXME:

struct Qr {
    char op[3];   // 哪种操作
    int l, r, k;  // 查询
    int p, v;     // 修改
} qr[N];

int n, m, a[N];    // 元素个数、操作个数、原数组
int q1[N], q2[N];  // 待查区间左右端点的前缀和，数组元素表示组成前缀和的权值线段树的根结点编号
int cnt1, cnt2;    // 两个前缀和被划分成的节点个数

// 离散化
int i2x[N], len;
inline int x2i(int x) {
    return lower_bound(i2x + 1, i2x + 1 + len, x) - i2x;
}

struct SegT {
    int cnt;                        // 已经使用的结点个数
    int rt[N * 2];                  // rt[i]：树状数组中下标为i的节点对应的权值线段树的根结点编号
    int lc[N * 300], rc[N * 300];  // 左右儿子编号
    int s[N * 300];                 // 结点中值的个数

    // o是当前结点的编号，[l,r]表示当前结点所表示的区间，v是要添加或删除的权值，d表示添加或删除
    void update(int& o, int l, int r, int v, int d) {
        if (!o)
            o = ++cnt;
        s[o] += d;
        if (l == r)
            return;
        int m = (l + r) / 2;
        if (v <= m)
            update(lc[o], l, m, v, d);
        else
            update(rc[o], m + 1, r, v, d);
    }
} st;

inline int lowbit(int x) {
    return -x & x;
}

// p是要修改的位置，v是要修改的权值，d=-1或1表示删除或添加
void upd(int p, int v, int d) {
    for (; p <= n; p += lowbit(p))
        st.update(st.rt[p], 1, len, v, d);
}
```

```
57
58  // 获取构成区间[1,p]的所有权值线段树的根结点，放到a[]中，共cnt个
59  void gtv(int p, int a[], int& cnt) {
60      cnt = 0;
61      for (; p; p -= lowbit(p))
62          a[++cnt] = st.rt[p];
63  }
64
65  // [l,r]表示当前两组结点表示的值域，查询第k小值，返回的是离散化之后的值
66  int qry(int l, int r, int k) {
67      if (l == r)
68          return l;
69
70      int m = (l + r) / 2, ltot = 0, i;
71
72      // 统计左子树中权值个数，前缀和之差
73      for (i = 1; i <= cnt2; i++)
74          ltot += st.s[st.lc[q2[i]]];
75      for (i = 1; i <= cnt1; i++)
76          ltot -= st.s[st.lc[q1[i]]];
77
78      if (ltot >= k) {  // 向左搜
79          for (i = 1; i <= cnt2; i++)
80              q2[i] = st.lc[q2[i]];
81          for (i = 1; i <= cnt1; i++)
82              q1[i] = st.lc[q1[i]];
83          return qry(l, m, k);
84      } else {  // 向右搜
85          for (i = 1; i <= cnt2; i++)
86              q2[i] = st.rc[q2[i]];
87          for (i = 1; i <= cnt1; i++)
88              q1[i] = st.rc[q1[i]];
89          return qry(m + 1, r, k - ltot);
90      }
91  }
92
93  int main() {
94      int i;
95
96      scanf("%d%d", &n, &m);
97      for (i = 1; i <= n; i++)
98          scanf("%d", &a[i]), i2x[++len] = a[i];
99
100     for (i = 1; i <= m; i++) {
101         scanf("%s", qr[i].op);
102         if (qr[i].op[0] == 'Q')
103             scanf("%d%d%d", &qr[i].l, &qr[i].r, &qr[i].k);
104         else
105             scanf("%d%d", &qr[i].p, &qr[i].v), i2x[++len] = qr[i].v;
106     }
107
108     // 离散化
109     sort(i2x + 1, i2x + 1 + len);
110     len = unique(i2x + 1, i2x + 1 + len) - i2x - 1;
111
112     // 初始化树状数组
113     for (i = 1; i <= n; i++)
114         upd(i, x2i(a[i]), 1);
115
116     // 操作
117     for (i = 1; i <= m; i++) {
118         if (qr[i].op[0] == 'C') {                // 修改
119             upd(qr[i].p, x2i(a[qr[i].p]), -1);  // 删去旧值
120             upd(qr[i].p, x2i(qr[i].v), 1);      // 添加新值
121             a[qr[i].p] = qr[i].v;
122         } else {  // 查询
```

```
123             gtv(qr[i].l - 1, q1, cnt1);
124             gtv(qr[i].r, q2, cnt2);
125             printf("%d\n", i2x[qry(1, len, qr[i].k)]);
126         }
127     }
128
129     return 0;
130 }
```

# Chapter 6

# 图论

# 6.1 最短路

## 6.1.1 单源最短路径

### 6.1.1.1 Dijkstra

```
void Dijkstra()
{
    memset(dist, 0x3f, sizeof(dist));
    memset(vis, 0, sizeof(vis));
    priority_queue<pii, vector<pii>, greater<pii>> q;
    dist[1] = 0;
    q.push({dist[1], 1});
    while (!q.empty())
    {
        int x = q.top().second;
        q.pop();
        if (!vis[x])
        {
            vis[x] = 1;
            for (auto it : v[x])
            {
                int y = it.first;
                if (dist[y] > dist[x] + it.second)
                {
                    dist[y] = dist[x] + it.second;
                    q.push({dist[y], y});
                }
            }
        }
    }
}
```

### 6.1.1.2 Bellman-Ford 和 SPFA

```
void SPFA(int S)
{
    memset(dis, 0x3f, sizeof(dis));
    memset(vis, 0, sizeof(vis));
    queue<int> q;
    dis[S] = 0;
    vis[S] = 1;
    q.push(S);
    while (!q.empty())
    {
        int x = q.front();
        q.pop();
        vis[x] = 0;
        for (int i = head[x]; i; i = nxt[i])
        {
            int y = ver[i], z = edge[i];
            if (dis[y] > dis[x] + z)
            {
                dis[y] = dis[x] + z;
                if (!vis[y])
                    q.push(y), vis[y] = 1;
            }
        }
    }
}
```

例题分析

POJ3662 Telephone Lines（分层图最短路/二分答案，双端队列 BFS）
P1073 最优贸易（原图与反图，枚举节点）
P3008 [USACO11JAN] 道路和飞机 Roads and Planes（DAG，拓扑序，连通块）

### 6.1.2 任意两点间最短路径

#### 6.1.2.1 Floyd

```
void get_path(int i, int j)
{
    if (!path[i][j])
        return;
    get_path(i, path[i][j]);
    p.push_back(path[i][j]);
    get_path(path[i][j], j);
}
void Floyd()
{
    memcpy(d, a, sizeof(d));
    for (int k = 1; k <= n; k++)
    {
        for (int i = 1; i < k; i++)
        {
            for (int j = i + 1; j < k; j++)
            {
                //注意溢出
                ll temp = d[i][j] + a[i][k] + a[k][j];
                if (ans > temp)
                {
                    ans = temp;
                    p.clear();
                    p.push_back(i);
                    get_path(i, j);
                    p.push_back(j);
                    p.push_back(k);
                }
            }
        }
        for (int i = 1; i <= n; i++)
        {
            for (int j = 1; j <= n; j++)
            {
                ll temp = d[i][k] + d[k][j];
                if (d[i][j] > temp)
                {
                    d[i][j] = temp;
                    path[i][j] = k;
                }
            }
        }
    }
}
```

例题分析

POJ1094 Sorting It All Out（传递闭包）
POJ1734 Sightseeing trip（无向图最小环）
POJ3613 Cow Relays（离散化，广义矩阵乘法，快速幂）

## 6.2 最小生成树

### 6.2.1 Kruskal

基于并查集

```
void Init()
{
    for (int i = 1; i <= n; i++)
        fa[i] = i;
}
int Find(int x)
{
    if (x == fa[x])
        return x;
    return fa[x] = Find(fa[x]);
}
void Kruskal()
{
    Init();
    sort(e.begin(), e.end());
    int ans=0;
    for (int i = 0; i < e.size(); i++)
    {
        int u = e[i].u, v = e[i].v;
        int fu = Find(u), fv = Find(v);
        if (fu != fv)
        {
            fa[fu] = fv;
            ans += e[i].w;
        }
    }
}
```

### 6.2.2 Prim

```
void Prim()
{
    memset(vis, 0, sizeof(vis));
    memset(d, 0x3f, sizeof(d));
    d[1] = 0;
    int temp = n;
    int ret = 0;
    while (temp--)
    {
        int min_pos = 0;
        for (int i = 1; i <= n; i++)
            if (!vis[i] && (!min_pos || d[i] < d[min_pos]))
                min_pos = i;
        if (min_pos)
        {
            vis[min_pos] = 1;
            ret += d[min_pos];
            for (int i = 1; i <= n; i++)
                if (!vis[i]) d[i] = min(d[i], weight[min_pos][i]);
        }
    }
}
```

例题分析

走廊泼水节（Kruskal，最小生成树扩充为完全图）

POJ1639 Picnic Planning（度限制最小生成树，连通块，树形 DP）

```
#include <algorithm>
#include <cstring>
#include <iostream>
```

```
4  #include <map>
5  #include <string>
6  #include <vector>
7  using namespace std;
8  #define inf 0x3f3f3f3f
9  #define N 25
10 #define M 500
11 map<string, int> name;
12 struct edge
13 {
14     int u, v, w;
15     bool operator<(const edge &e) const
16     {
17         return w < e.w;
18     }
19 };
20 int n, s, ptot = 0, a[N][N], ans, fa[N], d[N], ver[N];
21 vector<edge> e;
22 bool vis[N][N];
23 edge dp[N]; //dp[i]  1...i路径上的最大边
24 void Init()
25 {
26     for (int i = 1; i <= ptot; i++)
27         fa[i] = i;
28 }
29 int Find(int x)
30 {
31     if (x == fa[x])
32         return x;
33     return fa[x] = Find(fa[x]);
34 }
35 void Kruskal()
36 {
37     Init();
38     sort(e.begin(), e.end());
39     for (int i = 0; i < e.size(); i++)
40     {
41         int u = e[i].u, v = e[i].v;
42         if (u != 1 && v != 1)
43         {
44             int fu = Find(u), fv = Find(v);
45             if (fu != fv)
46             {
47                 fa[fu] = fv;
48                 vis[u][v] = vis[v][u] = 1;
49                 ans += e[i].w;
50             }
51         }
52     }
53 }
54 void DFS(int cur, int pre)
55 {
56     for (int i = 2; i <= ptot; i++)
57     {
58         if (i != pre && vis[cur][i])
59         {
60             if (dp[i].w == -1)
61             {
62                 if (dp[cur].w < a[cur][i])
63                 {
64                     dp[i].u = cur;
65                     dp[i].v = i;
66                     dp[i].w = a[cur][i];
67                 }
68                 else
69                     dp[i] = dp[cur];
```

```
            }
            DFS(i, cur);
        }
    }
}
int main()
{
    ios::sync_with_stdio(false);
    cin.tie(0);
    cin >> n;
    string s1, s2;
    int len;
    name["Park"] = ++ptot;
    memset(a, 0x3f, sizeof(a));
    memset(d, 0x3f, sizeof(d));
    //Park: 1
    for (int i = 0; i < n; i++)
    {
        cin >> s1 >> s2 >> len;
        if (!name[s1])
            name[s1] = ++ptot;
        if (!name[s2])
            name[s2] = ++ptot;
        int u = name[s1], v = name[s2];
        a[u][v] = a[v][u] = min(a[u][v], len); //无向图邻接矩阵
        e.push_back({u, v, len});
    }
    cin >> s; //度数限制
    ans = 0;
    Kruskal();
    for (int i = 2; i <= ptot; i++)
    {
        if (a[1][i] != inf)
        {
            int rt = Find(i);
            if (d[rt] > a[1][i])
                d[rt] = a[1][i], ver[rt] = i;
        }
    }
    for (int i = 2; i <= ptot; i++)
    {
        if (d[i] != inf)
        {
            s--;
            ans += d[i];
            vis[1][ver[i]] = vis[ver[i]][1] = 1;
        }
    }
    while (s-- > 0)
    {
        memset(dp, -1, sizeof(dp));
        dp[1].w = -inf;
        for (int i = 2; i <= ptot; i++)
        {
            if (vis[1][i])
                dp[i].w = -inf;
        }
        DFS(1, -1);
        int w = -inf;
        int v;
        for (int i = 2; i <= ptot; i++)
        {
            if (w < dp[i].w - a[1][i])
            {
                w = dp[i].w - a[1][i];
                v = i;
```

```
136             }
137         }
138         if (w <= 0)
139             break;
140         ans -= w;
141         vis[1][v] = vis[v][1] = 1;
142         vis[dp[v].u][dp[v].v] = vis[dp[v].v][dp[v].u] = 0;
143     }
144     cout << "Total miles driven: " << ans << endl;
145     system("pause");
146     return 0;
147 }
```

POJ2728 Desert King （最优比率生成树，0/1 分数规划，二分）
黑暗城堡（最短路径生成树计数，最短路，排序）

## 6.3 树的直径

### 6.3.1 树形 DP 求树的直径

仅能求出直径长度，无法得知路径信息，可处理负权边。

```
int dp[N];
//dp[rt] 以rt为根的子树 从rt出发最远可达距离
/*
    对于每个结点x f[x]:经过节点x的最长链长度
*/
void DP(int rt)
{
    dp[rt]=0;//单点
    vis[rt]=1;
    for(int i=head[rt];i;i=nxt[i])
    {
        int s=ver[i];
        if(!vis[s])
        {
            DP(s);
            diameter=max(diameter,dp[rt]+dp[s]+edge[i]);
            dp[rt]=max(dp[rt],dp[s]+edge[i]);
        }
    }
}
```

### 6.3.2 两次 BFS/DFS 求树的直径

无法处理负权边，容易记录路径

```
void DFS(int start,bool record_path)
{
    vis[start]=1;
    for(int i=head[start];i;i=nxt[i])
    {
        int s=ver[i];
        if(!vis[s])
        {
            dis[s]=dis[start]+edge[i];
            if(record_path) path[s]=i;
            DFS(s,record_path);
        }
    }
    vis[start]=0;//清理
}
```

例题分析

P3629 [APIO2010] 巡逻（两种求树直径方法的综合应用）

P1099 树网的核（枚举）

## 6.4 最近公共祖先（LCA）

### 6.4.1 树上倍增

```
void BFS()
{
    queue<int> q;
    q.push(1);
    d[1] = 1;
    while (!q.empty())
    {
        int x = q.front();
        q.pop();
        for (int i = head[x]; i; i = nxt[i])
```

```
        {
            int y = ver[i];
            if (!d[y])
            {
                d[y] = d[x] + 1;
                fa[y][0] = x;
                for (int j = 1; j <= k; j++)
                {
                    fa[y][j] = fa[fa[y][j - 1]][j - 1];
                }
                q.push(y);
            }
        }
    }
}
int LCA(int x, int y)
{
    if (d[x] < d[y])
        swap(x, y);
    for (int i = k; i >= 0; i--)
        if (d[fa[x][i]] >= d[y])
            x = fa[x][i];
    if (x == y)
        return y;
    for (int i = k; i >= 0; i--)
        if (fa[x][i] != fa[y][i])
            x = fa[x][i], y = fa[y][i];
    return fa[x][0];
}
```

### 6.4.2 Tarjan

```
int Find(int x)
{
    if (x == fa[x])
        return x;
    return fa[x] = Find(fa[x]);
}
void Tarjan(int x)
{
    vis[x] = 1;
    for (int i = head[x]; i; i = nxt[i])
    {
        int y = ver[i];
        if (!vis[y])
        {
            Tarjan(y);
            fa[y] = x;
        }
    }
    for (int i = 0; i < q[x].size(); i++)
    {
        int y = q[x][i].first, id = q[x][i].second;
        if (vis[y] == 2)
            lca[id] = Find(y);
    }
    vis[x] = 2;
}
```

## 6.5 树上差分

例题分析

POJ3417 Network（LCA，树上差分，边覆盖）

6302 雨天的尾巴（LCA，树上差分，点覆盖，权值线段树，线段树合并）
P1600 天天爱跑步（LCA，树上差分）

## 6.6 LCA 的综合应用

例题分析
CH56C 异象石（dfn 时间戳，LCA）
P4180 【模板】严格次小生成树 [BJWC2010]（树上倍增）

```
#include <algorithm>
#include <iostream>
#include <math.h>
#include <queue>
#include <stdio.h>
using namespace std;
const int N = 1e5 + 50;
const int M = 6e5 + 50;
#define ll long long
#define pii pair<int, int>
#define inf 0x3f3f3f3f
int n, m, k, F[N][20], d[N], fa[N];
int head[N], ver[M], nxt[M], edge[M], tot;
ll G[N][20][2];
void add(int x, int y, int z)
{
    ver[++tot] = y, nxt[tot] = head[x], head[x] = tot, edge[tot] = z;
}
struct edge
{
    int x, y, z;
    bool used;
    bool operator<(const edge &e) const
    {
        return z < e.z;
    }
} e[M];
int Find(int x)
{
    if (fa[x] == x)
        return x;
    return fa[x] = Find(fa[x]);
}
ll Kruskal()
{
    for (int i = 1; i <= n; i++)
        fa[i] = i;
    sort(e + 1, e + 1 + m);
    ll ans = 0;
    int cnt = 0;
    for (int i = 1; i <= m; i++)
    {
        int fx = Find(e[i].x), fy = Find(e[i].y);
        if (fx != fy)
        {
            fa[fx] = fy;
            ans += e[i].z;
            e[i].used = true;
            cnt++;
            if (cnt >= n - 1)
                break;
        }
    }
    return ans;
}
void BFS()
```

```
{
    k = log2(n) + 1;
    queue<int> q;
    q.push(1), d[1] = 1;
    for (int i = 0; i <= k; i++)
        G[1][i][0] = G[1][i][1] = -inf;
    while (q.size())
    {
        int x = q.front();
        q.pop();
        for (int i = head[x]; i; i = nxt[i])
        {
            int y = ver[i];
            if (!d[y])
            {
                d[y] = d[x] + 1;
                F[y][0] = x;
                G[y][0][0] = edge[i];
                G[y][0][1] = -inf;
                for (int j = 1; j <= k; j++)
                {
                    F[y][j] = F[F[y][j - 1]][j - 1];
                    G[y][j][0] = max(G[y][j - 1][0], G[F[y][j - 1]][j - 1][0]);
                    if (G[y][j - 1][0] == G[F[y][j - 1]][j - 1][0])
                        G[y][j][1] = max(G[y][j - 1][1], G[F[y][j - 1]][j - 1][1]);
                    else if (G[y][j - 1][0] > G[F[y][j - 1]][j - 1][0])
                        G[y][j][1] = max(G[F[y][j - 1]][j - 1][0], G[y][j - 1][1]);
                    else
                        G[y][j][1] = max(G[y][j - 1][0], G[F[y][j - 1]][j - 1][1]);
                }
                q.push(y);
            }
        }
    }
}
pii LCA(int x, int y)
{
    ll val1 = -inf, val2 = -inf;
    if (d[y] > d[x])
        swap(x, y);
    for (int i = k; i >= 0; i--)
    {
        if (d[F[x][i]] >= d[y])
        {
            if (G[x][i][0] > val1)
                val1 = G[x][i][0], val2 = max(val2, G[x][i][1]);
            else if (G[x][i][0] < val1)
                val2 = max(val2, G[x][i][0]);
            x = F[x][i];
        }
    }
    if (x == y)
        return make_pair(val1, val2);
    for (int i = k; i >= 0; i--)
    {
        if (F[x][i] != F[y][i])
        {
            val1 = max(val1, max(G[x][i][0], G[y][i][0]));
            val2 = max(val2, (val1 == G[x][i][0]) ? G[x][i][1] : G[x][i][0]);
            val2 = max(val2, (val1 == G[y][i][0]) ? G[y][i][1] : G[y][i][0]);
            x = F[x][i], y = F[y][i];
        }
    }
    val1 = max(val1, max(G[x][0][0], G[y][0][0]));
    val2 = max(val2, (val1 == G[x][0][0]) ? G[x][0][1] : G[x][0][0]);
    val2 = max(val2, (val1 == G[y][0][0]) ? G[y][0][1] : G[y][0][0]);
```

```
123     return make_pair(val1, val2);
124 }
125 int main()
126 {
127     scanf("%d%d", &n, &m);
128     for (int i = 1; i <= m; i++)
129         scanf("%d%d%d", &e[i].x, &e[i].y, &e[i].z), e[i].used = false;
130     ll sum = Kruskal(), ans = 0x3f3f3f3f3f3f3f3f;
131     for (int i = 1; i <= m; i++)
132         if (e[i].used)
133             add(e[i].x, e[i].y, e[i].z), add(e[i].y, e[i].x, e[i].z);
134     BFS();
135     for (int i = 1; i <= m; i++)
136     {
137         if (!e[i].used)
138         {
139             pii temp = LCA(e[i].x, e[i].y);
140             if (e[i].z > temp.first)
141                 ans = min(ans, sum - temp.first + e[i].z);
142             else if (e[i].z == temp.first)
143                 ans = min(ans, sum - temp.second + e[i].z);
144         }
145     }
146     cout << ans << endl;
147     //system("pause");
148     return 0;
149 }
```

P1084 疫情控制（二分，树上倍增，贪心）

## 6.7 基环树

基环树直径

```cpp
#include <iostream>
#include <queue>
#include <stdio.h>
using namespace std;
const int N = 1e6 + 50;
const int M = 2e6 + 50;
#define ll long long
int n;
int head[N], nxt[M], edge[M], ver[M], tot, in_degree[N];
int c[N], q[N << 1];  //c[]属于哪一个基环树 q[]双端队列
ll diameter[N], d[N]; //求子树直径
ll a[N << 1], sum[N << 1];
bool vis[N];
inline void add(int x, int y, int z)
{
    ver[++tot] = y, nxt[tot] = head[x], head[x] = tot, edge[tot] = z, in_degree[y]++;
}
void BFS(int start, int color)
{
    int l = 1, r = 1;
    c[start] = color;
    q[1] = start;
    while (l <= r)
    {
        for (int i = head[q[l]]; i; i = nxt[i])
        {
            int y = ver[i];
            if (!c[y])
            {
                c[y] = color;
                q[++r] = y;
            }
        }
        l++;
    }
}
void Topo()
{
    queue<int> q;
    for (int i = 1; i <= n; i++)
        if (in_degree[i] == 1)
            q.push(i); //不在环上点入队
    while (q.size())
    {
        int x = q.front();
        q.pop();
        for (int i = head[x]; i; i = nxt[i])
        {
            int y = ver[i];
            if (in_degree[y] > 1) //无向图避免访问父节点
            {
                diameter[c[x]] = max(diameter[c[x]], d[x] + d[y] + edge[i]);
                d[y] = max(d[y], d[x] + edge[i]); // 维护直径  这里的写法与树形DP中的递归并不完全一致 顺序是自底向上
                if (--in_degree[y] == 1)
                    q.push(y);
            }
        }
    }
}
void DP(int color, int x)
{
    int pcnt = 0; //环上点的计数
```

```
63      int y = x, k; //y:起点->终点
64      do
65      {
66          a[++pcnt] = d[y];
67          /*
68          a[] 保存从环上各节点出发走向以该节点为根的子树（不访问环上节点）
69          最远能到达的距离
70          sum[i] 保存到环上第i个节点需要经过距离和
71          */
72          in_degree[y] = 1;
73          for (k = head[y]; k; k = nxt[k])
74              if (in_degree[ver[k]] > 1)
75              {
76                  sum[pcnt + 1] = sum[pcnt] + edge[k];
77                  y = ver[k];
78                  break;
79              }
80      } while (k);
81      if (pcnt == 2)
82      {
83          //两点成环
84          int temp = 0;
85          for (int i = head[y]; i; i = nxt[i])
86              if (ver[i] == x)
87                  temp = max(temp, edge[i]);
88          diameter[color] = max(diameter[color], d[x] + d[y] + temp);
89          return;
90      }
91      for (int i = head[y]; i; i = nxt[i])
92          if (ver[i] == x)
93          {
94              sum[pcnt + 1] = sum[pcnt] + edge[i];
95              break;
96          }
97      for (int i = 1; i < pcnt; i++)
98          //断环为链 复制一倍
99          a[i + pcnt] = a[i], sum[i + pcnt] = sum[pcnt + 1] + sum[i];
100     //单调队列优化DP
101     q[1] = 1;
102     int l = 1, r = 1;
103     for (int i = 2; i < (pcnt << 1); i++)
104     {
105         while (l <= r && i - q[l] >= pcnt)
106             l++;
107         diameter[color] = max(diameter[color], a[i] + a[q[l]] + sum[i] - sum[q[l]]);
108         while (l <= r && a[q[r]] - sum[q[r]] <= a[i] - sum[i])
109             r--;
110         q[++r] = i;
111     }
112 }
113 int main()
114 {
115     scanf("%d", &n);
116     for (int x = 1, y, z; x <= n; x++)
117     {
118         scanf("%d%d", &y, &z);
119         add(x, y, z);
120         add(y, x, z);
121     }
122     int ccnt = 0;
123     for (int i = 1; i <= n; i++)
124         if (!c[i])
125             BFS(i, ++ccnt);
126     Topo();
127     ll ans = 0;
128     for (int i = 1; i <= n; i++)
```

```
129     {
130         if (in_degree[i] > 1 && !vis[c[i]])
131         {
132             //i为基环树环上点 且该基环树尚未访问
133             vis[c[i]] = true;
134             DP(c[i], i);
135             ans += diameter[c[i]]; //累加基环树的最长链
136         }
137     }
138     cout << ans << endl;
139     //system("pause");
140     return 0;
141 }
```

## 6.8 负环与差分约束

### 6.8.1 负环

例题分析

POJ3621 Sightseeing Cows（0/1 分数规划，SPFA 判定负环）

### 6.8.2 差分约束系统

例题分析

POJ1201 Intervals（单源最长路）

# 6.9 Tarjan 算法与无向图连通性

## 6.9.1 无向图的割点与桥

### 6.9.1.1 割边判定法则

```
void Tarjan(int x, int in_edge)
{
    dfn[x] = low[x] = ++num;
    for (int i = head[x]; i; i = nxt[i])
    {
        int y = ver[i];
        if (!dfn[y])
        {
            Tarjan(y, i);
            low[x] = min(low[x], low[y]);
            if (low[y] > dfn[x])
            {
                bridge[i] = bridge[i ^ 1] = true;
            }
        }
        else if (i != (in_edge ^ 1))
            low[x] = min(low[x], dfn[y]);
    }
}
```

### 6.9.1.2 割点判定法则

```
void Tarjan(int x)
{
    dfn[x] = low[x] = ++num;
    int flag = 0;
    for (int i = head[x]; i; i = nxt[i])
    {
        int y = ver[i];
        if (!dfn[y])
        {
            Tarjan(y);
            low[x] = min(low[x], low[y]);
            if (low[y] >= dfn[x])
            {
                flag++;
                if (x != root || flag >= 2)
                    cut[x] = true;
            }
        }
        else
            low[x] = min(low[x], dfn[y]);
    }
}
```

例题分析
P3469 [POI2008]BLO-Blockade（割点，连通块计数）

## 6.9.2 无向图的双连通分量

### 6.9.2.1 边双连通分量 e-DCC 与其缩点

```
void DFS(int x)
{
    color[x] = dcc;
    for (int i = head[x]; i; i = nxt[i])
    {
        int y = ver[i];
        if (!color[y] && !bridge[i])
```

```
8               DFS(y);
9       }
10  }
11  void e_DCC()
12  {
13      dcc = 0;
14      for (int i = 1; i <= n; i++)
15          if (!color[i])
16              ++dcc, DFS(i);
17      totc = 1;
18      for (int i = 2; i <= tot; i++)
19      {
20          int u = ver[i ^ 1], v = ver[i];
21          if (color[u] != color[v])
22              add_c(color[u], color[v]);
23      }
24  }
```

#### 6.9.2.2 点双连通分量 v-DCC 与其缩点

```
1   void Tarjan(int x)
2   {
3       dfn[x] = low[x] = ++num;
4       int flag = 0;
5       stack[++top] = x;
6       if (x == root && !head[x])
7       {
8           dcc[++cnt].push_back(x);
9           return;
10      }
11      for (int i = head[x]; i; i = nxt[i])
12      {
13          int y = ver[i];
14          if (!dfn[y])
15          {
16              Tarjan(y);
17              low[x] = min(low[x], low[y]);
18              if (low[y] >= dfn[x])
19              {
20                  flag++;
21                  if (x != root || flag >= 2)
22                      cut[x] = true;
23                  cnt++;
24                  int z;
25                  do
26                  {
27                      z = stack[top--];
28                      dcc[cnt].push_back(z);
29                  } while (z != y);
30                  dcc[cnt].push_back(x);
31              }
32          }
33          else
34              low[x] = min(low[x], dfn[y]);
35      }
36  }
37  void v_DCC()
38  {
39      cnt = 0;
40      top = 0;
41      for (int i = 1; i <= n; i++)
42      {
43          if (!dfn[i])
44              root = i, Tarjan(i);
45      }
```

```
    // 给每个割点一个新的编号(编号从cnt+1开始)
    num = cnt;
    for (int i = 1; i <= n; i++)
        if (cut[i]) new_id[i] = ++num;
    // 建新图，从每个v-DCC到它包含的所有割点连边
    tc = 1;
    for (int i = 1; i <= cnt; i++)
        for (int j = 0; j < dcc[i].size(); j++)
        {
            int x = dcc[i][j];
            if (cut[x]) {
                add_c(i, new_id[x]);
                add_c(new_id[x], i);
            }
            else c[x] = i; // 除割点外，其它点仅属于1个v-DCC
        }
}
```

例题分析

POJ3694 Network（e-DCC 缩点，LCA，并查集）

POJ2942 Knights of the Round Table（补图，v-DCC，染色法奇环判定）

### 6.9.3 欧拉路问题

欧拉图的判定

无向图连通，所有点度数为偶数。

欧拉路的存在性判定

无向图连通，恰有两个节点度数为奇数，其他节点度数均为偶数

```
// 模拟系统栈，答案栈
void Euler() {
    stack[++top] = 1;
    while (top > 0) {
        int x = stack[top], i = head[x];
        // 找到一条尚未访问的边
        while (i && vis[i]) i = Next[i];
        // 沿着这条边模拟递归过程，标记该边，并更新表头
        if (i) {
            stack[++top] = ver[i];
            head[x] = Next[i];
            vis[i] = vis[i ^ 1] = true;
        }
        // 与x相连的所有边均已访问，模拟回溯过程，并记录于答案栈中
        else {
            top--;
            ans[++t] = x;
        }
    }
}
```

例题分析

POJ2230 Watchcow（欧拉回路）

## 6.10 Tarjan 算法与有向图连通性

### 6.10.1 强连通分量（SCC）判定法则

```
void Tarjan(int x)
{
    dfn[x]=low[x]=++num;
    stack[++top]=x,in_stack[x]=true;
    for(int i=head[x];i;i=nxt[i])
    {
        int y=ver[i];
        if(!dfn[y])
        {
            Tarjan(y);
            low[x]=min(low[x],low[y]);
        }
        else if(in_stack[y])
            low[x]=min(low[x],dfn[y]);
    }
    if(dfn[x]==low[x])
    {
        cnt++;
        int y;
        do
        {
            y=stack[top--],in_stack[y]=false;
            color[y]=cnt, scc[cnt].push_back(y);
        } while (x!=y);
    }
}
```

### 6.10.2 SCC -> DAG

```
void SCC()
{
    for (int i = 0; i <= n; i++)
        if (!dfn[i])
            Tarjan(i);
    //缩点
    for (int x = 1; x <= n; x++)
    {
        for (int i = head[x]; i; i = nxt[i])
        {
            int y = ver1[i];
            if (color[x] != color[y])
                add_c(color[x], color[y]);
        }
    }
}
```

例题分析

POJ1236 Network of Schools（SCC->DAG，入度出度）

P3275 [SCOI2011] 糖果（SPFA TLE，SCC->DAG，Topo，DP）

### 6.10.3 有向图的必经点与必经边

对于有向无环图（DAG）：

在原图中按照拓扑序进行动态规划，求出起点 S 到图中每个点 x 的路径条数 fs[x]。

在反图上再次按照拓扑序进行动态规划，求出每个点 x 到终点 T 的路径条数 ft[x]。

显然，fs[T] 表示从 S 到 T 的路径总条数。根据乘法原理：

1：对于一条有向边 (x,y)，若 fs[x]*ft[y]=fs[T]，则 (x,y) 是有向无环图从 S 到 T 的必经边。

2：对于一个点 x，若 fs[x]*ft[x]=fs[T]，则 x 是有向无环图从 S 到 T 的必经点。

路径条数规模较大，可对大质数取模后保存，但有概率误判。

例题分析

6703 PKU ACM Team's Excursion （DAG 必经边，枚举，DP）

### 6.10.4 2-SAT 问题

原命题与逆否命题互为等价命题，在建立有向边关系时要注意对称性。

例题分析

POJ3678 Katu Puzzle（2-SAT，Tarjan,，SCC）

POJ3683 Priest John's Busiest Day（2-SAT 方案构造）

```
for (int i = 1; i <= 2 * n; i++)
{
    val[i] = color[i] > color[opp[i]];
    //若c[i] 大于 c[opp[i]]   opp[i]赋0
}
```

## 6.11 二分图的匹配

### 6.11.1 二分图判定

一张无向图是二分图，当且仅当图中不存在奇环（长度为奇数的环）。

```
//染色法判定奇环
bool DFS(int x,int color)
{
    vis[x]=color;
    for(int i=head[x];i;i=nxt[i])
    {
        int y=ver[i];
        if(!vis[y])
        {
            if(!DFS(y,3-color)) return false;
        }
        else if(vis[y]==color) return false;
    }
    return true;
}
```

例题分析
P1525 关押罪犯（判定二分图，二分）

### 6.11.2 二分图最大匹配

二分图匹配的模型要素
1：节点能分成独立的两个集合，每个集合内部有 0 条边。“0 要素”
2：每个节点只能与 1 条匹配边相连。“1 要素”

```
\\匈牙利算法
\\在主函数中对每个左部节点调用寻找增广路时，需要对 vis 重置。
bool DFS(int x)
{
    for (int i = head[x]; i; i = nxt[i])
    {
        int y = ver[i];
        if (!vis[y])
        {
            vis[y] = 1;
            if (!match[y] || DFS(match[y]))
            {
                match[y] = x;
                return true;
            }
        }
    }
    return false;
}
```

例题分析
6801 棋盘覆盖（奇偶染色）
6802 車的放置（行列）
6803 导弹防御塔（二分，拆点多重匹配）

### 6.11.3 二分图带权匹配

二分图带权最大匹配的前提是匹配数最大，然后再最大化匹配边的权值总和。

```
/*KM 稠密图上效率高于费用流，但是有较大局限性，只能在满足“带权最大匹配一定是完备匹配”的图中正确求解。
w[][]:边权
la[], lb[]: 左，右部点顶标
visa[], visb[]: 访问标记，是否在交错树中
ans: Σw[match[i]][i]
*/
bool DFS(int x)
```

```
{
    visa[x] = true;
    for (int y = 1; y <= n; y++)
    {
        if (!visb[y])
        {
            double temp = fabs(la[x] + lb[y] - w[x][y]);//对于浮点数，相等子图的判定
            if (temp < eps)
            {
                visb[y] = true;
                if (!match[y] || DFS(match[y]))
                {
                    match[y] = x;
                    return true;
                }
            }
            else
                upd[y] = min(upd[y], la[x] + lb[y] - w[x][y]);
        }
    }
    return false;
}
void KM()
{
    for (int i = 1; i <= n; i++)
    {
        la[i] = -inf;
        lb[i] = 0;
        for (int j = 1; j <= n; j++)
            la[i] = max(la[i], w[i][j]);
    }
    for (int i = 1; i <= n; i++)
    {
        while (true)
        {
            memset(visa, 0, sizeof(visa));
            memset(visb, 0, sizeof(visb));
            for (int j = 1; j <= n; j++)
                upd[j] = inf;
            if (DFS(i))
                break;
            else
            {
                delta = inf;
                for (int j = 1; j <= n; j++)
                    if (!visb[j])
                        delta = min(delta, upd[j]);
                for (int j = 1; j <= n; j++)
                {
                    if (visa[j])
                        la[j] -= delta;
                    if (visb[j])
                        lb[j] += delta;
                }
            }
        }
    }
}
```

例题分析

POJ3565 Ants（三角形不等式，二分图带权最小匹配）

## 6.12 二分图的覆盖与独立集

### 6.12.1 二分图最小点覆盖

二分图最小覆盖模型特点：
每条边有 2 个端点，二者至少选择一个。“2 要素”

#### 6.12.1.1 König's theorem

二分图最小点覆盖包含的点数等于二分图最大匹配包含的边数。
例题分析
POJ1325 Machine Schedule（二分图最小覆盖）
POJ2226 Muddy Fields（行列连续块，二分图最小覆盖）

### 6.12.2 二分图最大独立集

无向图 G 的最大团等于其补图 G' 的最大独立集。(补图转化)
设 G 是有 n 个节点的二分图，G 的最大独立集的大小等于 n 减去最大匹配数。
例题分析
6901 骑士放置（奇偶染色）

### 6.12.3 有向无环图的最小路径点覆盖

给定一张有向无环图，要求用尽量少的不相交的简单路径，覆盖有向无环图的所有顶点（也就是每个顶点恰好被覆盖一次）。这个问题被称为有向无环图的最小路径点覆盖，简称“最小路径覆盖”。

有向无环图 G 的最小路径点覆盖包含的路径条数，等于 n（有向无环图的点数）减去拆点二分图 G2 的最大匹配数。

若简单路径可相交，即一个节点可被覆盖多次，这个问题称为有向无环图的最小路径可重复点覆盖。

对于这个问题，可先对 G 求传递闭包，得到有向无环图 G'，再在 G' 上求一般的（路径不可相交的）最小路径点覆盖。
例题分析
6902 Vani 和 Cl2 捉迷藏（最小路径可重复点覆盖，构造方案）

```
// 构造方案，先把所有路径终点（左部非匹配点）作为藏身点
for (int i = 1; i <= n; i++) succ[match[i]] = true;
for (int i = 1, k = 0; i <= n; i++)
    if (!succ[i]) hide[++k] = i;
memset(vis, 0, sizeof(vis));
bool modify = true;
while (modify) {
    modify = false;
    // 求出 next(hide)
    for (int i = 1; i <= ans; i++)
        for (int j = 1; j <= n; j++)
            if (cl[hide[i]][j]) vis[j] = true;
    for (int i = 1; i <= ans; i++)
        if (vis[hide[i]]) {
            modify = true;
            // 不断向上移动
            while (vis[hide[i]]) hide[i] = match[hide[i]];
        }
}
for (int i = 1; i <= ans; i++) printf("%d ", hide[i]);
cout << endl;
```

# 6.13 网络流初步

## 6.13.1 最大流

### 6.13.1.1 Edmonds Karp 增广路

```
bool BFS() {
    memset(vis, 0, sizeof(vis));
    queue<int> q;
    q.push(S); vis[S] = 1;
    incf[S] = inf; // 增广路上各边的最小剩余容量
    while (q.size()) {
        int x = q.front(); q.pop();
        for (int i = head[x]; i; i = Next[i])
            if (edge[i]) {
                int y = ver[i];
                if (vis[y]) continue;
                incf[y] = min(incf[x], edge[i]);
                pre[y] = i; // 记录前驱，便于找到最长路的实际方案
                q.push(y), vis[y] = 1;
                if (y == t) return 1;
            }
    }
    return 0;
}
void Update() { // 更新增广路及其反向边的剩余容量
    int x = t;
    while (x != s) {
        int i = pre[x];
        edge[i] -= incf[t];
        edge[i ^ 1] += incf[t]; // 利用“成对存储”的xor 1技巧
        x = ver[i ^ 1];
    }
    maxflow += incf[t];
}
```

### 6.13.1.2 Dinic

```
//可加入当前弧优化（&）：在增广时复制head[]到cur[]，在增广时同步修改cur[]，目的是递归时跳过已增广的边。
bool BFS()
{
    memset(d, 0, sizeof(d));
    queue<int> q;
    q.push(S);
    d[S] = 1; //不为1  陷入死循环
    while (q.size())
    {
        int x = q.front();
        q.pop();
        for (int i = head[x]; i; i = nxt[i])
        {
            int y = ver[i];
            if (edge[i] && !d[y])
            {
                d[y] = d[x] + 1;
                q.push(y);
                if (y == T)
                    return true;
            }
        }
    }
    return false;
}
int Dinic(int x, int flow)
{
```

```
28      if (x == T)
29          return flow;
30      int rest = flow, k;
31      for (int i = head[x]; i && rest; i = nxt[i])
32      {
33          int y = ver[i];
34          if (edge[i] && d[y] == d[x] + 1)
35          {
36              k = Dinic(y, min(edge[i], rest));
37              if (!k)
38                  d[y] = 0;
39              edge[i] -= k;
40              edge[i ^ 1] += k;
41              rest -= k;
42          }
43      }
44      return flow - rest;
45  }
```

#### 6.13.1.3 二分图最大匹配的必须边与可行边

在一般的二分图中，可以用最大流计算任一组最大匹配。
此时：必须边的判定条件为：(x,y) 流量为 1 ，并且在残量网络上属于不同的 SCC。
可行边的判定条件为：(x,y) 流量为 1 ，或者在残量网络上属于同一个 SCC。
例题分析
CH17C 舞动的夜晚（Dinic，Tarjan，二分图可行边）

### 6.13.2 最小割

#### 6.13.2.1 最大流最小割定理

任何一个网络的最大流量等于最小割中边的容量之和。
例题分析
POJ1966 Cable TV Network （枚举，点边转化）

### 6.13.3 费用流

#### 6.13.3.1 Edmonds Karp 增广路

BFS 寻找增广路 -> SPFA 寻找单位费用之和最小的增广路（将费用作为边权，在残量网络上求最短路）。
注意：反向边的费用为相反数。

```
1   void add(int x, int y, int z, int c)
2   {
3       ver[++tot] = y, nxt[tot] = head[x], head[x] = tot, edge[tot] = z, cost[tot] = c;
4       ver[++tot] = x, nxt[tot] = head[y], head[y] = tot, edge[tot] = 0, cost[tot] = -c;
5   }
6   bool SPFA()
7   {
8       memset(dis, 0x3f, sizeof(dis)); //inf
9       memset(vis, 0, sizeof(vis));
10      queue<int> q;
11      dis[S] = 0, vis[S] = 1, incf[S] = 1 << 30;
12      q.push(S);
13      while (q.size())
14      {
15          int x = q.front();
16          q.pop();
17          vis[x] = 0;
18          for (int i = head[x]; i; i = nxt[i])
19          {
20              if (edge[i])
21              {
22                  int y = ver[i];
23                  if (dis[y] > dis[x] + cost[i])//注意修改不等号及初始化值
24                  {
```

```
                    dis[y] = dis[x] + cost[i];
                    incf[y] = min(incf[x], edge[i]);
                    pre[y] = i;
                    if (!vis[y])
                        q.push(y), vis[y] = 1;
                }
            }
        }
    }
    if (dis[T] >= inf)
        return false;
    return true;
}
ll max_flow, ans;
void Update()
{
    int x = T;
    while (x != S)
    {
        int i = pre[x];
        edge[i] -= incf[T];
        edge[i ^ 1] += incf[T];
        x = ver[i ^ 1];
    }
    max_flow += incf[T];
    ans += incf[T] * dis[T];
}
```

例题分析

POJ3422 Kaka's Matrix Travels（点边转化，费用流）