ACM Template

Macmillan School of Magic

Macmillan Expedition

Gene_Liu

July 5, 2020

Contents

0 常规操作 **1**

1 数学 **8**
1.1 快速幂与快速乘 . . . 8
1.1.1 快速幂 . . . 8
1.1.2 矩阵快速幂 . . . 8
1.1.3 快速乘取膜 . . . 9
1.2 筛 . . . 10
1.2.1 埃氏筛 . . . 10
1.2.2 线性筛 . . . 11
1.2.3 杜教筛 . . . 11
1.3 GCD 与逆元 . . . 13
1.3.1 GCD . . . 13
1.3.2 Exgcd 与逆元 . . . 13
1.3.3 O(n) 打表求逆元 . . . 14
1.4 唯一分解 . . . 14
1.5 中国剩余定理 . . . 15
1.5.1 中国剩余定理拓展版 . . . 15
1.5.2 中国剩余定理 python 版本 . . . 16
1.6 求组合数 . . . 17
1.6.1 Lucas (模数为质数) . . . 17
1.6.2 递推求组合数 . . . 18
1.7 线性基 . . . 18
1.7.1 线性基模板 . . . 18
1.7.2 无向图异或路径问题 . . . 20
1.7.3 异或为 X 的子集长度和 . . . 22
1.7.4 前缀线性基 . . . 24
1.7.5 线段树维护线性基交 . . . 25
1.8 二次剩余 . . . 27
1.9 常见数学式子 . . . 28
1.10 数论板子集合 . . . 28
1.11 斐波那契数列循环节 . . . 30
1.12 划分数 . . . 33
1.13 卡特兰数 . . . 36
1.14 高斯素数 . . . 36
1.15 莫比乌斯反演 . . . 37
1.16 容斥原理 . . . 39

2 博弈 **41**
2.1 SG 函数 . . . 41

3 动态规划 **43**
3.1 01 背包 . . . 43
3.2 完全背包 . . . 44
3.3 多重背包 . . . 47
3.4 状压 dp . . . 48
3.4.1 子集枚举 . . . 48
3.4.2 计算一对元素贡献 . . . 49
3.5 区间 dp . . . 51
3.5.1 区间 dp 例题 1 . . . 51
3.5.2 区间 dp 例题 2 . . . 52
3.6 数位 dp . . . 53
3.6.1 数位 dp 模板 . . . 53
3.6.2 巧妙数位 dp 例题 . . . 54
3.6.3 双数 dp 问题 . . . 55
3.6.4 三数 dp 问题 . . . 57
3.6.5 三进制状压 . . . 59
3.6.6 数字平方和 . . . 61

3.6.7 被数位整除 . . . 63
3.6.8 数位逆序对之和 . . . 64
3.7 概率 dp . . . 66
3.8 斜率优化 dp . . . 68
3.8.1 斜率优化例题 . . . 68
3.8.2 动态维护下凸壳 . . . 69
3.9 斯坦纳树 . . . 71
3.9.1 斯坦纳树模板 . . . 71
3.9.2 斯坦纳森林 . . . 73
3.9.3 斯坦纳树路径输出 . . . 75
3.10 长链剖分 . . . 77
3.10.1 k 级祖先 . . . 77
3.10.2 O(1) 优化树形 dp 模板 . . . 79
3.10.3 长链剖分维护后缀和 . . . 80
3.10.4 长链剖分 + 线段树 . . . 82
3.10.5 三元组两两距离相等 . . . 85
3.11 巧妙 dp 题 . . . 87
3.11.1 亏欠型 DP . . . 87
3.11.2 数位 DP+ 状压 DP . . . 88

4 数据结构 92
4.1 单调栈 . . . 92
4.2 单调队列 . . . 93
4.3 并查集 . . . 94
4.3.1 普通并查集 . . . 94
4.3.2 带权并查集 . . . 94
4.3.3 带权并查集与背包 . . . 96
4.3.4 按秩合并并查集 . . . 98
4.4 Hash . . . 100
4.5 KMP . . . 102
4.6 字典树 . . . 103
4.7 ST 表 . . . 104
4.8 树状数组 . . . 104
4.9 线段树 . . . 105
4.9.1 动态开点与区间修改 lazy . . . 105
4.9.2 区间递减序列和 . . . 105
4.9.3 线段树维护图连通 . . . 107
4.9.4 线段树维护树直径 . . . 109
4.9.5 区间不同 gcd 个数 . . . 111
4.10 扫描线 . . . 112
4.10.1 面积并 . . . 112
4.10.2 面积交 . . . 114
4.10.3 周长并 . . . 116
4.10.4 包星星问题 . . . 118
4.10.5 覆盖奇数次的面积 . . . 120
4.11 主席树 . . . 122
4.11.1 静态主席树 . . . 122
4.11.2 区间中不同数的个数 . . . 124
4.11.3 单点修改主席树 . . . 126
4.12 李超树 . . . 128
4.12.1 李超树模板题 . . . 128
4.12.2 离散化 + 维护最大最小直线 . . . 129
4.12.3 李超树 + 两个 lazy . . . 132
4.12.4 树剖 + 李超树 + 区间直线最大值 . . . 135
4.13 启发式合并 . . . 138
4.13.1 DSU 模板 . . . 138
4.13.2 例题 1 . . . 139
4.13.3 例题 2 . . . 141
4.14 Splay . . . 143

4.14.1 普通 Splay . . . 143
4.14.2 区间 Splay . . . 145
4.14.3 Splay (三个 lazy 标记) . . . 148
4.15 LCT . . . 152
4.15.1 LCT 模板题 . . . 152
4.15.2 边权 LCT . . . 157
4.15.3 LCT 最大连续和 . . . 159
4.15.4 最小差值生成树 . . . 162
4.15.5 维护实边与虚边 . . . 164
4.15.6 TopTree . . . 166
4.16 树链剖分 . . . 171
4.16.1 点剖模板 . . . 171
4.16.2 树上路径颜色段数量 . . . 174
4.16.3 动态开点树剖 . . . 177
4.16.4 树剖换根 . . . 179
4.16.5 边剖 + 主席树 . . . 182
4.17 CDQ . . . 185
4.17.1 陌上开花 (三维偏序) . . . 185
4.17.2 动态逆序对 . . . 186
4.17.3 矩阵前缀和 . . . 188
4.18 莫队 . . . 190
4.18.1 普通莫队 . . . 190
4.18.2 莫队求组合书前缀和 . . . 192
4.18.3 带修改莫队 . . . 192
4.18.4 树上带修改莫队 . . . 194
4.18.5 回滚莫队（增加） . . . 197
4.18.6 回滚莫队（减少） . . . 199
4.19 虚树 . . . 200
4.20 十字链表 . . . 203
4.21 柯朵莉树 . . . 205
4.22 AC 自动机 . . . 207
4.22.1 AC 自动机模板 . . . 207
4.22.2 AC 自动机 +DP1 . . . 209
4.22.3 AC 自动机 + 最短路 . . . 212
4.22.4 AC 自动机 +DP2 . . . 214
4.22.5 AC 自动机 + 状压 DP . . . 217
4.22.6 AC 自动机 + 缩点 . . . 220
4.22.7 阿狸打字机 . . . 222
4.23 回文串问题 . . . 225
4.23.1 Manacher 模板 . . . 225
4.23.2 回文自动机模板 . . . 226
4.23.3 模板题 . . . 228
4.23.4 Manacher 例题 . . . 230
4.23.5 Manacher+ 回文自动机 . . . 231
4.23.6 两颗回文自动机 . . . 233
4.23.7 马拉车 + 字典树 . . . 234
4.24 后缀数组 . . . 236
4.24.1 后缀数组模板 . . . 236
4.24.2 单字符串常见问题 . . . 238
4.24.3 最长公共子串 (双串) . . . 241
4.24.4 长度不小于 k 的公共子串个数 . . . 244
4.24.5 n 字符串问题 . . . 247
4.25 整体二分 . . . 247
4.25.1 无修改第 K 大 . . . 247
4.25.2 带修改第 K 大 . . . 249
4.25.3 线段树 + 整体二分 . . . 251
4.25.4 并查集 + 整体二分 . . . 253
4.26 分块 . . . 255

5 图论 **258**
5.1 最短路 . . . 258
5.1.1 Dijkstra . . . 258
5.1.2 Dijkstra 记录路径 . . . 259
5.1.3 分层图 . . . 261
5.1.4 spfa 判断负环 . . . 263
5.1.5 最短路计数问题 . . . 264
5.2 最小生成树 . . . 266
5.3 最近公共祖先 . . . 267
5.3.1 LCA (dfs 版本) . . . 267
5.3.2 LCA (bfs 版本) . . . 268
5.4 拓扑排序 . . . 270
5.5 欧拉路 . . . 272
5.6 双连通分量 . . . 273
5.6.1 边双缩点以及求割边 . . . 273
5.6.2 边双缩点 LCA . . . 275
5.6.3 点双缩点以及求割点 . . . 278
5.6.4 点双例题 . . . 281
5.7 强连通分量 . . . 283
5.7.1 强连通分量及缩点 . . . 283
5.7.2 kosaraju . . . 285
5.7.3 2-SAT . . . 286
5.8 二分图匹配 . . . 288
5.9 第 K 小问题 . . . 290
5.9.1 S 到 T 的第 K 短路 . . . 290
5.9.2 有向全图第 k 小路径 . . . 292
5.9.3 无向全图第 k 小最短路 . . . 293
5.9.4 无向全图第 k 小团 . . . 295
5.10 简单图构造 . . . 296

6 网络流 **299**
6.1 最大流 . . . 299
6.1.1 最小点覆盖 . . . 299
6.1.2 最大流 + 二分 . . . 301
6.1.3 最大流路径输出 . . . 304
6.1.4 最大权闭合子图 . . . 306
6.1.5 最小路径覆盖 . . . 309
6.2 费用流 . . . 312
6.3 线性规划 . . . 314

7 计算几何 **316**
7.1 计算几何板子 . . . 316
7.2 凸包三分 . . . 320
7.3 锐角三角形计数 . . . 322
7.4 点与凸包的切线 . . . 323
7.5 动态维护上凸壳 . . . 325
7.6 欧拉平面图公式 . . . 326

8 其他 **328**
8.1 三分 . . . 328
8.2 bitset 优化暴力 . . . 328
8.3 IDA* . . . 330
8.4 左右手路径 . . . 331
8.5 圆周率小数点后第 n 位 . . . 332
8.6 BM 板子 . . . 333
8.6.1 BM 板子 . . . 333
8.6.2 BM 递推式 . . . 335
8.7 高精度 . . . 336
8.7.1 高精度 . . . 336

8.7.2 大数模板 337
8.8 二叉树 338
8.9 全排列 338

0 常规操作

```cpp
#include <bits/stdc++.h>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define per(i,a,b) for(int i = a; i >= b; i--)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const int N = 1e5+100;
const int M = 1e5+100;
const db EPS = 1e-9;
using namespace std;

void dbg() {cout << "\n";}
template<typename T, typename... A> void dbg(T a, A... x) {cout << a << ' '; dbg(x...);}
#define logs(x...) {cout << #x << " -> "; dbg(x);}

/*快速读入、输出*/
namespace fastIO {
    #define BUF_SIZE 10000000        // 预定义缓冲区大小，读入数据大的话就多开点
    #define LL long long
    //fread->read
    bool IOerror=0;
    inline char nc(){
        static char buf[BUF_SIZE],*p1=buf+BUF_SIZE,*pend=buf+BUF_SIZE;
        if (p1==pend){
            p1=buf; pend=buf+fread(buf,1,BUF_SIZE,stdin);
            if (pend==p1){IOerror=1;return -1;}
        }
        return *p1++;
    }
    inline bool blank(char ch){return ch==' '||ch=='\n'||ch=='\r'||ch=='\t';}
    inline void read(int &x){
        bool sign=0; char ch=nc(); x=0;
        for (;blank(ch);ch=nc());
        if (IOerror)return;
        if (ch=='-')sign=1,ch=nc();
        for (;ch>='0'&&ch<='9';ch=nc())x=x*10+ch-'0';
        if (sign)x=-x;
    }
    inline void read(LL &x){
        bool sign=0; char ch=nc(); x=0;
        for (;blank(ch);ch=nc());
        if (IOerror)return;
        if (ch=='-')sign=1,ch=nc();
        for (;ch>='0'&&ch<='9';ch=nc())x=x*10+ch-'0';
        if (sign)x=-x;
    }
    inline void read(double &x){
        bool sign=0; char ch=nc(); x=0;
        for (;blank(ch);ch=nc());
        if (IOerror)return;
        if (ch=='-')sign=1,ch=nc();
```

```
        for (;ch>='0'&&ch<='9';ch=nc())x=x*10+ch-'0';
        if (ch=='.'){
            double tmp=1; ch=nc();
            for (;ch>='0'&&ch<='9';ch=nc())tmp/=10.0,x+=tmp*(ch-'0');
        }
        if (sign)x=-x;
    }
    inline void read(char *s){
        char ch=nc();
        for (;blank(ch);ch=nc());
        if (IOerror)return;
        for (;!blank(ch)&&!IOerror;ch=nc())*s++=ch;
        *s=0;
    }
    inline void read(char &c){
        for (c=nc();blank(c);c=nc());
        if (IOerror){c=-1;return;}
    }
    #undef LL
    #undef BUF_SIZE
};
using namespace fastIO;

inline void write(int x)
{
  static const int maxlen = 100;
  static char s[maxlen];
    if(x < 0) {putchar('-');x=-x;}
  if(!x){putchar('0'); return;}
  int len = 0; for(;x;x/=10) s[len++] = x % 10+'0';
  for(int i = len-1; i >= 0; --i) putchar(s[i]);
}

int n; read(n); wirte(n); printf("\n");
/*-------------------------------------------------------------*/

/*疯狂加速*/
#pragma GCC optimize(2)
#pragma GCC optimize(3)
#pragma GCC optimize("Ofast")
#pragma GCC optimize("inline")
/*-------------------------------------------------------------*/

/*__int128快速读入、输出*/
inline __int128 read() {
    __int128 x=0,f=1;
    char ch=getchar();
    while(ch<'0'||ch>'9') {
        if(ch=='-') f=-1;
        ch=getchar();
    }
    while(ch>='0'&&ch<='9') {
        x=x*10+ch-'0';
        ch=getchar();
    }
    return x*f;
}

inline void write(__int128 x) {
```

```
    if(x<0) { putchar('-'); x=-x; }
    if(x>9) write(x/10);
    putchar(x%10+'0');
}

example = read();
write(example);
/*-----------------------------------------------------------*/

/*离散化*/
sort(base+1,base+1+n);

int tot = unique(base+1,base+1+n)-base-1;

int find(ll x){
    return lower_bound(base+1,base+1+tot,x)-base;
}
/*-----------------------------------------------------------*/


/*生成随机数*/
#include <ctime>
srand(time(0));
int x = rand()%((int)1000)+1; //1~1000范围

#include<random>
#include<ctime>
std::mt19937 rnd(time(0));
printf("%lld\n",rnd()); //周期长度为2^19937-1, 比rand()优秀很多
/*-----------------------------------------------------------*/


/*浮点数取模*/
void FLoat_Mod(long double a, long double b){
    printf("%.9Lf\n",a-b*(long long)(a/b));
}
/*-----------------------------------------------------------*/

/* 相关于char[]的输入输出 */
  char s[105];
  scanf("%s", s); // 读入一个字符串，且首指针放到s+0
  scanf("%s%s%s", s+1, s+21, s+41); // 读入三个字符串，且首指针分贝放到s+1、s+21、s+41，注意本样例若前两个字符串长度超过19，就会在内存有重叠而造成使用出错
  scanf("\n%[^\n]%*c",s); // 读入一行(第一种形式)，首指针为s+0。也可以换为"%[^\n]%*c"，%*c表示忽略后一个字符
  //注意此种读法会忽略每行开始的空格
  //^表示非，[^/n]表示读入换行字符就结束读入，%*c表示该输入项读入后不赋予任何变量
  scanf("%[^\n]", s); getchar(); // 读入一行(第二种形式)，首指针为s+0。getchar()表示把\n取掉
  printf("%s %s", s+0, s+1); // 分别表示从第0、1位开始输出，直到遇到'\0'，所以想截断输出一个字符串还可以将字符串的某个位置置为'\0'即可输出前半部分
/*-----------------------------------------------------------*/

/* scanf */
  scanf("%*[ ]%[^/]/%d",t[i].s,&t[i].a); //%*[]表示越过[]中的字符，%[a-z]表示读入字符串，直到遇到不是a-z中的字符为止
  //%[^a]表示读入字符串知道遇到字符a为止，但a并没有被读入
  getline(cin,s); //s为一个string，推荐使用 getline 进行整行读入
/*-----------------------------------------------------------*/

```

```
/* 相关于string的输入输出 */
  string s;s.resize(100); // 想用scanf读string必须预先设置大小
  scanf("%s", &s[0]); // 读入到string
  scanf("\n%[^\n]%*c", &s[0]); // 读入1行到string
  char tmp[100];
  scanf("%s", tmp+1); s=tmp+1; // 先读入到char[]再令string=char[]
  printf("%s", s.c_str()); // 输出string
/*-------------------------------------------------------------*/


/*map中存放string*/
  mp[s.c_str()] //最好传入string形式，避免发生错误
  std::unordered_map<int,int> tmp; mp.swap(tmp); //设置一个空的map用于交换，即mp.clear()
  mp.reserve(1000); //预留map中元素个数
  mp.rehash(1000); //给元素个数预留足够的bucket用于hash
  /* unordered_map 操作 */
  #include <unordered_map>
  struct Node {
    int x, y;
    bool operator == (const Node &t) const {
      return  x==t.x && y==t.y;
    }
  };
  struct NodeHash {
    std::size_t operator () (const Node &t) const {
      return t.x * 100 + t.y;
    }
  };
  unordered_map<Node, string, NodeHash> h_map;
  h_map[(Node){1,2}] = "World";
  /* unordered_set */
  #include <unordered_set>
  unordered_set<Node, NodeHash> h_set;
  h_set.insert((Node){1,2});
  if(h_set.find((Node){1,2} == h_set.end()) printf("Not found!\n");
  else printf("Found!");
/*-------------------------------------------------------------*/

/*string函数*/
  /* 数字转字符串 */
  strings s = to_string(2323.232);
  /* 取长度 */
  int sz = s.size();
  /* 取下标字符 */
  char c = s[2];
  /* 截取字符串 */
  string ss = s.substr(2, 3);  // 从第2个字符开始截取长度为3的串
  /* 查找串 */
  size_t found = s.find("23", 0); // 从位置0开始找到第1个串"23"，找到返回下标，否则found=
    string::npos
  size_t found = s.rfind("23", s.size()-1); // 从右侧开始找到第1个串"23"，至多找到至pos
  size_t可以强转为int, int pos = (int)s.find("23",0);
  int pos = (int)s.find(':');  //string也可以直接查找字符
  /* 拼接 */
  s += "aaaa";
/*-------------------------------------------------------------*/

/*位运算*/
  /*unsigned int x*/
```

```
  x = 433 -> 110110001
  /*计算 x 的二进制中有多少个 1*/
  __builtin_popcount(x) -> 5
  /*判断 x 的二进制中 1 的个数的奇偶性*/
  __builtin_parity(x) -> 1
  /*计算 n 的二进制末尾最后一个 1 的位置，位置编号从 1 开始*/
  __builtin_ffs(x) -> 1
  /*计算 n 的二进制末尾连续 0 的个数*/
  __builtin_ctz(x) -> 0
  /*返回前导 0 的个数*/
  __builtin_clz(x) -> 23 = 32 - 9
  /*计算 n 的二进制开头第一个 1 的位置，位置编号从 1 开始*/
  32 - __builtin_clz(x) = 9
  /*在函数名末尾添加 ll 使参数类型变为 unsigned long long*/
  __builtin_clzll(x) -> 55
  /*注意：上述均为内建函数，经过了编译器的高度优化，运行速度十分快*/
/*-------------------------------------------------------------*/

/*bitset函数*/
  /* 声明 */
  bitset<16> a;           // 空，全零 0000000000000000
  bitset<16> b(0x3fff);   // 整数参数 0011111111111111
  bitset<16> c("00101");  // 字符串   0000000000000101

  /* 赋值 */
  a[0] = 1;    // 低位第0位设为1
  a.set(0, 1); // 低位第0位设为1
  a.set(0);    // 低位第0位设为1
  a.set(0, 0); // 低位第0位设为0
  a.set();     // 设为全1
  a.reset();   // 设为全0，该函数同样有两个参数，参数1为pos，参数2位value(为空时是0)
  a.flip(1);   // 翻转第1位
  a.flip();    // 翻转整个01串
  a<<=1;       // a左移1位

  /* 遍历 */
  cout << a << endl;                      // cout输出整个01串
  printf("%s\n", a.to_string().c_str()); // printf输出整个01串，较为麻烦一些
  cout << a[0] << endl;                   // cout输出低位第0位
  printf("%d\n", a[0]==1);                // printf输出低位第0位

  /* 判位、换型 */
  a.count();              //统计1的个数
  a = a1 & a2;            //按位与
  a = a1 | a2;            //按位或
  a = a1 ^ a2;            //按位异或
  a = ~a1;              //按位补
  a = a1 << 3;            //移位
  a.text(0);                            // 判断第0位是否为1
  a.any();                              // 判断是否至少1位为1
  s.none();                             // 判断是否全0
  int one = a.count;                    // 获得1的个数
  string s = a.to_string();             // 转换为字符串
  unsigned long = a.to_ulong();         // 转换为无符号整形
  unsigned long long = a.to_ullong(); // 转换为无符号整形
/*-------------------------------------------------------------*/


/*vector*/
```

```
  /* 声明 */
  vector<int> v;
  /* 排序 */
  sort(v.begin(),v.end());
  /* 遍历方式一（推荐）*/
  for(auto &tp : v){}
  /* 遍历方式二（不推荐）*/
  if(v.size()){ //一定要确保v不为空，否则会RE
    rep(i,0,v.size()-1){}
  }
  /* resize() 和 reserve() 函数 */
  v.reserve(100);  //改变了vector的capacity，但是没有改变其空间大小
  v.resize(100);   //同时改变了vector的capacity和size，且原有元素不会被覆盖
  /* vector 去重 */
  sort(v.begin(),v.end());
  v.erase(unique(v.begin(),v.end()),v.end());
/*-----------------------------------------------------------------*/


/*set函数*/
  /* 声明 */
  set<int> s;
  /* 赋值 */
  s.insert(1);
  s.insert(2);
  /* 遍历 */
  if(s.find(1) != s.end()) printf("had");
  for(set<int>::iterator it=s.begin();it!=s.end();it++) printf("%d", *it); // 全部遍历
  for(auto &x : s) printf("%d\n", x);
  int sz = s.size(); // 大小
  /* 清空 */
  s.clear();
  s.erase(x); //x为s中存的内容
  *(s.begin()); //取出set中首位元素
  multiset<int>::iterator it1 = st.begin(), it2 = st.end();
  it2--; //取出set/multiset首尾元素
  set<pair<int,int> >::iterator it = st1.lower_bound(make_pair(-pos,0)); //it是指针
  lower_bound: 找到第一个大于等于这个数的值，若要找第一个小于的可以考虑往set里丢负值
  /* multiset删除 */
  st.erase(st.find(x)) //只删一个x
  st.erase(x) //删除所有x
/*-----------------------------------------------------------*/


/*regex分割*/
  regex re("[\\.,!\\? ]"); //全局变量，.与?需要转义，此处分割的内容有'.'、','、'!'、'?'、' '
  //分割之后，最后的一个字符串可能会含有大量空格
  /* 切割单词后放入一个vector */
  vector<string> ans{
      sregex_token_iterator(yy.begin()+pos+1, yy.end(), re, -1),
      sregex_token_iterator()
  };
  for(auto &it: ans) printf("%s\n",it.c_str());
/*-----------------------------------------------------------*/

/*线段树节点编号*/
inline int get_id(int l,int r) {return (l+r)|(l!=r);}
/*-----------------------------------------------------------*/

```

```cpp
/*二进制中01操作*/
template<class T> int getbit(T s, int i) { return (s >> i) & 1; }
template<class T> T onbit(T s, int i) { return s | (T(1) << i); }
template<class T> T offbit(T s, int i) { return s & (~(T(1) << i)); }
template<class T> int cntbit(T s) { return __builtin_popcount(s);}
/*-----------------------------------------------------------*/

/*运行时间*/
  #include <ctime> //程序开头
  clock_t startTime,endTime;
  startTime = clock(); //计时开始

  //程序末尾
  endTime = clock();//计时结束
  cout << "The run time is: " <<(double)(endTime - startTime) / CLOCKS_PER_SEC << "s"
    << endl;
/*-----------------------------------------------------------*/
```

1 数学

1.1 快速幂与快速乘

1.1.1 快速幂

```
//快速幂
#include <cstdio>
#include <iostream>
typedef long long LL;
using namespace std;

LL poww(LL a,LL b)
{
    LL base = a,ans = 1;
    while(b!=0)
    {
        if(b&1)
            ans *= base;
        base *= base;
        b >>= 1;
    }
    return ans;
}

LL poww(LL a,LL b,LL p)
{
    LL base = a%p,ans = 1;
    while(b!=0)
    {
        if(b&1)
            ans = (ans*base)%p;
        base = (base*base)%p;
        b >>= 1;
    }
    return (ans%p);
}

int main()
{
    LL h = poww(11,11);
    cout<<h<<endl;
    return 0;
}
```

1.1.2 矩阵快速幂

```
#include <cstdio>
#include <iostream>
#include <cstring>
using namespace std;
const int mod = 9973;
typedef long long ll;

int n,m,t;
//i、k、j只能作为临时变量
struct Matrix{
    ll a[35][35];
    Matrix(){memset(a,0,sizeof(a));}
```

```

    Matrix operator * (const Matrix y)
    {
        Matrix ans;
        for(int i = 1;i <= n;i++)   //行
            for(int j = 1;j <= n;j++)  //列
                for(int k = 1;k <= n;k++)
                    ans.a[i][j] = (ans.a[i][j]+a[i][k]*y.a[k][j])%mod;
        return ans;
    }

    Matrix operator + (const Matrix y)
    {
        Matrix ans;
        for(int i = 1;i <= n;i++)
            for(int j = 1;j <= n;j++)
                ans.a[i][j] = (a[i][j]+y.a[i][j])%mod;
        return ans;
    }
};

Matrix q_pow(Matrix x,int k)
{
    Matrix ans;
    for(int i = 1;i <= n;i++)   ans.a[i][i] = 1;
    while(k)
    {
        if(k&1)
            ans = ans*x;
        x = x*x;
        k = k>>1;
    }
    return ans;
}

int main()
{
    cin>>t;
    while(t--)
    {
        scanf("%d%d",&n,&m);
        Matrix p;
        for(int i = 1;i <= n;i++)
            for(int j = 1;j <= n;j++)
                scanf("%lld",&p.a[i][j]);
        Matrix ans = q_pow(p,m);
        long long ans1 = 0;
        for(int i = 1;i <= n;i++)
            ans1 = (ans.a[i][i]+ans1)%mod;
        printf("%lld\n",ans1);
    }
    return 0;
}
```

1.1.3 快速乘取膜

方法一:
a b p 范围都在 1e18

求 a*b 的结果
为了不让结果爆 ll，则将 a 进行二进制分解，然后递推 *b 即可

方法二：
用 long double 存数, long double 的有效数字有 18-19 位

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#define rep(i,a,b) for(int i = a; i <= b; i++)
using namespace std;
typedef long long ll;

ll mul(ll a,ll b,ll p)
{
    a %= p, b %= p;
    long long c = (long double)a*b/p;
    long long ans = a*b-c*p;
    //此处会溢出，但是由于二者相差的部分不会太大，所以前面的数字都是一样的
    //因此溢出不但不影响答案，还是符合我们的要求的
    if(ans < 0) ans+=p;
    else if(ans >= p) ans-=p;
    return ans;
}

int main()
{
    ll a,b,p;
    while(~scanf("%lld%lld%lld",&a,&b,&p))
    {
        printf("%lld\n",mul(a,b,p));
    }
    return 0;
}
```

1.2 筛

1.2.1 埃氏筛

```
#include <cstdio>
#include <iostream>
#include <algorithm>
#include <cstring>
#define rep(i,a,b) for(int i = a;i <= b;i++)
using namespace std;
const int N = 1.5*1e7+1000;

int b[N],tot,prime[N];

int main()
{
    tot = 0; //素数个数

    //埃氏筛
    for(int i = 2; i <= 1e5; i++)
    {
        if(b[i] == 0)
```

```
            prime[++tot] = i; //哪几个是素数
        else continue;
        int base = i;
        while(base <= 1e5){
            base += i;
            b[base] = 1; //这个数是不是素数，为1则非素数
        }
    }
    return 0;
}
```

1.2.2 线性筛

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#define rep(i,a,b) for(int i = a; i <= b; i++)
using namespace std;
const int N = 1e7+1000;

int v[N],prime[N],tot;  //v[i]: i的最小质因子

void primes(int x)  //线性筛
{
    memset(v,0,sizeof v);
    tot = 0;
    rep(i,2,x){
        if(!v[i]) v[i] = i, prime[++tot] = i;
        rep(j,1,tot)
            if(prime[j] > v[i] || prime[j] > x/i) break;
            else v[i*prime[j]] = prime[j];
    }
}

int main()
{
    int n = 1e5;
    primes(n);
    rep(i,1,tot) printf("%d\n",prime[i]);
    return 0;
}
```

1.2.3 杜教筛

给定一个 n ($n \le 2^31-1$), 求 ans_1 和 ans_2

$ans_1 = \sum_{i=1}^{n} \varphi(i)$

$ans_2 = \sum_{i=1}^{n} \mu(i)$

```
#include<bits/stdc++.h>
#include<tr1/unordered_map>
#define N 6000010
using namespace std;
template<typename T>inline void read(T &x)
{
```

```
    x=0;
    static int p;p=1;
    static char c;c=getchar();
    while(!isdigit(c)){if(c=='-')p=-1;c=getchar();}
    while(isdigit(c)) {x=(x<<1)+(x<<3)+(c-48);c=getchar();}
    x*=p;
}
bool vis[N];
int mu[N],sum1[N],phi[N];
long long sum2[N];
int cnt,prim[N];
tr1::unordered_map<long long,long long>w1;
tr1::unordered_map<int,int>w;
void get(int maxn)
{
    phi[1]=mu[1]=1;
    for(int i=2;i<=maxn;i++)
    {
        if(!vis[i])
        {
            prim[++cnt]=i;
            mu[i]=-1;phi[i]=i-1;
        }
        for(int j=1;j<=cnt&&prim[j]*i<=maxn;j++)
        {
            vis[i*prim[j]]=1;
            if(i%prim[j]==0)
            {
                phi[i*prim[j]]=phi[i]*prim[j];
                break;
            }
            else mu[i*prim[j]]=-mu[i],phi[i*prim[j]]=phi[i]*(prim[j]-1);
        }
    }
    for(int i=1;i<=maxn;i++)sum1[i]=sum1[i-1]+mu[i],sum2[i]=sum2[i-1]+phi[i];
}
int djsmu(int x)
{
    if(x<=6000000)return sum1[x];
    if(w[x])return w[x];
    int ans=1;
    for(int l=2,r;l>=0&&l<=x;l=r+1)
    {
        r=x/(x/l);
        ans-=(r-l+1)*djsmu(x/l);
    }
    return w[x]=ans;
}
long long djsphi(long long x)
{
    if(x<=6000000)return sum2[x];
    if(w1[x])return w1[x];
    long long ans=x*(x+1)/2;
    for(long long l=2,r;l<=x;l=r+1)
    {
        r=x/(x/l);
        ans-=(r-l+1)*djsphi(x/l);
    }
    return w1[x]=ans;
```

```
}
int main()
{
    int t,n;
    read(t);
    get(6000000);
    while(t--)
    {
        read(n);
        printf("%lld %d\n",djsphi(n),djsmu(n));
    }
    return 0;
}
```

1.3 GCD 与逆元

1.3.1 GCD

```
typedef long long ll;

ll gcd(ll a,ll b)
{
    return b == 0 ? a:gcd(b,a%b);
}
```

1.3.2 Exgcd 与逆元

```
#include <cstdio>
#include <iostream>
#include <algorithm>
#include <cstring>
#define rep(i,a,b) for(int i = a;i <= b;i++)
using namespace std;
typedef long long ll;
const ll mod = 9973;

//求ax+by = gcd(a,b)的特解
ll exgcd(ll a,ll b,ll &x,ll &y)
{
    if(b == 0) {x = 1,y = 0; return a;}  //此处a为gcd(a,b)
    ll d = exgcd(b,a%b,x,y);
    ll z = x; x = y; y = z-y*(a/b);
    return d;
}
// ax+by = c的通解可以表示为 x = (c/d)x1+k*(b/d) 【d为gcd(a,b)】
//                           y = (c/d)y1-k*(a/d) 【k为正整数】

//调用mod_reverse就是求逆元，求a在%n意义下的逆元
ll mod_reverse(ll a,ll n)
{
    ll x,y;      //求a的逆元，a*ai 与 1 % mi同余，则a*ai-mi*k = 1；a与mi已知，求ai
    ll d = exgcd(a,n,x,y);
    if(d == 1) return (x%n+n)%n;  //保证逆元为正
    else return -1;
}
```

费马小定理求逆元
1. 模数 p 为素数

$a^{-1} = a^{p-2}$
2. 模数 p 不为素数
$a^{-1} = a^{\phi(p)-1}$

```

int main()
{
    ll x;
    while(~scanf("%lld",&x))
    {
        printf("%lld\n",mod_reverse(x,mod));
    }
    return 0;
}
```

1.3.3 O(n) 打表求逆元

```
#include <cstdio>
#include <cmath>
using namespace std;
typedef  long long ll;
const int N = 1e5 + 5;

int inv[N];

void inverse(int n, int p) { //O(n)求1-n所有逆元
    inv[1] = 1;
    for (int i=2; i<=n; ++i) {
        inv[i] = (ll) (p - p / i) * inv[p%i] % p;
    }
}
```

1.4 唯一分解

```
#include <cstdio>
#include <iostream>
#include <algorithm>
#include <cstring>
#define rep(i,a,b) for(int i = a;i <= b;i++)
using namespace std;
const int N = 1.5*1e7+1000;

int b[N],c[N][2]; //c[i][0] —— 表示第 i 个唯一分解数
//c[i][1] —— 表示第 i 个唯一分解数的指数
int prime[N],tot;
int n;

//唯一分解
void getFactors(ll x)
{
    ll tmp = x;
    for(ll i = 1; prime[i]*prime[i] <= tmp; i++)
    {
        if(tmp%prime[i] == 0)
        {
            c[++tot][0] = prime[i];
            while(tmp%prime[i] == 0){
```

```
                tmp /= prime[i];
                c[tot][1]++;
            }
        }
    }
    if(tmp != 1){
        c[++tot][0] = tmp;
        c[tot][1] = 1;
    }
}

int main()
{
    scanf("%d",&n);
    tot = 0;

    //埃氏筛
    for(int i = 2; i <= 1e5; i++)
    {
        if(b[i] == 0)
            prime[++tot] = i;
        else continue;
        int base = i;
        while(base <= 1e5){
            base += i;
            b[base] = 1;
        }
    }
    rep(i,1,n)
    {
        int x;
        scanf("%d",&x);
        getFactors(x);
    }
    return 0;
}
```

1.5 中国剩余定理

1.5.1 中国剩余定理拓展版

```
//拓展中国剩余定理 —— m1,m2,m3之间不互质
#include <cstdio>
#define ll long long
// #define ll __int128
// 如果 int128 也爆了的话，建议使用 python
const int MAXN = 2e6+10;
using namespace std;
ll K, C[MAXN], M[MAXN];

ll gcd(ll a, ll b) {
    return b == 0 ? a : gcd(b, a % b);
}
ll exgcd(ll a, ll b, ll &x, ll &y) {
    if (b == 0) {x = 1, y = 0; return a;}
    ll r = exgcd(b, a % b, x, y), tmp;
    tmp = x; x = y; y = tmp - (a / b) * y;
    return r;
}
```

```
ll inv(ll a, ll b) {
    ll x = 0, y = 0;
    ll r = exgcd(a, b, x, y);
    while (x < 0) x += b;
    return x;
}
ll excrt(int n, ll M[], ll C[]){ // % m = c
    for (int i = 2; i <= n; i++) {
        ll M1 = M[i - 1], M2 = M[i], C2 = C[i], C1 = C[i - 1], T = gcd(M1, M2);
        if ((C2 - C1) % T != 0) return -1;
        M[i] = (M1 * M2) / T;
        C[i] = ( inv( M1 / T , M2 / T ) * (C2 - C1) / T ) % (M2 / T) * M1 + C1;
        C[i] = (C[i] % M[i] + M[i]) % M[i];
    }
    return C[n];
}

int main() {
    while (~scanf("%lld", &K)) {
        for (int i = 1; i <= K; i++) scanf("%lld%lld", &M[i], &C[i]);
        printf("%lld\n", excrt(K, M, C));
    }
    return 0;
}
```

1.5.2 中国剩余定理 python 版本

```
def gcd(a, b):
    if b == 0:
        return a
    else:
        return gcd(b, a % b)

def exgcd(a, b, x, y):
    re, tmp = 0, 0
    if b == 0:
        return a, 1, 0
    re, x, y = exgcd(b, a % b, x, y)
    tmp = x
    x = y
    y = tmp - (a//b)*y
    return re, x, y


def inv(a, b):
    x, y, r = 0, 0, 0
    r, x, y = exgcd(a, b, x, y)
    while x < 0:
        x += b
    return x


def excrt(n, M, C):
    M1, M2, C1, C2, T = 0, 0, 0, 0, 0
    for i in range(2, n+1):
        M1, M2, C1, C2 = M[i-1], M[i], C[i-1], C[i]
        T = gcd(M1, M2)
        if (C2-C1) % T != 0:
```

```
            return 1e13
        M[i] = (M1 * M2) // T
        C[i] = (inv(M1//T, M2//T) * (C2-C1) // T) % (M2 // T) * M1 + C1
        C[i] = (mod[i] % M[i] + M[i]) % M[i]
        if C[i] > 1e12:
            return 1e13
    return C[n]
```

1.6 求组合数

1.6.1 Lucas (模数为质数)

```
#include <cstdio>
#include <iostream>
#include <algorithm>
#include <cstring>
#define rep(i,a,b) for(int i = a;i <= b;i++)
using namespace std;
typedef long long ll;

//求ax+by = gcd(a,b)的特解
ll exgcd(ll a,ll b,ll &x,ll &y)
{
    if(b == 0) {x = 1,y = 0; return a;}  //此处a为gcd(a,b)
    ll d = exgcd(b,a%b,x,y);
    ll z = x; x = y; y = z-y*(a/b);
    return d;
}
// ax+by = c的通解可以表示为 x = (c/d)x1+k*(b/a) 【d为gcd(a,b)】
//                           y = (c/d)y1-k*(a/d) 【k为正整数】

//调用mod_reverse就是求逆元，求a在%p意义下的逆元
ll mod_reverse(ll a,ll p)
{
    ll x,y;      //求a的逆元，a*ai 与 1 % mi同余，则a*ai-mi*k = 1；a与mi已知，求ai
    ll d = exgcd(a,p,x,y);
    if(d == 1) return (x%p+p)%p;  //保证逆元为正
    else return -1;
}

ll fact(ll n, ll p){//n的阶乘 % p
    ll ret = 1;
    for (int i = 1; i <= n ; i ++) ret = ret * i % p ;
    return ret ;
}

ll comb(ll n, ll m, ll p){//C(n, m) % p，将n和m限制在1-p之内
    if (m < 0 || m > n) return 0;
    //调用逆元和求解阶乘
    return fact(n, p) * mod_reverse(fact(m, p), p) % p * mod_reverse(fact(n-m, p), p) %
     p;
}

ll Lucas(ll n, ll m, ll p)  //求组合数C(n,m)%p
{
    return m ? Lucas(n/p, m/p, p) * comb(n%p, m%p, p) % p : 1;
}
```

```
int main()
{
/*注意事项:
    C(n,m)%p用Lucas定理求解，p必须为质数，并且p < 1e5，这个限制主要是针对如果p大于1e5，那就算n和m在1-p之内，如果n和m大于1e5，也难以求解
    本模板还可以进一步进行加速，那就是预处理每个数的阶乘
*/
    ll n,m,p;
    int T;
    scanf("%d",&T);
    while(T--)
    {
        scanf("%lld%lld%lld",&n,&m,&p);
        printf("%lld\n",Lucas(n+m,m,p));
    }
    return 0;
}
```

1.6.2 递推求组合数

$C(n,m) = C(n-1,m-1) + C(n-1,m)$

```
C[1][0] = C[1][1] = 1;

for (int i = 2; i < N; i++){
    C[i][0] = 1;
    for (int j = 1; j < N; j++)
        C[i][j] = (C[i - 1][j] + C[i - 1][j - 1]);
}
```

1.7 线性基

1.7.1 线性基模板

一、定义
1. 设数集 T 为一组数任意子集异或得到的集合。
2. T 的线性基是 T 的一个子集 $A = a_1, a_2, a_3, ..., a_n$。$A$ 中元素互相 xor 所形成的异或集合，等价于原数集 T 的元素互相 xor 形成的异或集合。可以理解为将原数集进行了压缩。

二、性质
1. 线性基没有异或为 0 的子集。
2. 线性基的异或集合中每个元素的异或方案唯一，其实这个跟性质 1 是等价的。
3. 线性基二进制最高位互不相同。
4. 线性基中元素互相异或，异或集合不变。

三. 求 x 在 Q 中的排名 (允许重复)
1. 共有 cnt 个基底, 则每个数出现的次数均为 $2^{n-cnt}-1$。因为没被选入基底的数有 $n-cnt$ 个, 其能用基底唯一表示, 再把基底对应选中异或即为 0 向量, 因此共有 $2^{n-cnt}-1$ 个零向量。
2. x 由第 $a_0, ..., a_i, ..., a_m$ 个向量异或而来, 没有参与的向量置 0, 令 $P = a_0, ..., a_i ..., a_m$ 的二进制表示。因此如果 0 可取, 则 x 排名为 $(P-1)*(2^{n-cnt}-1)+1$; 如果 0 不可取, 则 x 排名为 P。

四. 求 T 集合中第 i 位为 1 的数的个数 H
T 集合大小为 $2^{|Q|}$, 令 $ALL| = p[i], i \in [0, 62]$。若 ALL 第 i 位为 1，则 $H = 2^{|Q|-1}$, 否则 $H = 0$。

```
struct Linear_Basis{
    ll p[65],d[65];
```

```
    int cnt;
    Linear_Basis() {memset(p,0,sizeof p);}
    ~Linear_Basis() {}
    bool ins(ll x){ //向线性基中插入一个数
        for(int i = 62; i >= 0; i--){
            if(x&(1ll<<i)){
                if(!p[i]) {p[i] = x; break;}
                x ^= p[i];
            }
        }
        return x > 0ll;
    }
    ll MAX(ll x){ //求线性空间与ans异或的最大值
        for(int i = 62; i >= 0; i--)
            if((x^p[i]) > x) x ^= p[i];
        return x;
    }
    //如果是求一个数与线性基的异或最小值，则需要先rebuild，再从高位向低位依次进行异或
    ll MIN(){
        rep(i,0,62)
            if(p[i]) return p[i];
    }
    //将线性基改造成每一位相互独立，即对于二进制的某一位i，只有pi的这一位是1，其它都是0
    //注意此处只是避免多个基位上同时有第i位，但是可能会出现p[5]同时有第5位和第3位，但是p[3]为空
    void rebuild(){
        cnt = 0;
        for(int i = 62; i >= 0; i--)
            for(int j = i-1; j >= 0; j--)
                if(p[i]&(1ll<<j))
                    p[i]^=p[j];
        rep(i,0,62)
            if(p[i]) d[cnt++] = p[i];
    }
    //求线性基能够组成的数中的第K大
    ll Kth(ll k){
        ll ret = 0;
        if(k >= (1ll<<cnt)) return -1; //k大于子集总数，找不到
        for(int i = 62; i >= 0; i--)
            if(k&(1ll<<i)) ret ^= d[i];
        return ret;
    }
    //合并两个线性基
    Linear_Basis& merge(const Linear_Basis &xx){
        for(int i = 62; i >= 0; i--)
            if(xx.p[i]) ins(xx.p[i]);
        return *this;
    }
}LB;

//两个线性基求交
Linear_Basis merge(Linear_Basis a, Linear_Basis b){
    Linear_Basis A = a, tmp = a, ans; //tmp不断构建A+(B\ans)
    ll cur,d;
    rep(i,0,33) //从低到高，使得不存在一个基底可以仅由(tmp\A)表示
        if(b.p[i]){ //b中有这个基底
            cur = 0, d = b.p[i];
            per(j,i,0)
                if((d>>j)&1){
                    if(tmp.p[j]){
```

```
                        d ^= tmp.p[j], cur ^= A.p[j];
                        if(d) continue;
                        ans.p[i] = cur; //cur的第i位不为0
                    }
                    else tmp.p[j] = d, A.p[j] = cur;
                    //如果不能被表示, A的赋值是为了让高位中含有j这位的基底下放到A中j的位置
                    break;
                }
        }
    return ans;
}
```

1.7.2 无向图异或路径问题

题意: 给出一个 n 个点，m 条边的无向图。定义一个三元组 (u, v, s)，表示 u 到 v 的一条路径异或和为 s，该三元组对答案的贡献为 s。求出所有三元组的贡献和。$(1 \le n \le 10^5, 0 \le m \le 2 * 10^5)$

思路:
直接求答案显然不现实，而由于这是一道异或问题，因此考虑对每一个数位求贡献。首先对连通分量求一个 dfs 树，即求出从根节点到每一个点的异或和，因此两点路径异或和就等于两个点的异或和再异或。dfs 时再求出连通分量中所有环的异或和，将异或和插入线性基中。然后依次统计每一个数位对答案的贡献。

对于数位 i 来说，该数位统计到答案中只有两种情况。第一种情况，两点异或和在数位 i 处为 1，则求出线性基数集合中所有第 i 位为 0 的数个数。

第二种情况，两点异或和在数位 i 处为 0，则求出线性基数集合中所有第 i 位为 1 的数个数。

将两种情况分别统计累加到答案中，即可完成此题。此处需要注意，判断第 i 位为 1 的数能否由线性基组成，只需要将线性基中的 $p[i]$ 依次取或得到 ALL，若 ALL 中第 i 位为 1，则可以构造，否则不可。

```
#include <bits/stdc++.h>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl
    ;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2
    << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const int N = 1e5+100;
const int M = 4*1e5+100;
const ll mod = 1e9+7;
const db EPS = 1e-9;
using namespace std;

int n,m,head[N],tot,vis[N];
ll rt[65],dis[N];
vector<ll> ve;
struct Node{
    int to,next;
    ll w;
}e[M];
ll ans = 0;

void add(int x,int y,ll w){
    e[++tot].to = y, e[tot].next = head[x], head[x] = tot, e[tot].w = w;
```

```
}

struct Linear_Basis{
    ll p[65],cnt;
    Linear_Basis() {memset(p,0,sizeof p); cnt = 0;}
    ~Linear_Basis() {}
    void init(){
        memset(p,0,sizeof p);
        cnt = 0;
    }
    bool ins(ll x){
        // LOG1("x",x);
        for(int i = 62; i >= 0; i--){
            if(x&(1ll<<i)){
                if(!p[i]) {p[i]=x; cnt++; break;}
                x ^= p[i];
            }
        }
        return x > 0;
    }
    void rebuild(){
        cnt = 0;
        for(int i = 62; i >= 0; i--)
            for(int j = i-1; j >= 0; j--)
                if(p[i]&(1ll<<j)) p[i]^=p[j];
        rep(i,0,62)
            if(p[i]) cnt++;
    }
}LB;

void dfs(int x,ll tp){
    if(vis[x]) return;
    vis[x] = 1; ve.push_back(tp); dis[x] = tp;
    for(int i = head[x]; i; i = e[i].next){
        int y = e[i].to;
        if(vis[y]){
            ll xp = tp^e[i].w^dis[y];
            // LOG2("y",y,"xp",xp);
            LB.ins(xp);
            continue;
        }
        dfs(y,tp^e[i].w);
    }
}

int main()
{
    scanf("%d%d",&n,&m); tot = 1;
    rep(i,1,m){
        int a,b; ll c; scanf("%d%d%lld",&a,&b,&c);
        add(a,b,c); add(b,a,c);
    }
    rep(i,1,n)
        if(!vis[i]){
            LB.init(); ve.clear();
            dfs(i,0);
            ll thp = (ll)ve.size();
            // LOG1("thp",thp);
            memset(rt,0,sizeof rt);
```

```
            for(auto &v:ve){
                int tp = 0; ll hp = v;
                while(hp){
                    if(hp&1) rt[tp]++;
                    tp++; hp /= 2ll;
                }
            }
            LB.rebuild();
            ll Total = 1;
            rep(H,1,LB.cnt) Total = (Total*2ll)%mod;
            ll ALL = 0;
            rep(H,0,62) ALL |= LB.p[H];
            // LOG1("cnt",LB.cnt);
            rep(k,0,62){
                ll hp1 = rt[k], hp2 = thp-rt[k]; //有和没有
                ll tp = 0;
                if(ALL&(1ll<<k)){
                    tp = 1;
                    rep(H,1,LB.cnt-1) tp = (tp*2ll)%mod;
                }
                // LOG3("Total",Total,"k",k,"tp",tp);
                //01、10
                ll now1 = (hp1*(hp1-1ll)/2ll+hp2*(hp2-1ll)/2ll)%mod, now2 = (hp1*hp2)%
    mod;
                ans = (ans+(1ll<<k)%mod*now1%mod*tp%mod)%mod;
                ans = (ans+(1ll<<k)%mod*now2%mod*(Total-tp)%mod)%mod;
            }
        }
    printf("%lld\n",ans);
    return 0;
}
```

1.7.3 异或为 X 的子集长度和

意: 给出一个集合 A，求出 A 中所有异或和为 0 的子集的长度和。$(1 \le n \le 2*10^6, 0 \le a_i \le 10^{18})$

思路: 首先线性基中并没有可以直接求出长度和的方法，因此我们需要考虑其他方法进行求解。

既然无法从整体上进行求解，从单个异或和为 0 的子集入手也不现实，因此考虑最小单元 a_i。即求出每一个 a_i 对答案的贡献。此处思考的入手点在于将题目要求解的值不断细分，从整体到子集再到子集中的一个元素。

对于每一个 a_i 求贡献，问题就简化了很多。首先对集合 A 求线性基，则有 cnt 个元素成功插入，$n-cnt$ 个元素未插入，则该 $n-cnt$ 个元素对答案的贡献为 $(n-cnt)*2^{n-cnt}$。

再对于线性基中的 cnt 个元素求贡献。每次取出 cnt 中的一个元素 a_x，将剩下的 $n-1$ 个元素插入线性基中，新形成的线性基元素个数为 T。若新形成的线性基中无法插入 a_x，则 a_x 对答案对贡献为 2^{n-1-T}。至此本题结束。

总结: 本题中还需涉及从线性基中删除一个元素和合并线性基。合并线性基不用多说，删除一个元素则是记录每一个元素插入线性基的位置，从最后一个元素往回不断删除，每次将插入位置置 0 即可。

本题还可以进行修改，修改为求 A 中所有异或和为 X 的子集长度和。其余地方都无需改动，只需要将" 判断新形成的线性基中是否可以插入 a_x" 修改为" 判断新形成的线性基中是否可以插入 a_x xor X" 即可。

```
#include <bits/stdc++.h>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
```

```
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const int N = 1e5+100;
const int M = 1e5+100;
const ll mod = 1e9+7;
const db EPS = 1e-9;
using namespace std;

int n;
ll a[N],ans;
vector<pair<ll,int> > v;

struct Linear_Basis{
    ll p[65]; int cnt;
    Linear_Basis() {memset(p,0,sizeof p); cnt = 0;}
    ~Linear_Basis() {}
    void init() {memset(p,0,sizeof p); cnt = 0;}
    int ins(ll x){
        int pos = -1;
        for(int i = 62; i >= 0; i--)
            if(x&(1ll<<i)){
                if(!p[i]) {p[i] = x; pos = i; cnt++; break;}
                else x ^= p[i];
            }
        return pos;
    }
    Linear_Basis& merge(Linear_Basis x){
        for(int i = 62; i >= 0; i--)
            if(x.p[i]) ins(x.p[i]);
        return *this;
    }
}LB1,LB2;

ll poww(ll a,ll b){
    ll base = a, tp = 1;
    while(b){
        if(b&1) tp = (tp*base)%mod;
        base = (base*base)%mod;
        b >>= 1;
    }
    return (tp%mod);
}

int main()
{
    while(~scanf("%d",&n))
    {
        ans = 0; LB1.init(); LB2.init(); v.clear();
        rep(i,1,n) scanf("%lld",&a[i]);
        rep(i,1,n){
            int flag = LB1.ins(a[i]);
            if(flag != -1) v.push_back(make_pair(a[i],flag));
            else LB2.ins(a[i]);
        }
        int SIZE = v.size();
```

```
        // LOG1("SIZE",SIZE);
        ll tp = 1;
        if(n-SIZE-1 >= 1) tp = poww(2ll,n-SIZE-1);
        ans = (ans+((ll)n-(ll)SIZE)*tp%mod)%mod;
        for(int i = SIZE-1; i >= 0; i--){
            LB1.p[v[i].second] = 0; LB1.cnt--;
            Linear_Basis LB3 = LB1;
            LB3 = LB3.merge(LB2);
            tp = n-1-LB3.cnt;
            int flag = LB3.ins(v[i].first);
            if(flag == -1){
                ll hp = poww(2ll,tp);
                ans = (ans+hp)%mod;
            }
            LB2.ins(v[i].first);
        }
        printf("%lld\n",ans);
    }
    return 0;
}
```

1.7.4 前缀线性基

题意: 给定 n 个数字，q 个询问，每次询问对应一个 l、r，询问区间 $[l,r]$ 中所有子集异或和的最大值。$(1 \le n \le 5*10^5, 1 \le q \le 5*10^5)$

思路: 处理前缀线性基。$p[i][j]$ 表示前 i 个数的第 j 位的线性基，$pos[i][j]$ 表示前 i 个数的第 j 位的线性基由哪一位数字得到，保留最靠右的位置。
每次插入一个数字，如果可以更新 $pos[i][j]$ 则更新，并将 $p[i][j]$ 替换下来，用于后续的更新。查询 $[l,r]$ 最值只需将 $p[r]$ 取出，所有 $pos[r][j] \ge l$ 均为有效基。

总结: 前缀线性基，维护区间线性基。

```
#include <bits/stdc++.h>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl
    ;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2
    << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const int N = 5*1e5+100;
const int M = 1e5+100;
const db EPS = 1e-9;
using namespace std;

int n,q,p[N][27],pos[N][27],a[N];

int main()
{
    scanf("%d",&n);
    rep(i,1,n) scanf("%d",&a[i]);
    rep(i,1,n){
        rep(k,0,24) p[i][k] = p[i-1][k], pos[i][k] = pos[i-1][k];
        int ti = i;
```

```
        for(int k = 24; k >= 0; k--){
            if(a[i]&(1<<k)){
                if(!p[i][k]) {p[i][k] = a[i]; pos[i][k] = ti; break;}
                if(pos[i][k] < ti) swap(p[i][k],a[i]), swap(pos[i][k],ti);
                a[i] ^= p[i][k];
            }
        }
    }
    scanf("%d",&q);
    while(q--){
        int l,r; scanf("%d%d",&l,&r);
        int ans = 0;
        for(int k = 24; k >= 0; k--){
            if(pos[r][k] >= l && ans < (ans^p[r][k]))
                ans = ans^p[r][k];
        }
        printf("%d\n",ans);
    }
    return 0;
}
```

1.7.5 线段树维护线性基交

引理：若 V_1、V_2 是线性空间，B_1、B_2 分别是他们的一组基，令 $W = B_2 \cap V_1$，若 $B_1 \cup (B_2 \setminus W)$ 线性无关，则 W 是 $V_1 \cap V_2$ 的一组基。

证明:
采用反证法，假设 $\exists\, v \in V_1 \cap V_2$，且 v 不能由 W 线性组合表示。由于 v 一定可以由 W 和 $B_2 \setminus W$ 线性组合表示，即 $v = xor(B_1) = xor(W) \wedge xor(B_2 \setminus W)$，转换可得 $xor(B_2 \setminus W) = xor(B_1) \wedge xor(W)$，因此 $B_2 \setminus W$ 与 B_1 线性相关，与题设条件不符，因此假设不成立，原定理正确。

求解:
因此问题就变成了如何求取 W。令 $tmp = B_1 \cup (B_2 \setminus W)$，$tmp$ 初始等于 B_1，我们在构建 tmp 的过程中求取 W，因此我们不断取出 B_2 的基底加入 tmp。假设当前取出的元素为 x，若 x 不能被 tmp 表示，则直接在 tmp 中加入 x。否则 x 一定能被 tmp 线性组合而成，设 $x = xor(B_1) \wedge xor(tmp \setminus B_1), cur = xor(B_1) = x \wedge xor(tmp \setminus B_1)$，即 cur 既可以由 B_1 表示也可以由 B_2 表示，因此直接将 cur 加入 W 即可。
枚举 x 时，按照从低到高的顺序进行枚举，保证 x 不能仅由 $tmp \setminus B_1$ 表示。

问题描述：给出 n 个线性基，每次询问给出 l、r、x，询问区间 $[l, r]$ 的线性基交能否线性组合出 x。

问题解决：直接线段树维护区间线性基交即可。

```
#include <bits/stdc++.h>
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define per(i,a,b) for(int i = a; i >= b; i--)
typedef long long ll;
const int N = 50000+100;
using namespace std;

int n,m;
struct Linear_Basis{
    ll p[50];
    Linear_Basis() {memset(p,0,sizeof p);}
    ~Linear_Basis() {}
    bool ins(ll x,int jud){
        for(int i = 33; i >= 0; i--){
            if(x&(1ll<<i)){
```

```
                if(!p[i]) {
                    if(jud) p[i] = x;
                    break;
                }
                x ^= p[i];
            }
        }
        return x > 0ll;
    }
}t[2*N],base[N];
inline int get_id(int l,int r) {return (l+r)|(l!=r);}

Linear_Basis merge(Linear_Basis a, Linear_Basis b){
    Linear_Basis A = a, tmp = a, ans; //tmp不断构建A+(B\ans)
    ll cur,d;
    rep(i,0,33) //从低到高，使得不存在一个基底可以仅由(tmp\A)表示
        if(b.p[i]){ //b中有这个基底
            cur = 0, d = b.p[i];
            per(j,i,0)
                if((d>>j)&1){
                    if(tmp.p[j]){
                        d ^= tmp.p[j], cur ^= A.p[j];
                        if(d) continue;
                        ans.p[i] = cur; //cur的第i位不为0
                    }
                    else tmp.p[j] = d, A.p[j] = cur;
                    //如果不能被表示，A的赋值是为了让高位中含有j这位的基底下放到A中j的位置
                    break;
                }
        }
    return ans;
}

void build(int l,int r){
    int now = get_id(l,r);
    if(l == r) {t[now] = base[l]; return;}
    int mid = (l+r)>>1;
    build(l,mid); build(mid+1,r);
    t[now] = merge(t[get_id(l,mid)],t[get_id(mid+1,r)]);
}

bool query(int l,int r,int tl,int tr,ll x){
    int now = get_id(l,r);
    if(tl <= l && r <= tr) return !t[now].ins(x,0);
    int mid = (l+r)>>1;
    if(tr <= mid) return query(l,mid,tl,tr,x);
    else if(tl > mid) return query(mid+1,r,tl,tr,x);
    else return query(l,mid,tl,tr,x) && query(mid+1,r,tl,tr,x);
}

int main()
{
    scanf("%d%d",&n,&m);
    rep(i,1,n){
        int sz; scanf("%d",&sz);
        rep(j,1,sz){
            ll tp; scanf("%lld",&tp);
            base[i].ins(tp,1);
        }
```

```
    }
    build(1,n);
    rep(i,1,m){
        int l,r; ll x; scanf("%d%d%lld",&l,&r,&x);
        if(query(1,n,l,r,x)) printf("YES\n");
        else printf("NO\n");
    }
    return 0;
}
```

1.8 二次剩余

```
#include <bits/stdc++.h>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const db EPS = 1e-9;
using namespace std;
ll pow_mod(ll a, ll k, ll p) {
    ll ret = 1;
    while(k) {
        if(k & 1)   ret = ret * a % p;
        a = a * a % p;
        k >>= 1;
    }
    return ret % p;
}
ll modsqr(ll a, ll n) { //求解 x^2 = a(mod n)
    ll b, k, i, x;
    if(a == 0)  return 0;
    if(n == 2)  return a % n;
    if(pow_mod(a, (n - 1) / 2, n) == 1) {
        if(n % 4 == 3) {
            x = pow_mod(a, (n + 1) / 4, n);
        } else {
            for(b = 1; pow_mod(b, (n - 1) / 2, n) == 1; ++b);
            i = (n - 1) / 2;
            k = 0;
            do {
                i /= 2;
                k /= 2;
                if((pow_mod(a, i, n) * (ll)pow_mod(b, k, n) + 1) % n == 0)  k += (n - 1) / 2;
            } while(i % 2 == 0);
            x = (pow_mod(a, (i + 1) / 2, n) * (ll)pow_mod(b, k / 2, n)) % n;
        }
        if(x * 2 > n)   x = n - x;
        return x;
    }
    return -1;
}
int main() {
```

```
    int T;
    ll p = 1000000007ll;
    ll inv2 = pow_mod(2ll, p - 2, p);
    scanf("%d",&T);
    while (T--) {
        ll b, c;
        scanf("%lld %lld",&b,&c);
        ll a = ((b * b % p) - (4ll * c % p) + p) % p;
        ll x = modsqr(a, p);
        if(x == -1) {
            puts("-1 -1");
        } else {
            ll y = p - x;
            ll X = inv2 * (x + b) % p;
            ll Y = inv2 * (y + b) % p;
            printf("%lld %lld\n", min(X, Y), max(X, Y));
        }
    }
    return 0;
}
```

1.9 常见数学式子

1. $F(n) = \sum\limits_{x=1}^{n} x * 2^x$
错位相减，直接求出答案 $F(n) = (n-1) * 2^{n+1} + 2$。

2. $F(n) = \sum\limits_{x=1}^{n} 2^x * x^2$
先构造 $f(x) = \sum\limits_{i=1}^{n} x^i * i^2$，即直接求出 x 的通式，然后再将 x 替换成 2。
利用高数计算无穷级数的方法，先构造 $\frac{f(x)}{x}$，然后先积分再求导，得出答案 $F(n) = 2^{n+1} * (n^2 - 2 * n + 3) - 6$。

3. 常见组合数公式
$\sum_{i=1}^{n} 2^i i^2 = (n^2 - 2n + 3)2^{n+1} - 6$

$\sum_{i=1}^{n} 2^i i = (n-1)2^{n+1} + 2$

$1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(n+1)(2*n+1)}{6}$

$1^3 + 2^3 + 3^3 + \cdots + n^3 = \frac{n^2(n+1)^2}{4}$

$\sum_{i=0}^{k} C_{n+i}^{n} = C_{n+k+1}^{n+1}$

$\sum_{i=1}^{n} iC_n^i = n * 2^{n-1}$

$\sum_{i=1}^{n} i^2 C_n^i = n * (n+1) * 2^{n-2}$

$\sum_{i=1}^{n} (-1)^{i-1} \frac{C_n^i}{i} = \sum_{i=1}^{n} \frac{1}{i}$

$\sum_{i=0}^{n} (C_n^i)^2 = C_{2n}^{n}$

```
return 0;
```

1.10 数论板子集合

```cpp
#include <bits/stdc++.h>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const int N = 1e5+100;
const int M = 1e5+100;
const db EPS = 1e-9;
using namespace std;

//gcd
ll gcd(ll a,ll b)
{
    return b == 0 ? a:gcd(b,a%b);
}

//快速幂
ll poww(ll a,ll b){
    ll base = a, ans = 1ll;
    while(b){
        if(b&1) ans = (ans*base)%mod;
        base = (base*base)%mod;
        b >>= 1;
    }
    return ans;
}

//线性筛
void primes(int x){
    tot = 0;
    memset(v,0,sizeof v);
    rep(i,2,x){
        if(!v[i]){
            v[i] = i, prime[++tot] = i;
        }
        rep(j,1,tot){
            if(prime[j] * i > x) break;
            v[i*prime[j]] = prime[j];
            if(i % prime[j] == 0) break;
        }
    }
}

//唯一分解
void getFactors(ll x){
    ll tmp = x; tc = 0;
    for(ll i = 1; prime[i]*prime[i] <= tmp; i++){
        if(tmp%prime[i] == 0){
            c[++tc][0] = prime[i]; c[tc][1] = 0;
            while(tmp%prime[i] == 0){
                tmp /= prime[i];
                c[tc][1]++;
            }
        }
```

```
    }
    if(tmp != 1){
        c[++tc][0] = tmp;
        c[tc][1] = 1;
    }
}

//费马小定理 a^(p-2) 为逆元
ll comp(ll base,ll x){ //求组合数, C(base,x)
    ll ans1 = 1, ans2 = 1;
    for(ll i = base; i >= base-x+1ll; i--) ans1 = (ans1*i)%mod;
    for(ll i = 1ll; i <= x; i++) ans2 = (ans2*i)%mod;
    ll tp = (ans1*poww(ans2,mod-2ll))%mod;
    return tp;
}

//递推求组合数
c[1][0] = c[1][1] = 1; //c[i][j] = C(i,j)
for(int i = 2; i <= 3000; i++){
    c[i][0] = 1;
    for(int j = 1; j <= i; j++)
        c[i][j] = (c[i-1][j]+c[i-1][j-1])%mod;
}
```

1.11 斐波那契数列循环节

$(f[1], f[2]) - (f[3], f[4]) - (f[5], f[6]) - (f[7], f[8])$，给定 n 求 m，满足 $f[1+m]\%n = f[1]\%n$
在此题中，由于每次增加两个数，因此最后需要将答案/2。

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

#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
using namespace std;

#define ll unsigned long long

ll gcd(ll a, ll b) {
    return b == 0 ? a : gcd(b, a % b);
}

ll lcm(ll a, ll b) {
    return a / gcd(a, b) * b;
}

struct Matrix {
    ll mat[2][2];

    Matrix() { memset(mat, 0, sizeof mat); }
};

Matrix mul_M(Matrix a, Matrix b, ll mod) {
    Matrix ans;
    for (int i = 0; i < 2; ++i) {
        for (int j = 0; j < 2; ++j) {
            ans.mat[i][j] = 0;
            for (int k = 0; k < 2; ++k) {
                ans.mat[i][j] += a.mat[i][k] * b.mat[k][j] % mod;
```

```
                if (ans.mat[i][j] >= mod)ans.mat[i][j] -= mod;
            }
        }
    }
    return ans;
}

Matrix pow_M(Matrix a, ll n, ll mod) {
    Matrix ans;
    for (int i = 0; i < 2; ++i) ans.mat[i][i] = 1;
    Matrix tmp = a;
    while (n) {
        if (n & 1)ans = mul_M(ans, tmp, mod);
        tmp = mul_M(tmp, tmp, mod);
        n >>= 1;
    }
    return ans;
}

ll pow_m(ll a, ll n, ll mod) {
    ll ans = 1, tmp = a % mod;
    while (n) {
        if (n & 1)ans = ans * tmp % mod;
        tmp = tmp * tmp % mod;
        n >>= 1;
    }
    return ans;
}

const int maxn = 1e6 + 10;

int prime[maxn];

void getprime() {
    memset(prime, 0, sizeof prime);
    for (int i = 2; i <= maxn; ++i) {
        if (!prime[i])prime[++prime[0]] = i;
        for (int j = 1; j <= prime[0] && prime[j] * i <= maxn; ++j) {
            prime[prime[j] * i] = 1;
            if (i % prime[j] == 0)break;
        }
    }
}

ll factor[100][2];
int fatcnt;

int getfactors(ll x) {
    fatcnt = 0;
    ll tmp = x;
    for (int i = 1; prime[i] * prime[i] <= tmp; ++i) {
        factor[fatcnt][1] = 0;
        if (tmp % prime[i] == 0) {
            factor[fatcnt][0] = prime[i];
            while (tmp % prime[i] == 0) {
                factor[fatcnt][1]++;
                tmp /= prime[i];
            }
            fatcnt++;
```

```
        }
    }
    if (tmp != 1) {
        factor[fatcnt][0] = tmp;
        factor[fatcnt++][1] = 1;
    }
    return fatcnt;
}

int legendre(ll a, ll p) {
    if (pow_m(a, (p - 1) >> 1, p) == 1)return 1;
    else return -1;
}

int f0 = 1;
int f1 = 1;

ll getfib(ll n, ll mod) {
    if (mod == 1)return 0;
    Matrix A;
    A.mat[0][0] = 0;
    A.mat[1][0] = 1;
    A.mat[0][1] = 1;
    A.mat[1][1] = 1;
    Matrix B = pow_M(A, n, mod);
    ll ans = f0 * B.mat[0][0] + f1 * B.mat[1][0];
    return ans % mod;
}

ll fac[maxn];

ll G(ll p) {
    ll num;
    if (legendre(5, p) == 1)num = p - 1;
    else num = 2 * (p + 1);
    int cnt = 0;
    for (ll i = 1; i * i <= num; ++i) {
        if (num % i == 0) {
            fac[cnt++] = i;
            if (i * i != num) {
                fac[cnt++] = num / i;
            }
        }
    }
    sort(fac, fac + cnt);
    ll ans;
    for (int i = 0; i < cnt; ++i) {
        if (getfib(fac[i], p) == f0 && getfib(fac[i] + 1, p) == f1) {
            ans = fac[i];
            break;
        }
    }
    return ans;
}

ll find_loop(ll n) {
    getfactors(n);
    ll ans = 1;
    for (int i = 0; i < fatcnt; ++i) {
```

```
        ll record = 1;
        if (factor[i][0] == 2)record = 3;
        else if (factor[i][0] == 3)record = 8;
        else if (factor[i][0] == 5)record = 20;
        else record = G(factor[i][0]);
        for (int j = 1; j < factor[i][1]; ++j) {
            record *= factor[i][0];
        }
        ans = lcm(ans, record);
    }
    return ans;
}

int main() {
    __;
    getprime();
    ll n;
    while (cin >> n) {
        ll ans = find_loop(n);
        cout << ans / 2ll << endl;
    }
    return 0;
}
```

1.12 划分数

一、适用问题
只要涉及一些难以用组合公式直接求得的数的划分问题。比如将 n 个数划分成多个不大于 m 的划分方法，且不存在相同数。或者划分成若干奇数或偶数的问题。

划分要求千变万化，主要识别点在于将划分数类比为盒子与球的问题，类比成多个球放到多个盒子中，且球和盒子都没有区别。

这类问题与一般的划分问题都不同，需要二维 dp 求解，有时还需要处理前缀和进行计算。

二、算法举例
(1) 将 n 个数划分成多个不大于 m 的数，可以存在相同整数

$f[n][m]$ 表示整数 n 的划分中，每个数都不大于 m 的划分方案。转移时可以分成两种情况，第一种情况，划分方案中每个数都小于 m，因此划分方案为 $f[n][m-1]$。

第二种情况，划分方案中至少有一个数为 m，即在 n 中减去 m，得到 $f[n-m][m]$。

$f[n][m] = f[n][m-1] + f[n-m][m]$

(2) 将 n 个数划分成多个不大于 m 的数，不能存在相同整数

这个问题和上一个问题唯一的区别就在于不能存在相同整数，因此只需要修改第二种情况即可实现正确的转移。

$f[n][m] = f[n][m-1] + f[n-m][m-1]$

(3) 将 n 个数划分成 k 个数的划分方案

考虑 k 个数中是否存在 1。第一种情况从 n 个数中取出 k 个，保证剩下 k 组中每组数大于等于 2。第二种情况则保证 k 份中有至少一个 1，即先取出一个 1 分成 $k-1$ 组。

$f[n][k] = f[n-k][k] + f[n-1][k-1]$

(4) 将 n 个数划分成不超过 k 个数的划分方案

此处枚举的是划分方案中是否有 k 组，$f[n][k]$ 分别由划分组数小于 k 和划分组数等于 k 两种情况转移而来。

$f[n][k] = f[n-k][k] + f[n][k-1]$

(5) 将 n 个数划分若干奇数/偶数的划分方案

$g[n][m]$：将 n 划分为 m 个偶数的方案
$f[n][m]$：将 n 划分为 m 个奇数的方案

求 $g[n][m]$ 只需从 n 中取出 m 个，然后将 $f[n-m][m]$ 中的每个数 +1，从奇数变成偶数即可。求 $f[n][m]$ 则需要考虑划分方案中有无 1 的情况，因为直接从 $g[n-m][m]$ 转移过来，会丢失有 1 的这种情况。
$g[n][m] = f[n-m][m]$

$f[n][m] = f[n-1][m-1] + g[n-m][m]$

(6) 将 n 个数划分成若干个数，使这若干个数的乘积最大

1. 对于任意大于等于 4 的正整数 m，可以划分成 $m = m_1 + m_2$，使得 $m_1 * m_2 \geq m$。

证明：令 $m_1 = (int)m/2, m_2 = m - m_1$，$m_1 \geq 2, m_2 \geq m_1$ 且 $m_2 > 2$，则 $m_1 * m_2 \geq 2 * m_2 \geq m$。

2. 因此不断这样拆分下去，最后剩下的数一定是很多个 2 和 3 的组合，即 $ans = 2^r * 3^s$，且 $r \leq 2$。

若 $r > 2$，则至少有 3 个因子为 2，而 $2*2*2 < 3*3$，且 $2+2+2 = 3+3$，因此 $r \leq 2$。

3. 至此为止，不难发现答案只需对小于 4 的数分类讨论，对于大于 4 的数，枚举 2 的最少个数即可。

$n = 3*r$，$ans = 3^r$
$n = 3*r+1$，$ans = 3^{r-1} * 2 * 2$
$n = 3*r+2$，$ans = 3^r * 2$

(7) 将 n 个数划分成若干个不同的数，使这若干个数的乘积最大

1. 拆分成的数的个数越多，乘积越大。因此找到第一个大于等于 n 的 $S_x = \sum_{i=2}^{x} i$。
2. 若 $S_x = n$，则 S_x 即为划分答案。
3. 若 $S_x = n+1$，则将 S_x 中的 2 去掉，令 x 变成 $x+1$，即为最大值。
4. 若 $S_x > n+1, k = S_x - n$，则答案为将 S_x 序列中的 k 去掉，剩下数构成最终答案。

三、例题
题意: 给出一个 H 行 N 列的网格，(r,c) 指第 r 行第 c 列的格子。现在需要在网格中选择 N 个格子涂成黑色。这 N 个格子需要满足一定的条件。
1. $(1,1)$ 需要涂黑。
2. 如果 (r,a) 与 (r,b) 涂黑了，则 (r,k) 为黑。$a < k < b$，即每行涂黑的格子必须是连续的。
3. 如果 (r,c) 涂黑了，则 $(r-1,c)$ 如果存在，也必须涂黑。
4. 如果 (r,c) 涂黑了，并且现在没有任何 $k < c$ 使得 (r,k) 是黑的，则 $(r+1,c)$ 为白。

稍加模拟就会发现满足如上条件的格子组成了一个金字塔形状。如下图所示。
先要求给出一个 H 和 N，求出有多少种不同的涂法满足条件，答案模 10^9+7。$(1 \leq \mathrm{H}, \mathrm{N} \leq 10^5)$

思路: 首先不难发现，若想满足题意进行格子涂色，则每行涂色的格子数必定逐行递减，因此可以发现涂色格子的行数一定在 $\sqrt{\mathrm{N}}$ 级别，即最多不会超过 400。

然后观察这个问题，这是个计数问题，比较常见的考虑就是使用 dp 进行转移计算答案。而使用 dp 的话，我们的选择就无非是按行从上往下 dp 转移，或者按列从左往右 dp 转移。(不过比赛的时候绝大多数想到的都只有按行从上往下)

如果按行从上往下转移，那怎么想都需要一个三维 dp，$dp[i][j][k]$，第 i 行，有 j 个格子，还剩 k 个格子没有填的方案数，然后就可以全队自闭了。思路陷在这里如果出不来就基本是告别 AC 了。

既然行不可行，我们来考虑列转移，从左往右进行列的 dp 转移。我们从最高点将左右两边的金字塔划开，发现左边的金字塔列是不断递增上来的，即前一个与后一个要么高度不变，要么高度 $+1$。而右边的金字塔则只是递增，并没有严格的 $+1$ 关系。

因此我们考虑对于左右两边进行分开 dp。左边状态为 $f[i][j]$，表示 i 个数，最大的数为 j 的方案个数，并且在转移过程严格要求一个数列非递减关系。而定义右边状态为 $s[i][j]$，表示 i 个数，所有的数小于等于 j 的方案个数。然后来分别思考转移方程。

左半部分转移方程，由于必须满足划分的数为非递减关系，因此我们枚举前一个数为 j 或 $j-1$，因此得到如下转移方程。

$f[0][0] = 1$

$f[i][j] = f[i-j][j] + f[i-j][j-1]$

右半部分转移方程，只需满足所有数小于等于 j，因此可以类比之前的经典模型，i 个数进行划分，最大的数不超过 j 的划分方案数。

$s[0][i] = 1$

$s[i][j] = s[i][j-1] + s[i-j][j]$

最后，枚举高度，然后再枚举左半部分格子数统计答案即可。

```
#include <bits/stdc++.h>
#define rep(i,a,b) for(int i = a; i <= b; i++)
const int N = 1e5+100;
const int mod = 1e9+7;
using namespace std;

int n,h,f[N][460],s[N][460];

void init(){
    //f[i][j]: i个数，最大的数为j，方案个数
    f[0][0] = 1;
    rep(i,1,100000)
        rep(j,1,min(i,455)){
            f[i][j] = (f[i-j][j]+f[i-j][j-1])%mod;
        }

    //s[i][j]: i个数，所有的数小于等于j，方案个数
    rep(i,0,455) s[0][i] = 1;
    rep(i,1,100000){
        rep(j,1,455){
            s[i][j] = s[i][j-1];
            if(i >= j) s[i][j] = (s[i][j]+s[i-j][j])%mod;
        }
    }
}

int main()
{
    init();
    scanf("%d%d",&h,&n);
    long long ans = 0;
    //一开始先枚举高度，整个题的思路也是这样，但是根据内存连续性的问题，必T无疑...我也成功找了一个小
```

```
    时bug才找到...
    //今后枚举数组时，务必记得先枚举行，再枚举列，谨记今日教训
    rep(i,1,n)
        rep(j,1,min(h,455))
            ans = (ans+1ll*f[i][j]*s[n-i][j-1])%mod;
    printf("%lld\n",ans);
    return 0;
}
```

1.13 卡特兰数

一、基础公式
(1) 定义式
$f[n] = \sum_{i=0}^{n-1} f[i] * f[n-1-i]$

(2) 组合数公式 (用生成函数推导定义式)
1. $f[n] = C_{2n}^{n} - C_{2n}^{n-1}$
2. $f[n] = \dfrac{C_{2n}^{n}}{n+1}$
3. $f[n] = \dfrac{4n-2}{n+1} * f[n-1]$

(3) 卡特兰数列
1. $f[0] = 1, f[1] = 1, f[2] = 2, f[3] = 5...$
2. 1,1,2,5,14,42,132,429,1430,4862,16796,58786,208012...

二、应用方向
(1) 括号匹配问题
n 个左括号，n 个右括号，对于每一个位置，左括号数大于等于右括号数的方案总数。

等价于 n 个 1，n 个 -1，每个位置前缀和大于等于 0 的方案总数。

两种理解方向：
1. $f[n] = \sum_{i=0}^{n-1} f[i] * f[n-1-i]$。枚举第一次前缀和为 0 的位置，假如第 $2*x$ 个点为第一次前缀和为 0 的点，则固定第一个数为 1，第 x 个数为 -1，则对答案贡献为 $f[x-1] * f[n-x]$。
2. $f[n] = C_{2n}^{n} - C_{2n}^{n-1}$。总方案数为 C_{2n}^{n}，现需求不符合条件的方案数，将问题转化为网格上的折线问题。第 i 次在 (i,j) 处，第 $i+1$ 次在 $(i+1,j+1)$ 或 $(i+1,j-1)$ 处，终点为 $(2n,0)$。不符合条件则说明折线上出现了 $(x,-1)$ 这个点，x 为第一次到达 -1 的点，我们将 x 点之后的折线沿 $y=-1$ 对称过来，则终点为 $(2n,-2)$，则一共有 $n-1$ 个 1，$n+1$ 个 -1，即不合法的方案总数为 C_{2n}^{n-1}，因此 $f[n] = C_{2n}^{n} - C_{2n}^{n-1}$。

(2) 出栈次序问题
一个无穷大的栈，进栈序列为 $1,2,3...n$，求有多少个不同的出栈序列。

(3) 多边划分为三角形问题
将一个凸多边形区域分成三角形区域的方案数。
在圆上选择 $2*n$ 个点，将这些点对连接起来使得所得到的 n 条线段不想交的方案数。

(4) 二叉树计数问题
给定 n 个节点，能构成多少种形状不同的二叉树。
先取一个点作为顶点，然后左边依次可以取 0 $n-1$ 个点，右边则可以取 $n-1$ 0 个点，相乘再累加即可得到答案。

```
return 0;
```

1.14 高斯素数

一、高斯整数

$a = x + y * i \quad (x, y \in Z)$，则 a 为高斯整数。

a 的范为 $N(a) = |a^2| = x^2 + y^2$。

若存在高斯整数 y，使得 $ay = 1$，则 a 为高斯整数中的乘法可逆元，y 为 a 的逆。

高斯整数 a 是可逆元的充要条件是 $N(a) = 1$，高斯整数中只有 4 个可逆元，分别是 $-1\ 1\ i\ -i$ 。

$a\ b$ 为高斯整数，$a = by$，则 a 和 b 等价，即 $a = b\ -b\ ib\ -ib$ 。

二、高斯素数

定义：设 ϕ 为高斯整数中的非零非可逆元，则 ϕ 为高斯素数。即 ϕ 的因子或者为可逆元，或者是与 ϕ 等价的高斯整数。

ϕ 为高斯整数，且 $N(\phi) = p$ 为素数，则 ϕ 必定为高斯素数。

若 ϕ 为高斯素数，则其共轭元也是高斯素数。

三、高斯素数判断

高斯整数 $a + bi$ 是素数，当且仅当：
1. $a\ b$ 中有一个是零，另一个数的绝对值是形如 $4 * n + 3$ 的素数。
2. $a\ b$ 均不为零，而 $a^2 + b^2$ 为素数。

四、费马平方和定理

奇质数能表示为两个平方数之和的充分必要条件是该质数被 4 除余 1，即 $4 * n + 1$。
例题：找出 $[l, r]$ 中的素数 t，满足 $t = a^2 + b^2 \quad (a, b \in N^*)$，输出这种素数总数。

```
return 0;
```

1.15 莫比乌斯反演

题意：
长度为 n 的数组，m 次查询，每次查询给出 (l, r, x)，询问区间 [l, r] 中与 x 互质的数的个数，强制在线。($1\ n, m, a_i, x \leq 10^5$)

思路：
推出莫比乌斯方程，去除 mu[i] = 0 的数进行剪枝。

```
#include <bits/stdc++.h>
const int N = 5e5+10;
using namespace std;

int vis[N], primes, prime[N], mu[N];
int a[N], n, m, maxx;
vector<int> fac[N],pos[N];

void Init()
{
    primes = 0;
    mu[1] = 1;
    for(int i = 2; i < N; i++){
        if(!vis[i]){
            prime[primes++] = i;
```

```
            mu[i] = -1;
        }
        for(int j = 0;j < primes && i*prime[j] < N; j++){
            vis[i*prime[j]] = 1;
            if(i%prime[j]) mu[i*prime[j]] = -mu[i];
            else{
                mu[i*prime[j]] = 0;
                break;
            }
        }
    }

    for(int i = 1; i < N; i++){
        for(int j = i; j < N; j += i){
            if(mu[i] != 0) fac[j].push_back(i);//有贡献的因子
        }
    }
}

int F(int num,int posss)
{
    int l = 0,r = pos[num].size()-1,ret = -1;
    //二分最近位置
    while(l <= r){
        int mid = (l+r) / 2;
        if(pos[num][mid] <= posss){
            ret = mid;
            l = mid+1;
        }
        else{
            r = mid-1;
        }
    }
    return ret;
}

int query(int l, int r, int x){
    int ans = 0;
    for(int j = 0; j < fac[x].size(); j++) {
        int v = fac[x][j];
        int tmp1 = F(v,r);
        int tmp2 = F(v,l-1);
        ans += mu[v]*(tmp1-tmp2);
    }
    return ans;
}

int main()
{
    Init();
    int _; scanf("%d", &_);
    while(_--){
        maxx = 0;
        scanf("%d%d",&n,&m);
        for(int i = 1;i <= n;i ++){
            scanf("%d",&a[i]);
            maxx = max(maxx,a[i]);
            for(int j = 0; j < fac[a[i]].size(); j++){
                int v = fac[a[i]][j];//放倍数位置
```

```
                pos[v].push_back(i);
            }
        }
        int ans = 0, l, r, x;
        for(int i = 1; i <= m; i++){
            scanf("%d%d%d", &l, &r, &x);
            l ^= ans, r ^= ans, x ^= ans;
            ans = query(l, r, x);
            printf("%d\n", ans);
        }
        for(int i = 1; i <= maxx; i++) pos[i].clear();
    }
    return 0;
}
```

1.16 容斥原理

题意：给定一个长度为 n 的数组，$a[1], a[2], ..., a[n]$。维护一个集合，一开始集合为空。一共有 q 次操作，每次操作给定一个下标 pos，如果 $a[pos]$ 已经在集合中，则将 $a[pos]$ 从集合中删除，否则将 $a[pos]$ 加入集合。注意，集合允许有重复的数字。$(1 \le n, q \le 2 * 10^5, 1 \le a[i] \le 5 * 10^5, 1 \le pos \le n)$

问每次操作完之后，集合中互质的数字有多少对。

思路：此题算是一道比较经典的容斥题，也可以用莫比乌斯函数来求解，但二者本质是一样的。

首先比较容易想到的是，往集合中加一个数 $a[pos]$，则答案加上集合中与 $a[pos]$ 互质数的个数；从集合中删除一个数 $a[pos]$，则答案减去集合中与 $a[pos]$ 互质数的个数，这种做法的复杂度是 $O(nq)$。

想要进一步优化复杂度，则需要降低求集合中与 $a[pos]$ 互质数个数的复杂度。我们将求互质转为求不互质，即：

$$\text{与 } a[pos] \text{ 互质数的个数} = \text{集合大小} - \text{与 } a[pos] \text{ 不互质数的个数}$$

又因为如果 x 与 y 不互质，则必存在至少一个质数 c，满足 $x\%c == 0$ 且 $y\%c == 0$，因此我们假设 $a[pos]$ 的质因子有 $p_1 * p_2$，$C[p_1]$ 表示集合中有多少个数有 p_1 这个质因子，则可以根据容斥原理得到：

$$\text{与 } a[pos] \text{ 不互质的个数} = C[p_1] + C[p_2] - C[p_1 * p_2]$$

所有我们可以先求出每个数的质因子集合，然后用 dfs 枚举质因子子集，每当得到一个子集，则将子集中的数字全部乘起来，同时更新 C 数字与容斥答案，具体细节见代码。

```
#include <bits/stdc++.h>
#define rep(i,a,b) for(int i = a; i <= b; i++)
typedef long long ll;
const int N = 5e5+1000;
using namespace std;

int n,q,a[N],vis[N],flag[N],C[N];
ll ans, tmp, cot;

vector<int> prime[N];

void init() {
    scanf("%d%d",&n,&q);
    rep(i,1,n) scanf("%d",&a[i]);
    for(int i = 2; i <= 5e5; i++){
        if(vis[i]) continue;
```

```
        for(int j = i; j <= 5e5; j += i) {
            vis[j] = 1;
            prime[j].push_back(i);
        }
    }
}

void dfs(int now, int num, int cnt, int mul, int op) {
    if(num == prime[now].size()) return;
    if(op == 1) {
        // 加入 now
        tmp += cnt*C[mul*prime[now][num]];
        C[mul*prime[now][num]]++;
    }
    else {
        // 减去 now
        C[mul*prime[now][num]]--;
        tmp += cnt*C[mul*prime[now][num]];
    }
    dfs(now, num + 1, cnt, mul, op);
    dfs(now, num + 1, cnt*(-1), mul*prime[now][num], op);
}

int main()
{
    init();
    while(q--) {
        int pos; scanf("%d",&pos);
        flag[pos] ^= 1;
        tmp = 0;
        if(flag[pos]) {
            dfs(a[pos], 0, 1, 1, 1);
            ans += cot - tmp;
            cot++;
        }
        else {
            dfs(a[pos], 0, 1, 1, -1), cot--;
            ans -= cot - tmp;
        }
        printf("%lld\n", ans);
    }
    return 0;
}
```

2 博弈

2.1 SG 函数

题意：有 n 堆魔法珠，第 i 堆有 a_i 颗。选择 n 堆魔法珠中数量大于 1 的任意一堆。记该堆魔法珠数量为 p，p 有 $b_1\ b_2\ b_3\ ...\ b_m$ 这 m 个小于 p 的约数。然后将这一堆魔法珠变成 m 堆，每堆各有 $b_1\ b_2\ b_3\ ...\ b_m$ 颗魔法珠。最后选择这 m 堆的任意一堆，令其消失，不可操作者输，问谁能获胜。$(1 \le n \le 100, 1 \le a_i \le 1000)$

思路：不难发现整个游戏就是由多个魔法珠堆组成的，因此 $SG[x]$ 表示某一堆魔法珠，个数为 x 时的 SG 值。求 $SG[x]$ 时，若 x 有 $b_1\ b_2\ b_3\ ...\ b_m$ 这 m 个小于 x 的约数，则可以达到的后继状态一共有 m 个，即去掉任意一堆达到的状态。而每一个状态中仍有 $m-1$ 个堆，即 $m-1$ 个子游戏，因此每一个状态的 SG 值为这 $m-1$ 个子游戏 SG 值的异或和。具体过程见代码。

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const int N = 1000+10;
const int M = 1e5+100;
const db EPS = 1e-9;
using namespace std;

int sg[N],n;

int solve(int x)
{
    if(sg[x] != -1) return sg[x];
    int vis[2010]; memset(vis,0,sizeof vis);
    int ans[2010]; //记录每一个约数
    int tp = 0, ct = 0;
    rep(i,1,x-1){
        if(x%i == 0){
            ans[++ct] = solve(i);
            tp ^= ans[ct];
        }
    }
    rep(i,1,ct) vis[tp^ans[i]] = 1; //枚举每一个子状态
    rep(i,0,2000)
        if(vis[i] == 0) return sg[x] = i;
}

int main()
{
    memset(sg,-1,sizeof sg);
    sg[1] = 0;
    while(~scanf("%d",&n)){
        int ans = 0;
        rep(i,1,n){
            int xx; scanf("%d",&xx);
```

```
            ans ^= solve(xx);
        }
        if(ans == 0) printf("rainbow\n");
        else printf("freda\n");
    }
    return 0;
}
```

3 动态规划

3.1 01 背包

题意：一个比赛有 n 个裁判，每个裁判有一个权值 ai，并且给出通过或不通过。有一个判断值 p，若所有给出通过的裁判的权值和大于等于 p，那么整体就算通过。给出第二组裁判，权值为 bi，判断值为 q，判断所有 2^n 种判断下这两组给出的结果是否完全相同，如果不相同，给出一种不相同的情况。

思路：比赛的时候完全没思路，一个劲的贪心... 赛后才知道这题其实是个 dp，dp[i][j] 表示前 i 个人，序列 a 中选的人的值到达 j 的所有情况中，序列 b 中对应人的权值之和的最大值。因此当 dp[i][j] >= q，而 j < p 时，即是错误的情况。然后再判断一遍 b 序列即可。此处还有一个地方，记错误方案时如果是用 pre 数组不断回溯的话，会 T。正确做法是对于每个 dp[j] 记一个 bitset 的状态，检查出错误后，直接输出即可。

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <bitset>
#include <algorithm>
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl
    ;
typedef long long ll;
typedef double db;
const db EPS = 1e-9;
using namespace std;
const int N = 200;
const int M = 1e6+1000;

int n,a[N],b[N],dp[M],am,bm,tp[M],ans[N];
//dp[i]表示a组中，价值为i的所有组合中，b组最大价值的一组
bitset<110> base1[M],base2[M];

void solve()
{
    rep(i,1,n) ans[i] = 0;
    rep(i,1,n){
        for(int j = am-1; j >= a[i]; j--){
            if(j == a[i]){
                if(dp[j] < b[i]) dp[j] = b[i], base1[j].reset(),base1[j].set(i);
            }
            if(dp[j] < dp[j-a[i]]+b[i] && dp[j-a[i]]!=0){
                dp[j] = dp[j-a[i]]+b[i];
                base1[j] = base1[j-a[i]];
                base1[j].set(i);
            }
            if(dp[j] >= bm){
                printf("NO\n");
                rep(i,1,n) printf("%d",(int)base1[j][i]);
                printf("\n");
                return;
            }
        }
    }
    rep(i,1,n){
        for(int j = bm-1; j >= 1; j--){
            if(j == b[i]){
                if(tp[j] < a[i]) tp[j] = a[i], base2[j].reset(),base2[j].set(i);
```

```
            }
            if(j < b[i]) continue;
            if(tp[j] < tp[j-b[i]]+a[i] && tp[j-b[i]]!=0){
                tp[j] = tp[j-b[i]]+a[i];
                base2[j] = base2[j-b[i]];
                base2[j].set(i);
            }
            if(tp[j] >= am){
                printf("NO\n");
                rep(i,1,n) printf("%d",(int)base2[j][i]);
                return;
            }
        }
    }
    printf("YES\n");
}

int main()
{
    scanf("%d",&n);
    scanf("%d",&am);
    rep(i,1,n) scanf("%d",&a[i]);
    scanf("%d",&bm);
    rep(i,1,n) scanf("%d",&b[i]);
    solve();
    return 0;
}
```

3.2 完全背包

题意:
音符格式转换成长度最小且字典序最小的格式。一共有 7 种音节，分别是 $R1$、$R2$、$R4$、$R8$、$R16$、$R32$、$R64$，分别表示 1、$\frac{1}{2}$、$\frac{1}{4}$、$\frac{1}{8}$、$\frac{1}{16}$、$\frac{1}{32}$、$\frac{1}{64}$ 拍。对于 $R1R2$ 即为 $\frac{3}{2}$ 拍，也可以表示成 $R1.$。即可将相邻的一个音节化为 '.'，因此 $R2...$ 表示 $R2R4R16R32$。

此题给出一个音符表示格式，要求转化为长度最短且字典序最小的方式。

例如 $R1R4R16$ 转化为 $R16R1R4$ 。

思路:
比赛的时候遇到这题，错误地当成了贪心问题进行考虑，导致始终无法 AC。

现在我们来观察这个问题，这个表达式所有音节的拍数相加之和是不变的，而每一个音节单元的拍数和长度也都是确定的，因此等价于一个完全背包问题。

因此如果此题没有要求给出一个字典序最小的方案，只要求最短长度的话。$dp[i]$ 表示音符表达式拍数为 i 所需的最短长度，然后直接跑一个完全背包即可。

但是这题还要求求出长度最小时字典序最小，因此我们需要考虑状态方程。我一开始的错误做法是两维 for 循环，第一维枚举用到了第 i 个音节单元，第二维枚举当前拍数 j，然后在转移时若是小于，直接转移，若是等于，则比较当前音节单元字典序与 dp[j] 的最后一个音节单元字典序。这样会 wa，因为无法保证字典序最小。

但是如果换一种枚举方式就可以通过此题，第一维 for 循环枚举音节拍数 i，第二维枚举使用了第 j 个音节单元，然后 == 部分的转移仍比较当前 j 与 $dp[i]$ 的上一个音节单元字典序即可通过此题。原因在于求取 $dp[i]$ 的时候，已经使用了所有的音节单元，而前一种枚举方式并没有用上所有的音节单元所以导致出错。

当然此题也可以字典序倒序枚举前 i 个音节单元, 也可以通过此题。

处理完这部分之后，就是完全背包的路径输出了，由于完全背包 $dp[i][j]$ 是由 $dp[i][j-w[i]]$ 转移而来，即同维转移而来，因此不需要不需要 01 背包的二维记录路径，直接使用一维进行路径记录，即上一次由哪个音节单元转移而来即可。

总结:
此题考查了完全背包的两个内容,一是完全背包求取过程中加上了对于字典序最小的求解,二是完全背包的一维路径输出。

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <string>
#include <bitset>
#include <algorithm>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const int N = 3200000+10;
const int M = 1e5+100;
const db EPS = 1e-9;
using namespace std;

char s[N];
int len,n,tot,dp[N],pre[N],ttp[N];
// int str[N][33];
// string str[N];
//v: 拍数, w: 长度
//len: 总拍数
//用最少的长度凑满总拍数
//dp[i]: 表示凑满拍数为i所需的最少长度

struct Node{
    int num,sum,ctt,v,w; //num: 编号, sum: .个数, ctt: 选了多少个, v: 拍数, w: 字符长度
    string ss;
    bool operator < (Node xx) const {
        if(num == xx.num) return sum > xx.sum;
        else return (ss < xx.ss);
    }
}ans[30];

void solve()
{
    memset(dp,0x3f,sizeof dp);
    dp[0] = 0;
    rep(j,1,len) //音符长度
        rep(i,1,tot){ //枚举所有音符单元
            if(j >= ans[i].v){
                if(dp[j] > dp[j-ans[i].v]+ans[i].w){
                    dp[j] = dp[j-ans[i].v]+ans[i].w;
                    pre[j] = i;
                }
                else if(dp[j] == dp[j-ans[i].v]+ans[i].w){
                    if(pre[j] > i){ //之前的字典序更大
```

```
                        dp[j] = dp[j-ans[i].v]+ans[i].w;
                        pre[j] = i;
                    }
                }
            }
        }
    int tp = len;
    while(tp > 0){ //完全背包一维记录路径, 直接回溯
        ans[pre[tp]].ctt++;
        tp = tp-ans[pre[tp]].v;
    }
    rep(i,1,tot)
        rep(j,1,ans[i].ctt)
            cout << ans[i].ss;
    cout << endl;
}

int main()
{
    // freopen("e.in","r",stdin);
    // freopen("e.out","w",stdout);
    scanf("%s",s+1); n = strlen(s+1);
    int num;
    //获取拍数总长度
    rep(i,1,n){
        num = 0;
        if(s[i] == 'R'){
            i++;
            while(s[i] != 'R' && i <= n && s[i] != '.')
            {
                num = num*10+s[i]-'0';
                i++;
            }
        }
        int tt = 64/num;
        // LOG2("i",i,"tt",tt);
        len += tt;
        if(s[i] == '.'){
            while(s[i] != 'R' && i <= n){
                tt /= 2;
                len += tt;
                i++;
            }
        }
        i--;
    }
    //构造所有音符单元
    rep(i,1,7){
        rep(j,0,i-1){
            int tp = 1<<(i-1);
            ++tot;
            ans[tot].v += tp;
            rep(k,1,j){
                tp /= 2, ans[tot].v += tp;
            }
            ans[tot].w = j+1;
            if(i <= 3) ans[tot].w += 2;
            else ans[tot].w += 1;
            ans[tot].sum = j;
```

```
            ans[tot].num = 1<<(7-i);
        }
    }
    rep(i,1,tot){
        ans[i].ss = "\0";
        string tpp;
        tpp += 'R';
        tpp += to_string(ans[i].num);
        rep(j,1,ans[i].sum) tpp += ".";
        ans[i].ss = tpp;
    }
    sort(ans+1,ans+1+tot);
    //按字典序构造音符单元 —— 非关键操作，也可直接打表输入
    Node tp = ans[7];
    rep(i,8,10) ans[i-1] = ans[i];
    ans[10] = tp;
    // rep(i,1,tot)
        // LOG3("i",i,"ss",ans[i].ss,"v",ans[i].v);
    solve();
    return 0;
}
```

3.3 多重背包

```
#include <cstdio>
#include <iostream>
#include <algorithm>
#include <cstring>
#define rep(i,a,b) for(int i = a;i <= b;i++)
using namespace std;
const int N = 500;

int v[N],w[N],num[N];  //每袋的价格、重量和袋数
int dp[N];  //dp[i]:总经费为i时所能买的最多重量的大米
int n,total;  //大米种类、总经费

void ZeroOnePack(int cost,int weight)
{
    for(int i = total; i >= cost;i--)
        dp[i] = max(dp[i],dp[i-cost]+weight);
}

//完全背包，代价为cost，获得的价值为weight
void CompletePack(int cost,int weight)
{
    for(int i = cost;i <= total;i++) //total为上界
        dp[i] = max(dp[i],dp[i-cost]+weight);
}

//多重背包，代价为cost，获得的价值为weight
void MultiplePack(int cost,int weight,int amount)
{
    if(cost*amount >= total) CompletePack(cost,weight);  //相当于取任意袋数，变为完全背包
    else{
        int k = 1;
        while(k < amount){
            ZeroOnePack(k*cost,k*weight);  //二进制拆分
            amount -= k;
```

```
            k<<=1;
        }
        ZeroOnePack(amount*cost,amount*weight);  //把剩余个数用01背包求
        //这个很重要，不要忘记了
    }
}

int main()
{
    int T;
    scanf("%d",&T);
    while(T--)
    {
        memset(dp,0,sizeof dp);  //memset别忘了，很重要!
        scanf("%d%d",&total,&n);
        rep(i,1,n)
            scanf("%d%d%d",&v[i],&w[i],&num[i]);
        rep(i,1,n)
            MultiplePack(v[i],w[i],num[i]);
        printf("%d\n",dp[total]);
    }
    return 0;
}
```

3.4 状压 dp

3.4.1 子集枚举

题意：
一共 N 个人，给出任意两个人之间的胜负关系，你的编号是 M。现在需要安排一棵竞赛树使得 M 能够胜出，问使竞赛树高度最小且 M 获胜的安排方案一共有多少个。$(1 \le N \le 16)$

思路：
根据题意以及数据范围，可以很明显的发现这是一个状压 dp，因此我们来考虑 dp 的状态。

既然是状态，那肯定要记录当前的状态，即选了哪些人，然后还要记录当前的胜利者，以及当前树的高度，因此 $dp[i][j][k]$ 表示 i 状态下，胜出者为 j，树高度为 k 的安排方案数。

然后采用记忆化搜索，求取 $dp[i][j][k]$ 时将 i 分为两个子状态 x、y，然后递归求取 $dp[x][j][k-1]$，枚举 m 为 y 状态下的胜利者并且会输给 j，求取 $dp[y][m][k]$。具体细节见代码。

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <cmath>
#include <algorithm>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl
    ;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2
    << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const int N = 1e5+100;
const int M = 1e5+100;
```

```
const db EPS = 1e-9;
using namespace std;

template<class T> int getbit(T s, int i) { return (s >> i) & 1; }
template<class T> T onbit(T s, int i) { return s | (T(1) << i); }
template<class T> T offbit(T s, int i) { return s & (~(T(1) << i)); }
template<class T> int cntbit(T s) { return __builtin_popcount(s);}

int c[20][20],n,win,h[20];
ll dp[1<<16][17][7];

ll solve(int stat,int m,int height){
    if(dp[stat][m][height] != -1) return dp[stat][m][height];
    if(cntbit(stat) == 1){
        if(getbit(stat,m-1)) return dp[stat][m][height] = 1ll;
        else return dp[stat][m][height] = 0;
    }
    for(int i = (stat-1)&stat; i >= 1; i = (i-1)&(stat)){ //枚举子集
        int x = i, y = stat-i;
        if(!getbit(x,m-1) || h[cntbit(x)] > height-1 || h[cntbit(y)] > height-1)
    continue;
        // if(height-1 > cntbit(x) || height-1 > cntbit(y)) continue;
        ll ans1 = solve(x,m,height-1);
        ll ans2 = 0;
        rep(j,1,n)
            if(getbit(y,j-1) && c[m][j] == 1) ans2 += solve(y,j,height-1);
        if(dp[stat][m][height] == -1) dp[stat][m][height] = ans1*ans2;
        else dp[stat][m][height] += ans1*ans2;
    }
    return max(0ll,dp[stat][m][height]);
}

int main()
{
    freopen("f.in","r",stdin);
    freopen("f.out","w",stdout);
    scanf("%d%d",&n,&win);
    rep(i,1,n)
        rep(j,1,n) scanf("%d",&c[i][j]);
    rep(i,1,16) h[i] = (log(i-0.5)/log(2))+2;
    memset(dp,-1,sizeof dp);
    printf("%lld\n",max(0ll,solve((1<<n)-1,win,h[n])));
    return 0;
}
```

3.4.2 计算一对元素贡献

题意:
有一个 n 个数字的序列，数字范围在 1 ~ 20 之间，现在要对这些数字进行重新排列，使得相同数字出现在一个连续区间，如 113322、22222233444 都是符合题意的序列。重新排列这个序列的操作只有交换两个相邻的数字，现问最少需要交换多少次可以使这个序列变成符合题意的序列。$(2 \le n \le 4 * 10^5)$

思路:
由于所有数字都在 1 ~ 20 之间，因此应该对于有多少个数字进行考虑，而不是对 n 进行考虑。

如果直接考虑数字的移动会将问题复杂化，因为考虑的内容会非常多。因此我们直接构建一个最终序列，然后求原序列转化到这个最终序列的贡献。

先说如何求，再说如何构建。因为如果不会求的话，也不知道构建的入手点在哪。对于求贡献，此处主要对于一对元素进行考虑。

对一对元素进行考虑，即仅考虑 1、2 时，序列为 1 1 2 1 1 2 1 2 2，我们令 $val[i][j]$ 表示元素 i 在元素 j 之前对答案的影响，因此只需要考虑对每个 1 来说前面有多少个 2，即 $val[1][2] = 1+1+2 = 4$，$val[2][1] = 2+4+5+5 = 16$。

知道如何求贡献之后，我们来考虑如何解决这个问题。令 $dp[i][S]$ 表示当前放置第 i 个元素，前面放的元素状压的状态为 S，因此我们枚举第 i 个元素为 j，$dp[i][S] = min(dp[i-1][S \wedge (1 \ll j)])$。计算贡献时可以使用一对元素的考虑方法，需要进行预处理。
总结:
这其实是一道计算贡献的状压 dp 问题，主要难点应该在于从考虑过程转变为考虑结果，然后再思考如何根据初始序列和结果序列直接得到答案。再之后就是直接通过状压再加上一定的预处理通过此题。

考虑结果序列的贡献中，一对元素的考虑方式比较巧妙，类似于逆序对的感觉，主要抓住的特点在于一个元素在序列中的位置本质上是由它前面有多少个元素决定的，因此想到了一对元素的思考方式来解决问题。

```cpp
#include <bits/stdc++.h>
#define mem(a,b) memset(a,b,sizeof a);
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define per(i,a,b) for(int i = a; i >= b; i--)
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
typedef long long ll;
typedef double db;
const int N = 4e5+100;
const ll inf = 1e15;
const db EPS = 1e-9;
using namespace std;

void dbg() {cout << "\n";}
template<typename T, typename... A> void dbg(T a, A... x) {cout << a << ' '; dbg(x...);}
#define logs(x...) {cout << #x << " -> "; dbg(x);}

int n,a[N],b[N],tot,num[25],cnt[1<<20];
ll dp[22][1<<20],val[25][25];

int main()
{
    rep(S,0,(1<<20)-1){
        int hp = S, c1 = 0;
        while(hp){
            if(hp&1) c1++;
            hp /= 2;
        }
        cnt[S] = c1;
    }
    scanf("%d",&n);
    rep(i,1,n){
        scanf("%d",&a[i]);
        b[++tot] = a[i];
    }
    sort(b+1,b+1+tot);
    tot = unique(b+1,b+1+tot)-b-1;
    rep(i,1,n){
        num[a[i]]++;
        int pos = lower_bound(b+1,b+1+tot,a[i])-b;
        rep(j,1,tot)
            if(b[j] != a[i]) val[pos][j] += num[b[j]];
    }
```

```
    rep(i,0,tot)
        rep(S,0,(1<<tot)-1) dp[i][S] = inf;
    dp[0][0] = 0;
    rep(i,1,tot){
        rep(j,1,tot){ //枚举放的东西
            rep(S,0,(1<<tot)-1){ //前面的状态
                if(S&(1<<(j-1)) && cnt[S] == i){
                    ll base = dp[i-1][S^(1<<(j-1))];
                    rep(k,1,tot)
                        if(S&(1<<(k-1))){
                            base += val[k][j];
                        }
                    dp[i][S] = min(dp[i][S],base);
                }
            }
        }
    }
    printf("%lld\n",dp[tot][(1<<tot)-1]);
    return 0;
}
```

3.5 区间 dp

3.5.1 区间 dp 例题 1

题意:

给定一个序列。将序列中的一个数字消去的代价是与这个数字相邻的两个数字的 gcd，问将所有数字消去的最小代价。注意这个序列是环形的。

思路:

首先我们比较容易发现这是一个区间 DP 问题，于是问题就变成了如何列区间 DP 状态。一开始考虑的是 dp[i][j] 表示区间 [i,j] 全部消去的最小代价，然后在区间 [i,j] 中枚举第一个消去的 k 进行更新。然后会发现一个问题，如何先消的是 j ，那么 j 是右端点，因此 j 两端的元素是不确定的，因此这个转移方程不对。

因此我们来重新考虑这道题。由于左右端点不确定，因此我们重新定义 dp 状态，dp[i][j] 表示区间 [i, j] 中所有数全部消除，最后剩下 i 和 j 的最小代价。则在区间中枚举 k ，dp[i][j] = dp[i][k]+dp[k][j]+gcd(i,j) 。由于是个环形序列，因此需要将长度扩展两倍进行 dp。最后的答案就是枚举最后剩下的两个点，然后找最小值即可。

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl
    ;
typedef long long ll;
typedef double db;
const db EPS = 1e-9;
using namespace std;
const int N = 300;

int n,a[N],dp[N][N];

int gcd(int a,int b)
{
    return b == 0 ? a:gcd(b,a%b);
}
```

```

int main()
{
    while(~scanf("%d",&n))
    {
        if(n == 0) break;
        rep(i,1,n) scanf("%d",&a[i]);
        rep(i,n+1,2*n) a[i] = a[i-n];
        rep(len,1,n)
            rep(j,1,2*n){
                int x = j, y = x+len-1;
                if(y > 2*n) continue;
                if(y == x+1 || y == x) dp[x][y] = 0;
                else{
                    dp[x][y] = 10000;
                    rep(k,x+1,y-1) dp[x][y] = min(dp[x][y],dp[x][k]+dp[k][y]+gcd(a[x],a[y]));
                }
                // printf("dp[%d][%d]:%d\n",x,y,dp[x][y]);
            }
        int ans = 10000;
        rep(i,1,n)
            rep(j,i+1,i+n-1){
                if(ans > dp[i][j]+dp[j][i+n]+gcd(a[i],a[j])){
                    ans = dp[i][j]+dp[j][i+n]+gcd(a[i],a[j]);
                    // LOG1("ans",ans);
                    // LOG2("i",i,"j",j);
                }
            }
        printf("%d\n",ans);
    }
    return 0;
}
```

3.5.2 区间 dp 例题 2

题意：
一个序列，选手 A、B 轮流从序列中从左端或者右端选一段区间，然后区间和加到自己的权值中。两个选手都会按照最优的方式进行选取，问先手 A 最多可以比 B 多拿多少。$(1 \le n \le 100)$

思路：
很明显这是一道 DP 问题，又因为只能从左端点或右端点拿，因此不难想到用区间 DP 的方法来解决此题。

既然是区间 DP，那么最常见的状态就是 $DP[i][j]$ 表示对于区间 $[i,j]$ ，先手最多领先后手多少。又因为区间和是一定的，因此已知选手 A 获得的价值就可以知道选手 B 获得的价值，因此修改状态为 $DP[i][j]$ 表示区间 $[i,j]$，先手最多可以获得多少价值。

因此 $DP[i][j] = max(sum[i][j] - DP[x][j], sum[i][j] - DP[i][y])$ $i < x, y < j$，由于 n 比较小，直接枚举区间长度，从小区间到大区间进行转移即可。

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
```

```
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl
    ;
typedef long long ll;
typedef double db;
const db EPS = 1e-9;
using namespace std;
const int N = 200;

int n,a[N],sum[N],dp[N][N];
//dp[i][j]: 表示区间[i,j]先手能够获得的最大价值

int main()
{
    while(~scanf("%d",&n))
    {
        if(!n) break;
        memset(dp,0,sizeof dp);
        rep(i,1,n) scanf("%d",&a[i]);
        rep(i,1,n) sum[i] = sum[i-1]+a[i];
        rep(len,1,n)
            rep(i,1,n-len+1){
                int j = i+len-1;
                dp[i][j] = sum[j]-sum[i-1];
                rep(k,1,len-1){
                    dp[i][j] = max(sum[j]-sum[i-1]-dp[i+k][j],dp[i][j]);
                    dp[i][j] = max(sum[j]-sum[i-1]-dp[i][j-k],dp[i][j]);
                }
            }
        int ans = dp[1][n]-(sum[n]-sum[0]-dp[1][n]);
        printf("%d\n",ans);
    }
    return 0;
}
```

3.6 数位 dp

3.6.1 数位 dp 模板

通常来说，数位 dp 问题都是通过 dfs 解决的，因为 dfs 的做法更容易理解，也能一定地简化代码。

对于 dfs 求解的数位 dp 问题，其中设置的状态为 $f[pos][...]$，pos 表示最后 pos 位没有填，... 表示的是从 pos 位之前继承来的信息，然后 $f[pos][...]$ 的数值表示仅考虑这 pos 位所对答案产生的贡献。

这 ... 所表示的状态通常需要根据题意来进行定义，比较个性化，也是数位 dp 的核心难点。但数位 dp 问题还是非常套路化的，你只需要根据题意想明白想要计算后 pos 位的信息，到底需要从前几位继承哪些信息，想明白这个之后就可以直接套上 dfs 的模板进行求解了。

下面的习题给出的都是非套路化问题，具有一定的难度，初学者建议先写一些模板题再来进行挑战。

最后给出 dfs 问题的大致模板。(某一模板题的 AC 代码)

```
#include <bits/stdc++.h>
#define rep(i,a,b) for(int i = a; i <= b; i++)
typedef long long ll;
using namespace std;

ll f[21][21][2010],a[30]; //左-右

```

```
ll dfs(int pos,int balan,int k,bool flag){
    //位置 平衡点 左边继承来的数值 有无继承
    if(pos == 0){
        if(k == 0) return 1;
        else return 0;
    }
    if(!flag && f[pos][balan][k+1000] != -1) return f[pos][balan][k+1000];
    ll ans = 0;
    int end = flag?a[pos]:9;
    rep(i,0,end){
        ll tp = dfs(pos-1,balan,k+(pos-balan)*i,flag && i == end);
        ans += tp;
    }
    if(!flag) f[pos][balan][k+1000] = ans;
    return ans;
}

ll solve(ll n){
    //求a数组
    if(n == -1) return 0;
    int pos = 0;
    memset(a,0,sizeof a);
    while(n){
        a[++pos] = n%10;
        n /= 10;
    }
    ll ans = 0;
    rep(i,1,pos) ans += dfs(pos,i,0,1); //对每一个平衡点分开求
    ans -= pos-1;
    return ans;
}

int main()
{
    //初始化
    rep(i,0,20)
        rep(j,0,20)
            rep(k,0,2000) f[i][j][k] = -1;
    //读入
    int _; scanf("%d",&_);
    while(_--){
        ll L,R; scanf("%lld%lld",&L,&R); L--;
        printf("%lld\n",solve(R)-solve(L));
    }
    return 0;
}
```

3.6.2 巧妙数位 dp 例题

题意：

给定 k 与 b，求出所有 k 在 0 (2^b-1) 范围内的倍数，将这些倍数二进制中的 1 求 sum 和，模 1e9+9 输出。

思路：

首先，这是一个在数位上的 dp，重点就在于如何描述每个数的状态。

发现数的范围很大，想要直接描述是不可能的。但是 k 的范围很小，只有 1000，因此考虑存储这个数

然后就可以列出 dp 方程，dp[i][j] 表示前 i 个二进制位，mod k = j 的个数，再用 ans[i][j] 表示前 i 个二进制位，mod k = j 的每一种情况二进制拆分后 1 的总和。

因为在 mod 意义下的加减都是可以的。因此对于第 i 个位置，我们只需考虑此处为 0 还是 1，只有两个状态，然后就可以列出转移方程。

dp[i][j] = dp[i-1][j]+dp[i-1][(j-poww[i]+k)
ans[i][j] = ans[i-1][j]+ans[i-1][(j-poww[i]+k)

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#define rep(i,a,b) for(int i = a; i <= b; i++)
using namespace std;
const int mod = 1e9+9;

int k,b,dp[150][1100],poww[200],ans[150][1100];

int main()
{
    scanf("%d%d",&k,&b);
    poww[1] = 1%k;
    rep(i,2,130) poww[i] = (poww[i-1]*2)%k;
    dp[1][0] = 1, dp[1][1] = 1;
    ans[1][0] = 0, ans[1][1] = 1;
    if(k == 1) dp[1][0] = 2, dp[1][1] = 0, ans[1][0] = 1, ans[1][1] = 0;
    rep(i,2,b)
        rep(j,0,k-1){
            dp[i][j] = (dp[i-1][j]+dp[i-1][(j-poww[i]+k)%k])%mod;
            ans[i][j] = ((ans[i-1][j]+ans[i-1][(j-poww[i]+k)%k])%mod+dp[i-1][(j-poww[i]+k)%k])%mod;
        }
    printf("%d\n",ans[b][0]);
    return 0;
}
```

3.6.3 双数 dp 问题

题意:
给出区间 $[l,r]$，查询多少对 (a,b) 满足如下条件。$(0 \le l \le r \le 10^9)$
1. $a+b=a\ xor\ b$
2. $l \le a \le r$
3. $l \le b \le r$

思路:
$CF\ div2$ 的最后一题，比赛的时候有 $1h$ 来考虑这个问题。当时主要在思考这个题想要考察的是什么内容，思考过数位 dp，但是没有做过 $pair$ 类型的数位 dp，于是就没有从这个角度继续往下深入思考。因此剩下的大部分时间都在思考是不是一道结合某些数据结构的思维题，说实话，感觉数据结构开始限制我的思维了，什么题都老从数据结构考虑，这样非常容易被治，必须要改!

继续回到该题，此题其实想要询问的就是 $a\&b=0$ 的 $pair$ 对数。因此我们处理出一个 $solve(x,y)$ 函数，表示 $a \in [0,x], b \in [0,y]$，符合条件的 $pair$ 对数，所以最终答案就是 $solve(r,r)-solve(l-1,r)-solve(r,l-1)+solve(l-1,l-1)$。

接下来就是如何计算 $solve(x,y)$ 函数，其实维护 x 和 y 各自的数组，两个 $flag$，然后套上最基本的 dfs 板子，稍微改一下就可以过了。

总结:
此题其实应该是两个数同时数位 dp 的裸题，套上了一个最基本的容斥。而具体的函数实现过程还是比较套路的，并不难思考。

做不出来的原因也主要是没有从数位 dp 这个角度继续往下深挖，是自己思考的片面性错失了 AC。

```
#include <bits/stdc++.h>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
typedef long long ll;
typedef double db;
const int N = 1e5+100;
const int M = 1e5+100;
const db EPS = 1e-9;
using namespace std;

void dbg() {cout << "\n";}
template<typename T, typename... A> void dbg(T a, A... x) {cout << a << ' '; dbg(x...);}
#define logs(x...) {cout << #x << " -> "; dbg(x);}

ll f[40],a[40],b[40]; //左-右

ll dfs(int pos,bool flag1,bool flag2){
    if(pos == 0) return 1;
    if(!flag1 && !flag2 && f[pos] != -1) return f[pos];
    ll ans = 0;
    int end1 = flag1?a[pos]:1;
    int end2 = flag2?b[pos]:1;
    if(!flag1 && !flag2){
        ans += 3ll*dfs(pos-1,0,0);
    }
    else if(!flag1){
        rep(i,0,end2){
            if(i == 0) ans += 2ll*dfs(pos-1,0,flag2 && i == end2);
            else ans += dfs(pos-1,0,flag2 && i == end2);
        }
    }
    else if(!flag2){
        rep(i,0,end1){
            if(i == 0) ans += 2ll*dfs(pos-1,flag1 && i == end1,0);
            else ans += dfs(pos-1,flag1 && i == end1,0);
        }
    }
    else{
        rep(i,0,end1){
            rep(j,0,end2){
                if(i != 1 || j != 1) ans += dfs(pos-1,flag1 && i == end1,flag2 && j == end2);
            }
        }
    }
    if(!flag1 && !flag2) f[pos] = ans;
    return ans;
}

ll solve(ll x,ll y){
    if(x == -1 || y == -1) return 0;
    int p1 = 0, p2 = 0;
    memset(a,0,sizeof a);
    memset(b,0,sizeof b);
    while(x){
```

```
        a[++p1] = x%2;
        x /= 2;
    }
    while(y){
        b[++p2] = y%2;
        y /= 2;
    }
    return dfs(max(p1,p2),1,1);
}

int main()
{
    int _; scanf("%d",&_);
    memset(f,-1,sizeof f);
    while(_--){
        ll L,R; scanf("%lld%lld",&L,&R);
        ll ans = solve(R,R)-solve(L-1,R)-solve(R,L-1)+solve(L-1,L-1);
        printf("%lld\n",ans);
    }
    return 0;
}
```

3.6.4 三数 dp 问题

题意:
给出四个数，$L\ R\ A\ B$，表示 $x \in [L,R], y \in [0,A], z \in [0,B]$，求 $(x \wedge y) + (y \& z) + (z \wedge x)$ 的最大值。$(0 \le L \le R \le 10^{18}, 0 \le A, B \le 10^{18})$

思路:
这个问题考察的是涉及三个数字的数位 dp，首先我们来思考一下如何表示状态。

此题是求取最大值，因此很明显需要从数位的角度入手进行考虑，所以状态的设置一定会包含各个数位的状态，我们另 $f[pos][p1][p2][p3][p4]$ 表示仅考虑最后 pos 位，从前面转移来的状态为 $p1\ p2\ p3\ p4$ 时答案的最大值。
1. $p1$ 为 1 表示比 L 大
2. $p2$ 为 1 表示比 R 小
3. $p3$ 为 1 表示比 A 小
4. $p4$ 为 1 表示比 B 小

然后在 dfs 的时候枚举 pos 位放的值，然后更新答案即可。代码中将每个数拆成了二进制进行计算，因为仅考虑 0 1 可以简化问题。

```
#include <bits/stdc++.h>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl
    ;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2
    << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const int N = 1e5+100;
const int M = 1e5+100;
const db EPS = 1e-9;
using namespace std;

ll f[70][2][2][2][2],a1[70],a2[70],a3[70],a4[70];
```

```

ll pow_mod(ll a,ll b){
    ll base = a, ans = 1;
    while(b){
        if(b&1) ans *= base;
        base *= base;
        b >>= 1;
    }
    return ans;
}

ll dfs(int pos,bool f1,bool f2,bool f3,bool f4){
    //比L大, 比R小, 比A小, 比B小
    if(pos == 0) return 0;
    ll ans = f[pos][f1][f2][f3][f4];
    if(ans != -1) return ans;

    int b1 = 0, b2 = 1, d1 = 0, d2 = 1, e1 = 0, e2 = 1;
    if(!f1 && a1[pos] != 0) b1 = 1;
    if(!f2 && a2[pos] != 1) b2 = 0;
    if(!f3 && a3[pos] != 1) d2 = 0;
    if(!f4 && a4[pos] != 1) e2 = 0;
    bool k1,k2,k3,k4;
    rep(i,b1,b2)
        rep(j,d1,d2)
            rep(k,e1,e2){
                k1 = f1, k2 = f2, k3 = f3, k4 = f4;
                ll x = (i^j)+(j&k)+(k^i);
                x *= pow_mod(2,pos-1);
                // printf("********\n");
                // LOG3("i",i,"j",j,"k",k);
                // LOG2("pos",pos,"x",x);
                if(i > a1[pos]) k1 = 1;
                if(i < a2[pos]) k2 = 1;
                if(j < a3[pos]) k3 = 1;
                if(k < a4[pos]) k4 = 1;
                ans = max(ans,x+dfs(pos-1,k1,k2,k3,k4));
            }
    return (f[pos][f1][f2][f3][f4] = ans);
}

ll solve(ll L,ll R,ll A,ll B){
    memset(a1,0,sizeof a1);
    memset(a2,0,sizeof a2);
    memset(a3,0,sizeof a3);
    memset(a4,0,sizeof a4);
    int pos = 0;
    while(L || R || A || B){
        ++pos;
        a1[pos] = L & 1;
        a2[pos] = R & 1;
        a3[pos] = A & 1;
        a4[pos] = B & 1;
        L /= 2; R /= 2; A /= 2; B /= 2;
    }
    return dfs(pos,0,0,0,0);
}

int main()
```

```
{
    int _; scanf("%d",&_);
    while(_--){
        rep(i,0,65)
            rep(j,0,1)
                rep(k,0,1)
                    rep(t1,0,1)
                        rep(t2,0,1)
                            f[i][j][k][t1][t2] = -1;
        ll L,R,A,B; scanf("%lld%lld%lld%lld",&L,&R,&A,&B);
        printf("%lld\n",solve(L,R,A,B));
    }
    return 0;
}
```

3.6.5 三进制状压

题意:
查找区间 $[l,r]$ 中，符合如下条件的数字的个数:
1. 所有出现过的偶数位，都出现了奇数次
2. 所有出现过的奇数位，都出现了偶数次
$(1 \le l \le r \le 10^{19})$

思路:
这个问题主要的困难之处在于如何表示那些没有出现过的数字，如果用二进制状压的方式显然无法表示那些从未出现过的数字。

因此我们考虑使用三进制状压的方式，0 表示没有出现过，1 表示出现了奇数次，2 表示出现了偶数次。

因此我们设置状态数组 $f[pos][S]$，表示还有 pos 个位置没有填数，之前填了的数字继承下来的状态为 S，符合题干条件的数的个数。

```
#include <bits/stdc++.h>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl
    ;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2
    << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const int N = 1e5+100;
const int M = 1e5+100;
const db EPS = 1e-9;
using namespace std;

ll f[21][60000],a[25],tp[25];

ll pow_mod(ll a,ll b){
    ll base = a, ans = 1;
    while(b){
        if(b&1) ans *= base;
        base *= base;
        b >>= 1;
    }
    return ans;
```

```
}

ll dfs(int pos,ll S,bool flag){ //S-各数位出现次数的状压
    //flag只是表示是否表示了该位的全部情况
    if(pos == 0){
        int jud = 1;
        memset(tp,0,sizeof tp); int tot = -1;
        while(S){
            tp[++tot] = S % 3;
            S /= 3;
        }
        rep(i,0,9){
            if(tp[i] == 0) continue;
            if(i%2){ //奇数
                if(tp[i] != 2) {jud = 0; break;}
            }
            else{ //偶数
                if(tp[i] != 1) {jud = 0; break;}
            }
        }
        if(jud) return 1;
        else return 0;
    }
    if(!flag && f[pos][S] != -1) return f[pos][S];
    int end = flag?a[pos]:9;
    ll ans = 0;
    rep(i,0,end){
        if(i == 0 && S == 0) ans += dfs(pos-1,S,flag && i == end); //不能把前面的前导0继承过来
        else{
            int tot = -1, p = 0; ll hp = S;
            while(hp){
                ++tot;
                if(tot == i) {p = hp%3; break;}
                hp /= 3ll;
            }
            hp = S;
            if(p == 0 || p == 1) hp += pow_mod(3,i);
            else hp -= pow_mod(3,i);
            ans += dfs(pos-1,hp,flag && i == end);
        }
    }
    // LOG3("pos",pos,"S",S,"ans",ans);
    if(!flag) f[pos][S] = ans;
    return ans;
}

ll solve(ll n){
    if(n == 0) return 1;
    int pos = 0;
    memset(a,0,sizeof a);
    while(n){
        a[++pos] = n % 10;
        n /= 10;
    }
    return dfs(pos,0,1);
}

int main()
```

```
{
    ll base = pow_mod(3,10);
    // LOG1("base",base);
    rep(i,0,20)
        rep(j,0,base)
            f[i][j] = -1;
    int _; scanf("%d",&_);
    while(_--){
        ll L,R; scanf("%lld%lld",&L,&R); L--;
        printf("%lld\n",solve(R)-solve(L));
    }
    return 0;
}
/*
数位DP主要就是数位移动时，对答案贡献的求取
在数位移动时，要仔细考虑各个细节点，包括前导0等信息
*/
```

3.6.6 数字平方和

题意:
寻找区间 $[l, r]$ 中所有与 7 无关的数字的平方和。与 7 有关需要符合下述三个条件之一:
1. 整数中某一位是 7
2. 整数的每一位加起来的和是 7 的整数倍
3. 这个整数是 7 的整数倍
$(1 \le l \le r \le 10^{18})$

思路:
如果这题只是单纯地求个数，那就是一个普通数位 dp 问题，但此题要求的是数字平方和，因此我们需要对每一个数位进行考虑。

当我们枚举 pos 位时，先递归到 $pos-1$ 位，返回一个结构体 B，表示若 $pos-1$ 位前全为空时满足题意的 cnt、$sum1$、$sum2$，即个数、和、平方和。

设当前结构体为 A，因此 $A.cnt = A.cnt + B.cnt$，$A.sum1 = A.sum1 + B.sum1 + B.cnt * 10^{pos-1}$，$A.sum2 = A.sum2 + B.sum2 + 10^{pos-1} * 10^{pos-1} + 2 * 10^{pos-1} * B.sum1$。

总结:
数位 dp 的主要难点还是在于列出状态，然后采用递归的思想来思考如何根据第 $pos-1$ 位的答案推出 pos 位的答案。

```
#include <bits/stdc++.h>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl
    ;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2
    << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const int N = 1e5+100;
const int M = 1e5+100;
const ll mod = 1e9+7;
const db EPS = 1e-9;
using namespace std;

struct Node{
```

```
    ll cnt,sum1,sum2; //个数、和、平方和
}dp[25][10][10];
ll a[30];

ll pow_mod(ll a, ll b, ll p){
    ll base = a, ans = 1;
    while(b){
        if(b&1) ans = (ans*base)%p;
        base = (base*base)%p;
        b >>= 1;
    }
    return ans;
}

Node dfs(int pos, int pre1, int pre2, bool flag){ //pre1: 前面数位之和, pre2: 前面数位*贡献之和
    //flag = 1, 前面为答案继承而来
    if(pos == 0){
        if(pre1 == 0 || pre2 == 0) return {0,0,0};
        else return {1,0,0};
    }
    if(!flag && dp[pos][pre1][pre2].cnt != -1) return dp[pos][pre1][pre2];
    Node ans = {0,0,0}, tmp;
    int end = flag?a[pos]:9;
    rep(i,0,end){
        if(i == 7) continue;
        tmp = dfs(pos-1, (pre1+i)%7, (pre2*10ll+i)%7, flag && i == end);
        ans.cnt = (ans.cnt+tmp.cnt)%mod;
        ans.sum1 = (ans.sum1+tmp.sum1+(tmp.cnt*(ll)i)%mod*pow_mod(10,pos-1,mod)%mod)%mod;
        ans.sum2 = (ans.sum2+tmp.sum2+(2ll*tmp.sum1*(ll)i)%mod*pow_mod(10,pos-1,mod)%mod)%mod;
        ans.sum2 = (ans.sum2+(ll)i*(ll)i*pow_mod(10,pos-1,mod)%mod*pow_mod(10,pos-1,mod)%mod*tmp.cnt%mod)%mod;
    }
    if(!flag) dp[pos][pre1][pre2] = ans;
    return ans;
}

ll solve(ll n){
    int pos = 0;
    memset(a,0,sizeof a);
    while(n){
        a[++pos] = n%(10ll);
        n /= 10ll;
    }
    return dfs(pos,0,0,1).sum2;
}

int main()
{
    rep(i,0,20)
        rep(j,0,6)
            rep(k,0,6) dp[i][j][k].cnt = -1;
    int _; scanf("%d",&_);
    while(_--){
        ll L,R; scanf("%lld%lld",&L,&R); L--;
        printf("%lld\n",(solve(R)-solve(L)+mod)%mod);
    }
```

```
    return 0;
}
```

3.6.7 被数位整除

题意:
一个正整数被称为漂亮数，当且仅当其能够被其所有非零数位整除，现需要求 $[l, r]$ 中漂亮数的个数。$(1 \le l \le r \le 9*10^{18})$

思路:
整除所有非零数位，即整除非零数位的 LCM。因此我们需要在 dfs 的过程中，记录出现数位的 LCM，以及记录当前数字的大小。

这里会出现一个问题，即当前数字非常大，直接记录不可行，因此我们需要将当前数字对 2520 取模，即 1 9 所有数的 LCM。思考到这一步，剩下的就是一些代码细节了，可以参考一下下述代码。

```
#include <bits/stdc++.h>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const int N = 1e5+100;
const int M = 1e5+100;
const db EPS = 1e-9;
using namespace std;

ll a[30],b[3000],tot,f[20][55][3000];

ll gcd(ll a,ll b){
    return b == 0 ? a:gcd(b,a%b);
}

int find(ll x){
    return lower_bound(b+1,b+1+tot,x)-b;
}

ll dfs(int pos,int lcm,ll m,bool flag){
    if(pos == 0){
        if(m%b[lcm] == 0) return 1;
        else return 0;
    }
    if(!flag && f[pos][lcm][m] != -1) return f[pos][lcm][m];
    int end = flag?a[pos]:9;
    ll ans = 0;
    rep(i,0,end){
        if(i == 0) ans += dfs(pos-1,lcm,(m*10ll+i)%(2520ll),flag && i == end);
        else{
            ll tp = gcd(b[lcm],i);
            ll hp = b[lcm]*i/tp;
            int xp = find(hp);
            ans += dfs(pos-1,xp,(m*10ll+i)%(2520ll),flag && i == end);
        }
    }
```

```
    if(!flag) f[pos][lcm][m] = ans;
    return ans;
}

ll solve(ll n){
    if(n == 0) return 1;
    int pos = 0;
    memset(a,0,sizeof a);
    while(n){
        a[++pos] = n % 10;
        n /= 10;
    }
    return dfs(pos,1,0,1);
}

int main()
{
    rep(i,1,((1<<9)-1)){
        ll ans = 1;
        rep(j,1,9){
            if(i&(1<<(j-1))){
                ll tp = gcd(ans,j);
                ans = ans*j/tp;
            }
        }
        b[++tot] = ans;
    }
    sort(b+1,b+1+tot);
    tot = unique(b+1,b+1+tot)-b-1;
    rep(i,0,19)
        rep(j,0,50)
            rep(k,0,2900) f[i][j][k] = -1;
    int _; scanf("%d",&_);
    while(_--){
        ll L,R; scanf("%lld%lld",&L,&R); L--;
        printf("%lld\n",solve(R)-solve(L));
    }
    return 0;
}
```

3.6.8 数位逆序对之和

题意:
给出区间 $[l,r]$，求所有数字中数位逆序对之和。$(1 \le l \le r \le 10^8)$

思路:
这是一个涉及到组合计数的数位 dp 问题，初次看到肯定觉得很棘手，但是我们可以将这个问题不断地进行分解。

我们先考虑 pos 个位置，每个位置可以填 $[0,9]$ 时，所有可能的数的数位逆序对之和。这个问题不难处理，直接从 pos 中取两个位置组成逆序队，其它位置任意取，答案为 $ans1(pos) = C(pos,2) * 45 * 10^{pos-2}$。

然后再考虑如果 pos 个位置，第一个位置不能为 0 时的答案。我们先计算第一个位置和之后位置的贡献，再计算后面 $pos-1$ 位置产生的贡献。因此 $ans2(pos) = 36 * C(pos-1,1) * 10^{pos-2} + 9 * ans1(pos-1)$。

处理完这两个子问题之后，我们直接按照数位 dp 的套路 dfs 求解即可。如果不是很清楚，可以查看代码进行进一步了解。

```cpp
#include <bits/stdc++.h>
#define rep(i,a,b) for(int i = a; i <= b; i++)
typedef long long ll;
using namespace std;

void dbg() {cout << "\n";}
template<typename T, typename... A> void dbg(T a, A... x) {cout << a << ' '; dbg(x...);}
#define logs(x...) {cout << #x << " -> "; dbg(x);}

ll a[30],POS;

//快速幂
ll calc(ll a,ll b){
    if(b < 0) return 0;
    ll ans = 1, base = a;
    while(b){
        if(b&1) ans *= base;
        base *= base;
        b >>= 1;
    }
    return ans;
}

//数字长度为pos且第一个数字不为0
ll calc1(ll pos){
    ll ans = 0;
    ans += 36ll*(pos-1ll)*calc(10,pos-2); //第一个数不为0，后续数字与其组成的贡献
    ans += 9ll*(pos-2ll)*(pos-1ll)*45ll*calc(10,pos-3)/2ll; //除第一个数字之外的贡献
    return ans;
}

//数字长度为pos且第一个数字可以为0
ll calc2(ll pos){
    return pos*(pos-1ll)*45ll*calc(10,pos-2)/2ll;
}

ll dfs(int pos,int *base,bool flag,ll hp){
    ll ans = 0;
    if(pos == 0){
        return hp;
    }
    if(!flag){
        ans = calc2(pos);
        rep(i,0,8) ans += (ll)base[i]*(ll)pos*(ll)(9-i)*calc(10,pos-1);
        ans += hp*calc(10,pos);
        return ans;
    }
    int end = a[pos];
    ll tmp = 0;
    rep(i,0,end){
        if(POS == pos && i == 0) continue; //保证起始位不为0
        if(i > 0) tmp += (ll)base[i-1];
        base[i]++;
        ans += dfs(pos-1,base,flag && i == end,hp+tmp);
        base[i]--;
    }
    return ans;
}
```

```

ll solve(ll n){
    if(n <= 9) return 0;
    POS = 0; memset(a,0,sizeof a);
    while(n){
        a[++POS] = n%10ll;
        n /= 10ll;
    }
    int base[11];
    rep(i,0,9) base[i] = 0;
    ll ans = dfs(POS,base,1,0);
    rep(i,2,POS-1) ans += calc1(i);
    return ans;
}

int main(){
    int _; scanf("%d",&_);
    rep(Ca,1,_){
        ll x,y; scanf("%lld%lld",&x,&y);
        x--;
        printf("Case %d: %lld\n",Ca,solve(y)-solve(x));
    }
}
```

3.7 概率 dp

题意:
给出一个有向无环图，从 1 号点向 n 号点前进。每次等概率地转移到后继节点，也有可能留在当前节点。定义每一天的花费是当前走过的天数，即第 i 天的花费是 i，求从起点走到终点的总花费。$(2 \le n \le 10^5, 1 \le m \le 2 * 10^5)$

思路:
令 $day[i]$ 表示从第 i 个点到达第 n 个点的期望天数，$cost[i]$ 表示从第 i 个点到达第 n 个点的期望花费，$deg[i]$ 表示第 i 个点的出度。

$$day[i] = \sum_{j \in deg[i]} \frac{day[j]}{deg[i]+1} + \frac{day[i]}{deg[i]+1} + 1$$
$$day[i] = \sum_{j \in deg[i]} \frac{day[j]}{deg[i]} + \frac{deg[i]+1}{deg[i]}$$

然后再来求取 $cost[i]$。

$$cost[i] = \sum_{j \in deg[i]} \frac{cost[j]}{deg[i]+1} + \frac{cost[i]}{deg[i]+1} + day[i]$$
$$cost[i] = \sum_{j \in deg[i]} \frac{cost[j]}{deg[i]} + \frac{deg[i]*(deg[i]+1)}{deg[i]}$$

为什么求 $cost[i]$ 的最后要加上一个 $day[i]$，因为当前之后还有 $day[i]-1$ 天，每天的期望要加上一个 1，即当前走的这一步。因此 $day[i]-1+1 = day[i]$。

接下来就是把边反向跑拓扑序转移即可。

总结:
概率 dp 题目，主要难点在于考虑清所有的情况，精准地列出式子。

有时最终的式子也有可能由多个式子组成，需要求出多个变量，需要对问题进行一定的拆分。

```
#include <bits/stdc++.h>
#define mem(a,b) memset(a,b,sizeof a);
#define rep(i,a,b) for(int i = a; i <= b; i++)
```

```
#define per(i,a,b) for(int i = a; i >= b; i--)
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
typedef long long ll;
typedef double db;
const int N = 4e5+100;
const int M = 5e5+100;
const db EPS = 1e-9;
using namespace std;

void dbg() {cout << "\n";}
template<typename T, typename... A> void dbg(T a, A... x) {cout << a << ' '; dbg(x...)
    ;}
#define logs(x...) {cout << #x << " -> "; dbg(x);}

int n,m,tot,head[N],deg[N],base[N];
db day[N],cost[N];
struct Node{
    int to,next;
}e[M];

void add(int x,int y){
    e[++tot].to = y, e[tot].next = head[x], head[x] = tot;
}

void init(){
    tot = 1;
    rep(i,0,n){
        head[i] = deg[i] = 0;
        day[i] = cost[i] = 0;
    }
}

void solve(){
    queue<int> q;
    while(q.size()) q.pop();
    rep(i,1,n) base[i] = deg[i];
    rep(i,1,n)
        if(!deg[i]) q.push(i);
    while(q.size()){
        int x = q.front(); q.pop();
        if(base[x] != 0){
            day[x] += (db)(base[x]+1)/(db)base[x];
            cost[x] += ((db)day[x]*(db)(base[x]+1))/(db)base[x];
        }
        for(int i = head[x]; i; i = e[i].next){
            int y = e[i].to;
            deg[y]--;
            day[y] += (db)day[x]/(db)base[y];
            cost[y] += (db)cost[x]/(db)base[y];
            if(!deg[y]) q.push(y);
        }
    }
    printf("%.2f\n",cost[1]);
}

int main()
{
    int _; scanf("%d",&_);
    while(_--){
```

```
        scanf("%d%d",&n,&m);
        init();
        rep(i,1,m){
            int u,v; scanf("%d%d",&u,&v);
            deg[u]++; add(v,u);
        }
        solve();
    }
    return 0;
}
```

3.8 斜率优化 dp

3.8.1 斜率优化例题

题意：

n 个需要被处理的任务，机器启动时间为 s。每个任务都有时间 Ti 和花费 Ci，计算方法为完成这批任务所需的时间是各个任务需要时间的总和。注意，同一批任务将在同一时刻完成，新的一批任务开始时，机器需要重新启动。确定一个分组方案，使得总费用最小。【每批任务包含相邻的若干任务】

思路：

1 <= n <= 3*1e5，0 <= S,Ci <= 512，-512 <= Ti <= 512。

首先我们需要列出 dp 方程，dp[i] 表示完成前 i 个任务的最小费用。此处由于涉及到 s，因此我们需要用到一个叫做“费用提前计算”的思想。因为 s 对于后续的每个任务都有影响，因此我们应当将 s 对后续任务的影响提前计算。

假设 dp[i] 由 dp[j] 更新而来，则 dp[i] = dp[j] + sumT[i]*(sumC[i]-sumC[j]) + s*(sumC[N]-sumC[j]); 将转移方程拆开，则可以得到 dp[j] - s*sumC[j] = sumT[i]*sumC[j]+dp[i]-s*sumC[N]-sumT[i]*sumC[i]，由此我们可以发现令横坐标 x = sumC[j]，纵坐标 y = dp[j]-s*sumC[j]，则对于每一个 j，都可以在平面中确定一个 (x,y) 坐标，而直线的斜率也是固定的，为 sumT[i]。

因此在平面中的这么多点中，我们需要维护一个下凸壳，更新 dp[i] 的时候，只需要在下凸壳中，二分 x 点与 x+1 点的斜率是否小于 sumT[i]，找到一个点，该点左边的斜率小于 k，右边斜率大于 k，则该点即为该下凸壳中的最优点。

再谈一下如何维护下凸壳，用一个数组，末尾为 qt，则判断 qt-1 与 x 的斜率是否小于等于 qt-1 与 qt 的斜率，如果小于等于，则需要弹出 qt，然后不断继续往下更新。注意此处如果是等于也需要弹出。

本题在维护凸壳时，可能会爆 long long，需要转成 long double。

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#define rep(i,a,b) for(int i = a; i <= b; i++)
using namespace std;
const int N = 1e6+100;
typedef long long ll;

int n,qt;
ll s,T[N],C[N],sumT[N],sumC[N],dp[N],q[N],xx[N],yy[N];

void calc(int x)    //在凸壳中二分最优点，斜率优化
{
    int l = 1, r = qt-1, ans = q[l];
    ll k = sumT[x];
    while(l <= r)
    {
```

```
        int mid = (l+r)>>1;
        if((long double)(yy[q[mid+1]]-yy[q[mid]]) <= (long double)(xx[q[mid+1]]-xx[q[mid]])*(long double)k)
            ans = q[mid+1], l = mid+1;
        else r = mid-1;
    }
    dp[x] = yy[ans]-k*xx[ans]+s*sumC[n]+sumT[x]*sumC[x];
}

void solve()
{
    qt = 1, q[1] = 0;
    xx[0] = yy[0] = 0;
    rep(i,0,n) xx[i] = 0, yy[i] = 0;
    rep(i,1,n)
    {
        calc(i);
        xx[i] = sumC[i], yy[i] = dp[i]-s*sumC[i];
        while(qt >= 2){
            //重点在于 >=, > 会wa
            if((long double)(yy[q[qt]]-yy[q[qt-1]])*(long double)(xx[i]-xx[q[qt-1]]) >= (long double)(yy[i]-yy[q[qt-1]])*(long double)(xx[q[qt]]-xx[q[qt-1]]))
                qt--;
            else break;
        }
        qt++; q[qt] = i;
    }
    printf("%lld\n",dp[n]);
}

int main()
{
    while(~scanf("%d%lld",&n,&s))
    {
        sumT[0] = sumC[0] = 0;
        rep(i,1,n){
            scanf("%lld%lld",&T[i],&C[i]);
            sumT[i] = sumT[i-1]+T[i];
            sumC[i] = sumC[i-1]+C[i];
        }
        solve();
    }
    return 0;
}
```

3.8.2 动态维护下凸壳

题意:

给定一颗树，单向边，给出每个点的价值。然后任选树中一个点 i 进入，从 j 点出来，获得的价值为 val[i]*1+val[i+1]*2+...+val[j]*(j-i+1)，也可以选择不进入，节点价值有正有负，求最多可以获得多少价值。

思路:

这题可以简化成在一个序列上,找出一段 [x,y],求 SUM(val[i]*(i-x+1)) 的最大值,即 1*val[1]+2*val[2]+3*val[3]+... 的问题。

我们可以列一下 dp 方程，dp[i] 表示以 i 为右端点的区间和最大值，假设 dp[i] 的最优解是 [j+1, i] 这一段，则 dp[i] = f[i]-f[j]-(sum[i]-sum[j])*j，sum[i] 表示从根节点到 i 路径上所有点的权值之和，f[i] 表示从根节点到 i 路径上所有点按照 1*val[1]+2*val[2]+3*val[3] 的方式得到的累加和。

将 dp 方程拆成斜率优化的形式，可以得到 f[j]-j*sum[j] = -sum[i]*j+f[i]-dp[i]，而 dp[i] 就是答案。因此令横坐标为 j，纵坐标为 f[j]-j*sum[j]，在图中标出这些点，维护一个下凸壳，然后在下凸壳中二分左边斜率最接近-sum[i]的点即为最优点。因此问题变成了如何维护下凸壳，正常的维护方法是用单调栈，添加一个点之后就将其他的点一一弹出，直到该点到达合理位置。但由于现在是在树上维护凸壳，因此下凸壳的序列会不断变化，因此我们需要进行动态维护。

可以发现每次往下凸壳中加入一个点，这个点最终都会到达原凸壳中的某一个位置，因此每次添加一个点，只是 O(1) 修改，然后我们在修改完之后再改回来，然后再维护原来的凸壳长度，即可实现动态维护。如何求该点最终到达的位置，只需要进行二分，找到点 x 与 x+1 之间斜率最接近 x 与新点之间斜率的点 x，x+1 处即为新点最终的更新位置。因此本题就是两个二分 +dfs 结束。

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#define rep(i,a,b) for(int i = a; i <= b; i++)
using namespace std;
typedef long long ll;
const int N = 1e5+1000;

int n,q[N],fa[N],head[N],tot,qh,qt;
ll val[N],ans,x0[N],y0[N];
struct Edge{
    int to,next;
}e[N];

void init()
{
    tot = 1;
    rep(i,0,n) head[i] = 0;
}

void add(int x,int y)
{
    e[++tot].to = y, e[tot].next = head[x], head[x] = tot;
}

bool jud(int a,int b,int c)
{
    return ((y0[a]-y0[b])*(x0[c]-x0[b]) > (y0[c]-y0[b])*(x0[a]-x0[b]));
}

int calc1(int l,int r,int x)    //维护凸壳
{
    int tp = r+1; r = r-1;
    while(l <= r)
    {
        int mid = (l+r)>>1;
        if(jud(q[mid+1],q[mid],x)) tp = mid+1, r = mid-1;
        else l = mid+1;
    }
    return tp;
}

void calc2(int l,int r,ll k,ll f)   //斜率优化dp,找到第一个右边斜率比k大的点
{
    int tp = q[l]; r--;
    while(l <= r)
    {
```

```
        int mid = (l+r)>>1;
        if(y0[q[mid+1]]-y0[q[mid]] < k*(x0[q[mid+1]]-x0[q[mid]])) tp = q[mid+1], l = mid+1;
        else r = mid-1;
    }
    ans = max(ans,-y0[tp]+k*x0[tp]+f);
}

void dfs(int x,int dep,ll sum,ll f) //从1号点往下dfs
{
    x0[x] = dep, y0[x] = f-dep*sum;
    calc2(qh,qt,-sum,f);
    int oldh = qh, oldt = qt;
    int pos = calc1(qh,qt,x), old = q[pos]; q[pos] = x, qt = pos;
    for(int i = head[x]; i; i = e[i].next)
    {
        int y = e[i].to;
        dfs(y,dep+1,sum+val[y],f+(dep+1)*val[y]);
    }
    qh = oldh, qt = oldt, q[pos] = old;
}

int main()
{
    int T; scanf("%d",&T);
    while(T--)
    {
        init();
        ans = 0;
        scanf("%d",&n);
        rep(i,1,n) scanf("%lld",&val[i]);
        rep(i,2,n) scanf("%d",&fa[i]), add(fa[i],i);
        qh = qt = 1; q[1] = 0;
        dfs(1,1,val[1],val[1]);
        printf("%lld\n",ans);
    }
    return 0;
}
```

3.9 斯坦纳树

3.9.1 斯坦纳树模板

适用问题：一个图中，有若干个关键点，将这几个关键点连在一起的最小花费。直观的理解，就是带关键节点的最小生成树。

时间复杂度：$O(n*3^k+c*|E|*2^k)$。n 为点数，$|E|$ 为边数，k 为关键点数，c 为 $spfa$ 常数，前一部分为子集枚举的复杂度，第二部分为枚举边松弛的复杂度。

```
#include <bits/stdc++.h>
#define rep(i,a,b) for(int i = a; i <= b; i++)
const int N = 30+5;
const int K = 10;
const int inf = 1e8;
using namespace std;

int dp[N][1<<K], st[N], endS;
bool vis[N][1<<K];
queue<int> q;
```

```cpp
//dp[i][state]: 以i为根，关键点在当前斯坦纳树中的连通状态为state的最小花费。

//初始化函数
void init(){
    //定义末状态
    endS = (1<<K)-1;

    //初始化
    rep(i,0,n)
        rep(S,0,endS) dp[i][S] = inf, vis[i][S] = 0;

    //求每个点的状态，假设前k个点为特殊点
    rep(i,1,k) st[i] = 1<<(i-1);

    //dp函数关键点赋初值
    rep(i,1,k) dp[i][st[i]] = 0;
}

//斯坦纳树第二部分转移
void SPFA(int state){
    while(q.size()){
        int x = q.front(); q.pop(); vis[x][state] = 0;
        //枚举连边
        for(int i = head[x]; i; i = e[i].next){
            int y = e[i].to;
            //松弛操作
            if(dp[y][st[y]|state] > dp[x][state]+e[i].w){
                dp[y][st[y]|state] = dp[x][state]+e[i].w;
                //状态保持一致或已经在队列中了
                if((st[y]|state) != state || vis[y][state]) continue;
                vis[y][state] = 1;
                q.push(y);
            }
        }

    }
}

//斯坦纳树主函数
void steinerTree(){
    rep(S,1,endS){
        rep(i,1,n){
            //i为关键节点，判断i是否在状态S中
            if(st[i] && (S|st[i]) != S) continue;
            //第一部分转移，枚举子集
            for(int sub = S&(S-1); sub; sub = (sub-1)&S){
                int x = st[i]|sub, y = st[i]|(S-sub);
                if(dp[i][x] != inf && dp[i][y] != inf)
                    dp[i][S] = min(dp[i][S],dp[i][x]+dp[i][y]);
            }
            if(dp[i][S] != inf)
                q.push(i), vis[i][S] = 1;
        }
        //第二部分转移，对于每个状态进行一次转移
        SPFA(S);
    }
}
```

3.9.2 斯坦纳森林

题意：n 个点，m 条边，前 k 个点为居住处，后 k 个点为避难处。现在 k 个居民需要从居住处逃避到避难处，问最少的连边的花费。注意只要满足居民有避难处可待即可，因此是个森林。$(1 \le n \le 50, 0 \le m \le 1000, 1 \le k \le 5, 2*k \le n)$

思路：这其实是个斯坦纳森林模板题。首先正常的求出斯坦纳树的结果，即 $dp[i][state]$ 表示第 i 个点为树根，关键点连通状态为 $state$ 的最小花费。

然后再设置一个新的函数 $f[1 << K]$，$f[state]$ 表示关键节点连通状态为 $state$ 时的最小代价。由于答案可以是森林，因此枚举 $state$ 的子状态进行转移即可。

$dp[i][state] = min(dp[i][state], dp[i][sub] + dp[i][state - sub])$。这里需要注意要检验状态 $state$ 是否合法，$state$ 状态合法当且仅当该状态中居住处点个数与避难处点个数相同。

```
#include <bits/stdc++.h>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl
    ;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2
    << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const db EPS = 1e-9;
const int N = 50+5;
const int M = 2*1000+100;
const int inf = 1e8;
using namespace std;

int n,m,k,head[N],dp[N][1<<10],f[1<<10],endS,st[N],tot;
bool vis[N][1<<10];
struct Node{
    int to,next,w;
}e[M];
queue<int> q;

void add(int x,int y,int w){
    e[++tot].to = y, e[tot].next = head[x], head[x] = tot, e[tot].w = w;
}

void init(){
    tot = 1;
    rep(i,0,n) head[i] = 0;
    endS = (1<<(2*k))-1;
    rep(i,0,n) memset(vis[i],0,sizeof vis[i]);
    rep(i,0,n) st[i] = 0;
    rep(i,0,n)
        rep(S,0,endS) dp[i][S] = inf, vis[i][S] = 0;
    rep(i,1,k) st[i] = 1<<(i-1);
    rep(i,n-k+1,n) st[i] = 1<<(i-(n-k)+k-1);
    rep(i,1,n)
        if(st[i]) dp[i][st[i]] = 0;
    rep(S,0,endS) f[S] = inf;
}

void SPFA(int state){
    while(q.size()){
```

```
        int x = q.front(); q.pop(); vis[x][state] = 0;
        for(int i = head[x]; i; i = e[i].next){
            int y = e[i].to;
            // LOG2("x",x,"y",y);
            if(y == x) continue;
            if(dp[y][st[y]|state] > dp[x][state]+e[i].w){
                dp[y][st[y]|state] = dp[x][state]+e[i].w;
                if((st[y]|state) != state || vis[y][state]) continue;
                q.push(y); vis[y][state] = 1;
            }
        }
    }
}

void steinerTree(){
    rep(S,1,endS){
        rep(i,1,n){
            //i在S中
            if(st[i] && (st[i]|S) != S) continue;
            //枚举子集
            for(int sub = (S-1)&S; sub; sub = (sub-1)&S){
                int x = st[i]|sub, y = st[i]|(S-sub);
                // LOG2("x",x,"y",y);
                if(dp[i][x] != inf && dp[i][y] != inf)
                    dp[i][S] = min(dp[i][S],dp[i][x]+dp[i][y]);
            }
            if(dp[i][S] != inf){
                q.push(i), vis[i][S] = 1;
                // LOG3("i",i,"S",S,"dp[i][S]",dp[i][S])
            }
        }
        SPFA(S);
    }
}

int check(int state){
    int cnt = 0;
    rep(i,0,2*k-1){
        if(state&(1<<i))
            cnt += i<k?1:-1;
    }
    return cnt == 0 ? 1:0;
}

void solve(){
    rep(S,1,endS)
        if(check(S)){
            // LOG1("S",S);
            rep(i,1,n)
                f[S] = min(f[S],dp[i][S]);
        }
    rep(S,1,endS)
        for(int sub = S&(S-1); sub; sub = (sub-1)&S){
            int x = sub, y = S-sub;
            if(check(x) && check(y))
                f[S] = min(f[S],f[x]+f[y]);
        }
    if(f[endS] == inf) printf("No solution\n");
    else printf("%d\n",f[endS]);
```

```
}

int main()
{
    int _; scanf("%d",&_);
    while(_--){
        scanf("%d%d%d",&n,&m,&k);
        init();
        rep(i,1,m){
            int x,y,w; scanf("%d%d%d",&x,&y,&w);
            add(x,y,w); add(y,x,w);
        }
        steinerTree();
        solve();
    }
    return 0;
}
```

3.9.3 斯坦纳树路径输出

题意：$n*m$ 的格子，每个格子上的值表示这个点上的志愿者个数，若该点为 0，表示该点为景区。问最少需要多少个志愿者，可以连通所有的景区，需要输出路径。$(1 \le n, m, k \le 10)$

思路：答案直接上板子即可得到。（板子还是理解完记住比较好，养成学一个记住一个的好习惯）。路径输出的话，我们考虑一下 dp 的两部分转移方程。

可以发现 $(i, j, state)$ 构成了一个状态，(i, j) 表示坐标。每个状态可能由一个或两个子状态转移而来，因此我们用 $pre[i][j][state]$ 结构体记住两个子状态，输出路径的时候 dfs 递归所有子状态进行格点标记即可。

```
#include <bits/stdc++.h>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl
    ;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2
    << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const db EPS = 1e-9;
const int N = 11;
const int inf = 1e8;
using namespace std;

int n,m,a[N][N],st[N][N],cnt,dp[N][N][1<<N],endS,mp[N][N];
bool vis[N][N][1<<N];
int dir[4][2] = {{-1,0},{1,0},{0,1},{0,-1}};
queue<pair<int,int> > q;
struct Node{
    int x,y,s1,s2;
    Node() {x = y = s1 = s2 = 0;}
    Node(int a,int b,int c,int d):x(a),y(b),s1(c),s2(d) {}
}pre[N][N][1<<N];

void init(){
    rep(i,1,n)
        rep(j,1,m){
```

```
            scanf("%d",&a[i][j]);
            if(a[i][j] == 0) st[i][j] = 1<<cnt, cnt++;
        }
    endS = (1<<cnt)-1;
    rep(i,0,n)
        rep(j,0,m)
            rep(S,0,endS) dp[i][j][S] = inf, vis[i][j][S] = 0;
    //DP赋初值
    rep(i,0,n)
        rep(j,0,m)
            if(st[i][j]) dp[i][j][st[i][j]] = 0;
}

void SPFA(int state){
    while(q.size()){
        int x = q.front().first, y = q.front().second; q.pop(); vis[x][y][state] = 0;
        rep(i,0,3){
            int tx = x+dir[i][0], ty = y+dir[i][1];
            if(tx < 1 || tx > n || ty < 1 || ty > m) continue;
            if(dp[tx][ty][state|st[tx][ty]] > dp[x][y][state]+a[tx][ty]){
                dp[tx][ty][state|st[tx][ty]] = dp[x][y][state]+a[tx][ty];
                //更新前驱
                pre[tx][ty][state|st[tx][ty]] = {x,y,state,state};
                if((state|st[tx][ty]) != state || vis[tx][ty][state]) continue;
                q.push(make_pair(tx,ty)); vis[tx][ty][state] = 1;
            }
        }
    }
}

void steinertree(){
    rep(S,1,endS){
        rep(i,1,n)
            rep(j,1,m){
                if(st[i][j] && (st[i][j]|S) != S) continue;
                for(int sub = S&(S-1); sub; sub = (sub-1)&S){
                    int x = sub|st[i][j], y = st[i][j]|(S-sub);
                    if(dp[i][j][x] != inf && dp[i][j][y] != inf){
                        if(dp[i][j][S] > dp[i][j][x]+dp[i][j][y]-a[i][j]){
                            dp[i][j][S] = dp[i][j][x]+dp[i][j][y]-a[i][j];
                            pre[i][j][S] = {i,j,x,y};
                        }
                    }
                }
                if(dp[i][j][S] != inf)
                    q.push(make_pair(i,j)), vis[i][j][S] = 1;
            }
        SPFA(S);
    }
}

void dfs(int x,int y,int state){
    if(vis[x][y][state] || x < 1 || x > n || y < 1 || y > m || !state) return;
    mp[x][y] = 1; vis[x][y][state] = 1;
    int tx = pre[x][y][state].x, ty = pre[x][y][state].y, s1 = pre[x][y][state].s1, s2
    = pre[x][y][state].s2;
    if(s1 != s2){
        dfs(tx,ty,s1); dfs(tx,ty,s2);
    }
```

```
    else{
        dfs(tx,ty,s1);
    }
}

void solve(){
    int ans = inf, xp = 0, yp = 0;
    rep(i,1,n)
        rep(j,1,m)
            if(dp[i][j][endS] < ans){
                ans = dp[i][j][endS]; xp = i, yp = j;
            }
    memset(vis,0,sizeof vis);
    dfs(xp,yp,endS);
    printf("%d\n",ans);
    rep(i,1,n){
        rep(j,1,m){
            if(a[i][j] == 0) printf("x");
            else if(mp[i][j] == 1) printf("o");
            else printf("_");
        }
        printf("\n");
    }
}

int main()
{
    scanf("%d%d",&n,&m);
    init();
    steinertree();
    solve();
    return 0;
}
```

3.10 长链剖分

3.10.1 k 级祖先

题意：
N 个点的一棵树，M 次询问，每次询问给出一个 x、k，表示询问点 x 的 k 级祖先。$(1 \le N \le 3*10^5, 1 \le M \le 1800000)$

思路：
k 级祖先是长链剖分的经典问题，主要应用的性质是“任意一个点的 k 级祖先 y 所在的长链的长度大于等于 k”。

因此我们对于每个长链的端点维护两个 $vector$，$down[x]$ 中存储了点 x 所属长链上的所有节点，$up[x]$ 中存储了点 x 向上寻找能够寻找到的 $min(d[x], dep[x])$ 个点。此处 $dep[x]$ 表示节点 x 在树中的深度，$d[x]$ 表示以 x 为端点的长链的长度。

因此我们查询点 x 的 k 级祖先时，只需要判断点 x 所在长链的端点距 x 的距离是否大于 k，如果大于 k，则需要使用点 x 长链端点的 up 数组，否则使用 $down$ 数组。大致思路是这样，具体的实现过程中还有一些其它的细节，详见代码。

```
#include <bits/stdc++.h>
#define rep(i,a,b) for(int i = a; i <= b; i++)
typedef long long ll;
const int N = 3*1e5+100;
const int M = 6*1e5+100;
const db EPS = 1e-9;
using namespace std;
```

```

int n,m,tot,head[N],top[N],son[N],len[N],fa[N],f[N][25],d[N],hbit[N];
vector<int> up[N], down[N];
struct Edge{
    int to,next;
}e[M];

void add(int x,int y){
    e[++tot].to = y, e[tot].next = head[x], head[x] = tot;
}

void dfs1(int x,int y){
    len[x] = 1; d[x] = d[y]+1; f[x][0] = fa[x];
    for(int i = 1; (1<<i) <= d[x]; i++)
        f[x][i] = f[f[x][i-1]][i-1];
    for(int v,i = head[x]; i; i = e[i].next)
        if((v = e[i].to) != y){
            fa[v] = x;
            dfs1(v,x);
            if(len[son[x]] < len[v]) son[x] = v;
        }
    len[x] = len[son[x]]+1;
}

void dfs2(int x,int y){
    top[x] = y;
    if(son[x]) dfs2(son[x],y);
    for(int v,i = head[x]; i; i = e[i].next)
        if((v = e[i].to) != fa[x] && v != son[x]) dfs2(v,v);
}

int query(int x,int k){
    if(k == 0) return x;
    if(d[x] < k+1) return 0;
    int r = f[x][hbit[k]];
    if(d[r] == d[x]-k) return r;
    else{
        int an = top[r];
        if(d[an] <= d[x]-k){
            int len = d[x]-k-d[an];
            return down[an][len];
        }
        else{
            int len = d[an]-d[x]+k;
            return up[an][len];
        }
    }
}

int main()
{
    scanf("%d",&n); tot = 1;
    rep(i,1,n-1){
        int x,y; scanf("%d%d",&x,&y);
        add(x,y); add(y,x);
    }
    scanf("%d",&m);
    dfs1(1,0);
    dfs2(1,1);
```

```
    int ans = 0;
    rep(i,1,n)
        if(top[i] == i){
            int x = i;
            while(x) {down[i].push_back(x); x = son[x];}
            int ct = len[i];
            x = i;
            while(x && ct) {up[i].push_back(x); x = fa[x]; ct--;}
        }
    rep(i,1,300000)
        rep(j,0,20)
            if(i & (1<<j)) hbit[i] = max(hbit[i],j);
    rep(i,1,m){
        int x,y; scanf("%d%d",&x,&y);
        x ^= ans; y ^= ans;
        ans = query(x,y);
        printf("%d\n",ans);
    }
    return 0;
}
```

3.10.2 O(1) 优化树形 dp 模板

题意：
n 个点的一棵树，对于树上每个点 x，求出一个 $ans[x]$，表示在 x 的子树中，$ans[x]$ 深度处点的个数最多，若个数相同，令下标最小。$(1 \le N \le 10^6)$

思路：
这是一道长链剖分的经典例题，也可以说是模板题。可以查看这道题的代码，来学会长链剖分的基础操作。

这道题思路比较经典，就是 $O(1)$ 继承重儿子，暴力继承轻儿子，更新的时候记录最大值即可。

```
#include <bits/stdc++.h>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl
    ;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2
    << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const db EPS = 1e-9;
const int N = 1e6+100;
using namespace std;

int n,tot,head[N],ans[N],d[N],son[N];
int *f[N],tmp[N],*id=tmp;
struct Node{
    int to,next;
}e[2*N];

void add(int x,int y){
    e[++tot].to = y, e[tot].next = head[x], head[x] = tot;
}

void dfs1(int x,int fa){
```

```
    for(int i = head[x]; i; i = e[i].next){
        int y = e[i].to;
        if(y == fa) continue;
        dfs1(y,x);
        if(d[y] > d[son[x]]) son[x] = y;
    }
    d[x] = d[son[x]]+1;
}

void dfs2(int x,int fa){
    f[x][0] = 1;
    if(son[x]){
        f[son[x]] = f[x]+1;
        dfs2(son[x],x);
        ans[x] = ans[son[x]]+1;
    }
    for(int i = head[x]; i; i = e[i].next){
        int y = e[i].to;
        if(y == fa || y == son[x]) continue;
        f[y] = id; id += d[y];
        dfs2(y,x);
        rep(j,0,d[y]-1){
            f[x][j+1] += f[y][j];
            if(f[x][j+1] > f[x][ans[x]] || (f[x][j+1] == f[x][ans[x]] && j+1 < ans[x]))
 ans[x] = j+1;
        }
    }
    if(f[x][ans[x]] == 1) ans[x] = 0;
}

int main()
{
    scanf("%d",&n); tot = 1;
    rep(i,1,n-1){
        int x,y; scanf("%d%d",&x,&y);
        add(x,y); add(y,x);
    }
    dfs1(1,0);
    f[1] = id; id += d[1];
    dfs2(1,0);
    rep(i,1,n) printf("%d\n",ans[i]);
    return 0;
}
/*
题意：n个点的一棵树，对于树上每个点x，求出一个ans[x]，表示在x的子树中，ans[x]深度处点的个数最多。
*/
```

3.10.3 长链剖分维护后缀和

题意：
n 个点的一棵树，有 q 组询问，每组询问给出一个 p、k，询问有多少个有序三元组 (p,b,c) 满足要求。其中 p、b、c 分别为树上三个不同的点，其中 p 和 b 都是 c 的祖先，且 p 与 b 之间的距离小于等于 k。$(1 \le n \le 3*10^5, 1 \le q \le 3*10^5, 1 \le k \le n)$

思路：
首先我们来分析一下这个三元组的性质。(p,b,c)，p 和 b 是 c 的祖先，p 与 b 之间距离小于等于 k。因此 b 可能是 p 的儿子，也可能是 p 的祖先。当 b 是 p 的祖先时，$ans = ans + min(dep[p]-1,k)*(sz[p]-1)$；当 b 是 p 的儿子时，$ans = ans + sz[b]-1$。

因此我们可以发现，b 是 p 的祖先的情况好处理，因此我们只需要处理 b 是 p 的儿子时情况即可。而 b 是 p 的儿子时，对答案的贡献就是 p 子树中所有节点 $sz[b]-1$ 的和。

因此我们定义 $dp[u][j]$ 表示节点 u 中所有距 u 距离大于等于 j 的节点 $size-1$ 之和。

$$dp[u][j+1] = dp[u][j+1] + dp[v][j]$$

$$dp[u][0] = dp[u][1] + sz[u] - 1$$

然后我们再将所有查询离线，接着进行长链剖分的常规操作，$O(1)$ 继承重儿子，暴力继承轻儿子即可完成此题。

这里需要注意，由于长链剖分的数组只能暂时保存每个节点状态的值，因此我们需要将所有查询离线进行预先处理。

```
#include <bits/stdc++.h>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl
    ;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2
    << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const int N = 3*1e5+100;
const db EPS = 1e-9;
using namespace std;

int n,q,tot,head[N],son[N],d[N],dep[N];
ll *f[N],tmp[N],*id=tmp,sz[N],ans[N];
struct Node{
    int to,next;
}e[2*N];
vector<pair<int,int> > Q[N];

void add(int x,int y){
    e[++tot].to = y, e[tot].next = head[x], head[x] = tot;
}

void dfs1(int x,int fa){
    sz[x] = 1; dep[x] = dep[fa]+1;
    for(int i = head[x]; i; i = e[i].next){
        int y = e[i].to;
        if(y == fa) continue;
        dfs1(y,x); sz[x] += sz[y];
        if(d[y] > d[son[x]]) son[x] = y;
    }
    d[x] = d[son[x]]+1;
}

void dfs2(int x,int fa){
    if(son[x]){
        f[son[x]] = f[x]+1;
        dfs2(son[x],x);
    }
    for(int i = head[x]; i; i = e[i].next){
        int y = e[i].to;
        if(y == son[x] || y == fa) continue;
```

```
        f[y] = id; id += d[y];
        dfs2(y,x);
        for(int j = 0; j < d[y]; j++)
            f[x][j+1] += f[y][j];
    }
    for(auto &v:Q[x]){
        int num = v.first, k = v.second;
        ans[num] = (sz[x]-1ll)*(ll)min(dep[x]-1,k);
        if(k >= d[x]-1) ans[num] += f[x][1]-f[x][d[x]-1];
        else ans[num] += (ll)f[x][1]-(ll)f[x][k+1];
    }
    f[x][0] = f[x][1]+sz[x]-1ll;
}

int main()
{
    scanf("%d%d",&n,&q); tot = 1;
    rep(i,1,n-1){
        int u,v; scanf("%d%d",&u,&v);
        add(u,v); add(v,u);
    }
    dfs1(1,0);
    f[1] = id; id += d[1];
    rep(i,1,q){
        int p,k; scanf("%d%d",&p,&k);
        Q[p].push_back(make_pair(i,k));
    }
    dfs2(1,0);
    rep(i,1,q) printf("%lld\n",ans[i]);
    return 0;
}
```

3.10.4 长链剖分 + 线段树

题意：
n 个点的一棵树，树上每条边都有一个权值。现要在树中找到一条长度在 $[L,R]$ 范围内的一条路径，使得 $AvgValue = \frac{\sum_{e\in S} v(e)}{|S|}$ 最大，即道路平均价值最大。$(1 \le n \le 10^5, 1 \le L \le R \le n-1, v_i \le 10^6)$

思路：
看完题干中的道路平均价值 $AvgValue = \frac{\sum_{e\in S} v(e)}{|S|}$ 最大，应该就能反应过来这是一道 01 分数规划问题。而 01 分数规划问题，解法比较固定，即二分答案，然后令 $Sum = \sum_{e\in S}(v(e)-ans)$，其中 ans 为你二分的答案。因此将图中所有边的边权减去 ans，然后在图中找一条长度在 $[L,R]$ 范围内的路径，使得 Sum 最大。若 $Sum \ge 0$，则二分右半部分，否则二分左半部分。

因此我们的问题就变成了在树中找一条长度在 $[L,R]$ 范围内的路径，使得路径权值最大。

涉及到了路径长度，因此我们考虑树形 dp，定义 $dp[u][j]$ 表示从节点 u 向其子树出发，路径长度为 j 时的最大路径权值和。

$$dp[u][j+1] = max(dp[u][j+1], dp[v][j] + e[i].w)$$

$$ans = max(ans, dp[v][j] + dp[u][x] + e[i].w)$$

$$L \le 1 + j + x \le R$$

从上述的 dp 方程中，我们可以发现我们需要在 $dp[u]$ 的一个范围内寻找最大值，由此想到了利用线段树来寻找最值。

因此我们在原先的长链剖分数组上维护一个线段树，每个节点保存该节点到根的距离，由此 $dp[u][x] = dis[u][x] - dis[u]$，就可以避免继承重儿子信息时的区间修改等操作。如若定义 $dp[u][x]$ 为 u 到深度为 x 的节点的距离，则在继承重儿子信息时，无可避免地需要用到区间修改。而定义 $dp[u][x]$ 为根到 u 深度为 x 的节点的距离的话，即可避免修改问题。

```
#include <bits/stdc++.h>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const int N = 1e5+100;
const db EPS = 1e-5;
const db inf = 1e14;
using namespace std;

int n,L,R,head[N],tot,d[N],son[N];
db *f[N],tmp[N],*id = tmp, ans, maxn[2*N], dis[N];
struct Node{
    int to,next;
    ll wt;
    db w;
}e[2*N];
inline int get_id(int l,int r) {return (l+r)|(l!=r);}

void add(int x,int y,ll w){
    e[++tot].to = y, e[tot].next = head[x], head[x] = tot, e[tot].wt = w;
}

void build(int l,int r){
    maxn[get_id(l,r)] = -inf;
    if(l == r) return;
    int mid = (l+r)>>1;
    build(l,mid); build(mid+1,r);
}

void insert(int l,int r,int pos,db v){
    int now = get_id(l,r);
    if(l == r) {maxn[now] = max(maxn[now],v); return;}
    int mid = (l+r)>>1;
    if(pos <= mid) insert(l,mid,pos,v);
    else insert(mid+1,r,pos,v);
    maxn[now] = max(maxn[get_id(l,mid)],maxn[get_id(mid+1,r)]);
}

db query(int l,int r,int ls,int rs){
    if(rs < ls) return -inf;
    int now = get_id(l,r);
    if(ls <= l && r <= rs) return maxn[now];
    int mid = (l+r)>>1;
    db tmp = -inf;
    if(ls <= mid) tmp = max(tmp,query(l,mid,ls,rs));
    if(rs > mid) tmp = max(tmp,query(mid+1,r,ls,rs));
```

```
    return tmp;
}

void dfs1(int x,int fa){
    for(int i = head[x]; i; i = e[i].next){
        int y = e[i].to;
        if(y == fa) continue;
        dfs1(y,x);
        if(d[y] > d[son[x]]) son[x] = y;
    }
    d[x] = d[son[x]]+1;
}

void dfs2(int x,int fa){
    int pos = f[x]-tmp;
    f[x][0] = dis[x];
    insert(1,n,pos+1,f[x][0]);
    for(int i = head[x]; i; i = e[i].next){
        if(son[x] == e[i].to){
            f[son[x]] = f[x]+1;
            dis[son[x]] = dis[x]+e[i].w;
            dfs2(son[x],x);
        }
    }
    for(int i = head[x]; i; i = e[i].next){
        int y = e[i].to;
        if(y == fa || y == son[x]) continue;
        f[y] = id; id += d[y];
        dis[y] = dis[x]+e[i].w;
        dfs2(y,x);
        rep(j,0,d[y]-1){ //找最大值
            if(j+1 <= R){
                db base = query(1,n,max(1,pos+L-j),min(pos+d[x],pos+R-j))-2.0*dis[x];
                ans = max(ans,base+f[y][j]);
            }
        }
        rep(j,0,d[y]-1){ //更新最值
            if(f[y][j] > f[x][j+1]){
                f[x][j+1] = f[y][j];
                insert(1,n,pos+j+2,f[x][j+1]);
            }
        }
    }
    db base = query(1,n,pos+L+1,min(pos+d[x],pos+R+1))-dis[x];
    ans = max(ans,base);
}

int check(db base){
    ans = -inf;
    rep(i,2,tot) e[i].w = (db)e[i].wt-base;
    build(1,n);
    rep(i,0,n) tmp[i] = 0, dis[i] = 0;
    id = tmp; f[1] = id; id += d[1];
    dfs2(1,0);
    if(ans-0 >= EPS) return 1;
    else return 0;
}

int main()
```

```
{
    scanf("%d",&n); tot = 1;
    scanf("%d%d",&L,&R);
    rep(i,1,n-1){
        int x,y; ll w; scanf("%d%d%lld",&x,&y,&w);
        add(x,y,w); add(y,x,w);
    }
    dfs1(1,0);
    db l = 1, r = 1000000;
    while(r-l >= EPS){
        db mid = (l+r)/2.0;
        if(check(mid)) l = mid;
        else r = mid;
    }
    printf("%.3f\n",r);
    return 0;
}
```

3.10.5 三元组两两距离相等

题意：
n 个点的一棵树，求树上的三元无序集合 (a,b,c)，满足集合中任意两点距离相同的条件的集合个数。$(1 \le n \le 10^5)$

思路：
这是一道树形 dp 用长链剖分优化的典型问题，主要难点在于树形 dp 的状态方程如何设置。

我们首先要思考如何才能不重不漏地计算所有的集合个数。而想要不重不漏地计算个数，就必须要找到一个不变量。而在一棵树中，最常见的不变量就是 lca。因此我们对于树上每一个点，枚举其作为 lca 时对答案的贡献。由于 lca 是固定的，所以枚举 lca 可以避免重复计算。

然后我们考虑对于一个点作为 lca 时，它对答案的贡献是什么？首先对于树形 dp，我们一定是从左往右枚举子节点的一个过程，当枚举到节点 v 时，对答案的贡献要么是左边子树中的两个节点 + v 子树中的一个节点，不然就是左边子树中的一个节点 + v 子树中的两个节点。只有这两种情况需要进行枚举。

思考到这一步，我们的 dp 状态方程就好定了。首先令 $f[u][j]$ 表示距节点 u 距离为 j 的节点个数，由此我们可以解决子树中一个节点的情况。然后考虑子树中两个节点时，如果设置状态？

由于我们需要两个子节点之间的距离，因此我们令 $g[u][j]$ 表示 u 子树中，(a,b) 点对数量。点对满足条件为 a、b 到其 lca 距离均为 d，而其 lca 到 u 距离为 $d-j$，d 为任意值。这样设置状态的原因是，$d-j+j=d$，这样就可以满足三点距离相同了。至此，状态便设置完成，然后考虑如何书写转移方程。

$$f[u][j] = f[u][j] + f[v][j-1]$$

$$g[u][j] = g[u][j] + g[v][j+1]$$

上述两条转移式子比较显然，但是对于 g 数组来说，我们只考虑了 $d-j$ 不为 0 的情况，即 u 不为 $lca(a,b)$ 的情况，因此还需要考虑 $u = lca(a,b)$ 的情况，即 (a,b) 分别位于 u 的两个子节点的子树中。

$$g[u][j] = g[u][j] + f[u][j] * f[v][j-1]$$

至此，我们考虑完了 f、g 两个数组的转移方程。接下来我们需要考虑对答案的贡献如何表示。如上所述，每一个节点作为三点 lca 时，对答案的贡献为左子树两点 + 当前子节点子树中一点，或左子树一点 + 当前子节点子树中两点。由此可以得到下述方程。

$$ans = ans + f[u][j-1] * g[v][j] + g[u][j+1] * f[v][j]$$

现在，我们得到了此题完整的四个转移方程。接下来只有最后一个注意点就是，由于 $g[u][j] = g[u][j] + g[v][j+1]$，因此 g 数组我们需要开两倍，避免子节点转移时下标 -1 影响到了其它节点的区间造成越界。

```
#include <bits/stdc++.h>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl
    ;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2
    << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const db EPS = 1e-9;
const int N = 1e5+100;
using namespace std;

int n,tot,head[N],son[N],d[N];
ll *f[N], *g[N], tf[N], tg[2*N], *id1 = tf+1, *id2 = tg+1, ans;
struct Node{
    int to,next;
}e[2*N];

void add(int x,int y){
    e[++tot].to = y, e[tot].next = head[x], head[x] = tot;
}

void dfs1(int x,int fa){
    for(int i = head[x]; i; i = e[i].next){
        int y = e[i].to;
        if(y == fa) continue;
        dfs1(y,x);
        if(d[y] > d[son[x]]) son[x] = y;
    }
    d[x] = d[son[x]]+1;
}

void dfs2(int x,int fa){
    f[x][0] = 1;
    if(son[x]){
        f[son[x]] = f[x]+1; g[son[x]] = g[x]-1;
        dfs2(son[x],x);
    }
    ans += g[x][0];
    for(int i = head[x]; i; i = e[i].next){
        int y = e[i].to;
        if(y == son[x] || y == fa) continue;
        f[y] = id1, g[y] = id2+d[y];
        id1 += d[y], id2 += d[y]*2;
        dfs2(y,x);
        rep(j,0,d[y]-1){
            ans += g[x][j+1]*f[y][j];
            if(j >= 1) ans += f[x][j-1]*g[y][j];
        }
        rep(j,0,d[y]-1){
```

```
            g[x][j+1] += f[x][j+1]*f[y][j];
            f[x][j+1] += f[y][j];
            if(j >= 1) g[x][j-1] += g[y][j];
        }
    }
}

int main()
{
    scanf("%d",&n); tot = 1;
    rep(i,1,n-1){
        int a,b; scanf("%d%d",&a,&b);
        add(a,b); add(b,a);
    }
    dfs1(1,0);
    f[1] = id1, g[1] = id2+d[1]; id1 += d[1], id2 += d[1]*2;
    dfs2(1,0);
    printf("%lld\n",ans);
    return 0;
}
```

3.11 巧妙 dp 题

3.11.1 亏欠型 DP

题意:
一条街上一共 N 个点，需要在某些点上建路灯，使得整条街被照亮，一个路灯可以照亮左右两个点，每个点都有一个建路灯的花费。现在你还有 K 次机会，可以交换两个路灯的建造费用，求使得整条街被照亮的最小花费。$(1 \le N \le 250000, 0 \le k \le 9)$

思路:
如果这道题没有 K 次交换路灯花费的操作的话，问题将会变得简单很多，只需要记录每个点某尾和前一个点的状态即可递推求解。

但是此题有了 K 次交换路灯的条件。仔细观察，可以发现，交换路灯花费时，一定是在便宜的那个点上建路灯，在贵的那个点上不建路灯，交换两个都要建路灯的节点是没有意义的。因此我们考虑将建灯花费先欠着，即对于点 i 来说，我们让这个点灯亮，但是具体花费先欠着，等到后面一个点的时候再还这个费用。

因此我们构建 DP 状态，$dp[i][j][k][l]$ 表示第 i 个点，欠了 j 次，还了 k 次，l 表示节点最后两个点的状态。$l = 0$，亮 + 不亮。$l = 1$，不亮 + 亮。$l = 2$，(亮/不亮)+ 亮。

然后就可以进行转移了，具体转移过程见代码，并不复杂，主要难点在于这个思想。

总结:
亏欠型 DP 的核心关键点在于“欠”与“还”。“欠”对应着预支花费，“还”对应着预付花费，可以有效的解决交换花费的问题。

这种方法的重要思想是“预支”，与此类似的还有一种思想是“反悔”，就是我先把这个值拿过来，如果之后发现更好的，我再把它扔掉。(如网络流中的反向边)

```
#include <bits/stdc++.h>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl
    ;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2
    << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
```

```
typedef double db;
const db EPS = 1e-9;
const int N = 250000+100;
using namespace std;

int n,K;
ll dp[N][10][10][3],w[N];

int main()
{
    scanf("%d%d",&n,&K);
    rep(i,1,n) scanf("%lld",&w[i]);
    memset(dp,0x3f,sizeof dp);
    dp[0][0][0][0] = 0; //第i个位置，欠了j个，还了k个
    rep(i,1,n)
        rep(j,0,K){
            rep(k,0,K){
                dp[i][j][k][0] = dp[i-1][j][k][2];
                if(k > 0) dp[i][j][k][0] = min(dp[i][j][k][0],dp[i-1][j][k-1][2]+w[i]);

                dp[i][j][k][1] = dp[i-1][j][k][0];
                if(k > 0) dp[i][j][k][1] = min(dp[i][j][k][1],dp[i-1][j][k-1][0]+w[i]);


                rep(h,0,2)
                    dp[i][j][k][2] = min(dp[i][j][k][2],dp[i-1][j][k][h]+w[i]);
                if(j > 0){
                    rep(h,0,2)
                        dp[i][j][k][2] = min(dp[i][j][k][2],dp[i-1][j-1][k][h]);
                }
            }
        }
    ll ans = 1e17;
    rep(i,0,K)
        ans = min(ans,min(dp[n][i][i][0],dp[n][i][i][2]));
    printf("%lld\n",ans);
    return 0;
}
```

3.11.2 数位 DP+ 状压 DP

题意:
给出一个 n，现可以从 1 n 中任意选取若干个数字，要求所有数字不存在重复的数位。例如$1,2,3$ 和$2,11$ 是合法的，但是$1,2,10$ 和$2,5,12$ 是不合法的。

现给出一个 n，询问存在多少个集合满足上述要求，结果模 $1e9+7$。$(1 \le n \le 10^9)$

思路:
首先，不难想到此题需要将 0 ~ 9 的数位状压起来，然后再进行数位 dp 求解，最后再进行一个类似于容斥或背包的操作求出答案。

我们定义 $num[i]$ 表示 1 n 中有多少个数字，数字中包含的 0 ~ 9 的状态恰好为 i。再定义 $ans[i]$ 表示 1 n 中有多少个不同的满足条件的集合，集合中数位的状态恰好为 i。

因此我们先求 $num[i]$，定义数位 dp 的状态为 $dp[S][pos]$ 表示长度为 pos 的数位，数位状态为 S 的数的个数。

然后数位 dp 经典操作 dfs 求取 $num[i]$，中间用 $dp[S][pos]$ 进行记忆化搜索。枚举第 pos 位的数位时，答案既可以由 $dp[S][pos-1]$ 更新而来，也可以由 $dp[S\ xor\ (1<<i)][pos-1]$ 更新而来。因为 pos 位已经提供了 i 这个数位，因

此之后的数位可以提供也可以不提供，因此有两种情况。

所以我们可以只用这个二维的状态即可求解出 $num[i]$。比赛时数位 dp 部分写了一个比较复杂的三维状态，最后卡时通过。

然后我们再来考虑如何求取 $ans[i]$。两种考虑方法，一种是利用的背包的想法，一种是利用组合的想法。

先讲背包的想法。我们已经求出了 $num[i]$ 表示 $1 \quad n$ 中数位状态为 i 的数的个数。然后可以将 $num[i]$ 中的每一个 i 看成一个物品，然后 $ans[i][j]$ 表示利用前 i 个物品组成的状态为 j 的方案总数。

```
ans[0][0] = 1;
for(nt i = 0; i < 1024; i++)
    for(int j = 0; j < 1024; j++){
        ans[i+1][j] += ans[i][j];
        if((j&i) == 0){
            ans[i+1][j|i] += ans[i][j] * num[i];
        }
    }
```

再讲组合容斥的想法。定义 $ans[i][j]$ 表示有多少个集合，组成了 i 这个状态，一共含有 j 个不同的数位状态。比如数位 12 这个状态，既可以由 1 和 2 两个状态组成，也可以直接由 12 这一个状态组成。然后我们就可以直接枚举子集进行计算。这里需要注意，枚举子集时，每次由一个单体和一个集合一起算贡献，而不是每次用两个集合算贡献，这样可以尽量避免重复。利用单体算贡献，每个单体被重复计算 k 次，因此最后答案除 k。

```
rep(S, 1, endS){
     rep(k, 2, 9){
         ll cnt = 0;
         for(int sub = S & (S - 1); sub; sub = (sub - 1) & S){
             int x = sub, y = S - sub;
             cnt = (cnt + DP[x] * ans[y][k - 1]) % mod;
         }
         cnt = cnt * inv[k] % mod;
         ans[S][k] = (ans[S][k] + cnt) % mod;
    }
 }
```

完整代码如下。

```
#include <bits/stdc++.h>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define mem(a,b) memset(a,b,sizeof a);
#define rep(i, a, b) for(int i = a; i <= b; i++)
#define LOG1(x1, x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1, x2, y1, y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl;
#define LOG3(x1, x2, y1, y2, z1, z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const db EPS = 1e-9;
const ll mod = 1e9 + 7;
using namespace std;

ll dp[1 << 11][13], endS, ans[1 << 11][10], DP[1 << 11], inv[100];
int len, a[13];

void init() {
    mem(dp,-1);
    mem(ans,0);
```

```
    mem(DP,0);
}

ll dfs(int pos, int state, bool flag, int jud) { //jud表示有无数字
    if (pos == 0) {
        if(state == 0 && jud == 1) return 1;
        else return 0;
    }
    if (!flag && dp[state][pos] != -1 && jud == 1) return dp[state][pos];
    ll base = 0;
    int end = flag ? a[pos] : 9;
    rep(i, 0, end){
        if(jud == 0 && i == 0){ //无数字
            base = (base + dfs(pos - 1, state, flag && i == end, 0)) % mod;
        }
        else{
            if (((1 << i) & state) == 0) continue;
            base = (base + dfs(pos - 1, state, flag && i == end, 1)) % mod;
            base = (base + dfs(pos - 1, state^(1<<i), flag && i == end, 1)) % mod;
        }
    }
    if (!flag && jud == 1) dp[state][pos] = base;
    return base;
}

void solve(ll n) {
    len = 0;
    memset(a, 0, sizeof a);
    while (n) {
        a[++len] = n % (10ll);
        n /= 10ll;
    }
    rep(S, 1, endS) DP[S] = dfs(len,S,1,0);
}

ll pow_mod(ll a, ll b, ll m) {
    ll ans = 1;
    while (b) {
        if (b & 1)ans = ans * a % mod;
        a = a * a % mod;
        b >>= 1;
    }
    return ans;
}

int main() {
    int _;
    scanf("%d", &_);
    for (int i = 1; i <= 100; ++i) {
        inv[i] = pow_mod(i, mod - 2, mod);
    }
    endS = (1 << 11) - 1;
    rep(Ca, 1, _) {
        init();
        ll n;
        scanf("%lld", &n);
        solve(n);
        rep(S, 1, endS) ans[S][1] = DP[S];
        rep(S, 1, endS){
```

```
            rep(k, 2, 9){
                ll cnt = 0;
                for(int sub = S & (S - 1); sub; sub = (sub - 1) & S){
                    int x = sub, y = S - sub;
                    cnt = (cnt + DP[x] * ans[y][k - 1]) % mod;
                }
                cnt = cnt * inv[k] % mod;
                ans[S][k] = (ans[S][k] + cnt) % mod;
            }
        }
        ll sum = 0;
        rep(S, 1, endS) rep(i, 1, 9) sum = (sum + ans[S][i]) % mod;
        printf("Case %d: %lld\n", Ca, sum);
    }
    return 0;
}
```

4 数据结构

4.1 单调栈

给出一个序列，长度为 n。定义区间价值为区间和 * 区间最小值，求出这个序列中的最大区间价值。($n \le 10^5, 0 \le ai \le 10^6$)

回忆一下单调栈和单调队列，单调栈——对于序列中每个点，求出序列中左/右边第一个比它大/小的点，单调队列——对于序列中每个点，求出序列中距离该点 K 步范围内的最小/大值。

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <stack>
#include <algorithm>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const int N = 1e5+100;
const int M = 1e5+100;
const db EPS = 1e-9;
using namespace std;

int n,a[N],tt1[N],tt2[N];
ll sum[N];
stack<int> st;

int main()
{
    scanf("%d",&n);
    rep(i,1,n) scanf("%d",&a[i]);
    rep(i,1,n) tt1[i] = 0, tt2[i] = n+1;
    st.push(1);
    rep(i,2,n){
        while(st.size() && a[i] < a[st.top()]){
            int x = st.top();
            st.pop();
            tt2[x] = i;
            // LOG1("x",x);
        }
        if(st.size()) tt1[i] = st.top();
        st.push(i);
    }
    rep(i,1,n) sum[i] = sum[i-1]+(ll)a[i];
    ll ans = 0;
    int l = 1,r = 1;
    rep(i,1,n){
        if(ans < a[i]*(sum[tt2[i]-1]-sum[tt1[i]])){
            ans = a[i]*(sum[tt2[i]-1]-sum[tt1[i]]);
            l = tt1[i]+1;
            r = tt2[i]-1;
        }
    }
```

```
    printf("%lld\n",ans);
    printf("%d %d\n",l,r);
    return 0;
}
```

4.2 单调队列

题意:

给定一串 01 串，m 次询问，每次询问给你一个数 k。k 为对于这个 01 串所能进行的最多次操作，每次操作可以将该串中任意一个位置的数移到任意一个其他的位置。

每次询问之后，输出在这个操作数之内，所能达到的最长的连续 0 的长度。

```
#include <cstdio>
#include <iostream>
#include <algorithm>
#include <cstring>
#define rep(i,a,b) for(int i = a;i <= b;i++)
using namespace std;
const int N = 1e6+1000;

char s[N];
int m,n,a[N],sum[N],q[N],cnt;

void solve(int k)
{
    int ans = 0;
    q[1] = 0;
    int l = 1, r = 1;
    rep(i,1,n)
    {
        while(l <= r && a[q[r]] >= a[i]) r--;   //维护单调队列，先对右端点进行更新
        q[++r] = i;
        while(l <= r && sum[i]-sum[q[l]] > k) l++;  //再对左端点进行更新
        ans = max(ans,a[q[r]]-a[q[l]]+k);
    }
    printf("%d\n",min(ans,cnt));
}

int main()
{
    while(~scanf("%s",s))
    {
        cnt = 0;
        scanf("%d",&m);
        int len = strlen(s);
        n = len;
        sum[0] = 0;a[0]=0;
        rep(i,1,len)
        {
            if(s[i-1] == '0'){
                sum[i] = sum[i-1];
                cnt++;
            }
            else sum[i] = sum[i-1]+1;
            a[i] = i-2*sum[i];
```

```

        }
        rep(i,1,m)
        {
            int x;
            scanf("%d",&x);
            solve(x);
        }
    }
    return 0;
}
```

4.3 并查集

4.3.1 普通并查集

```
#include <cstdio>
#include <iostream>
using namespace std;
const int maxn=10000+10;

int n,m,p[maxn];

int find(int x) {
    return p[x]==x?x:p[x]=find(p[x]);
}

void merge(int x,int y){
    int r1=find(x),r2=find(y);
    if(r1!=r2){
        p[r1]=r2;
    }
}

int main()
{
    cin>>n>>m;
    for(int i=1;i<=n;i++){
        p[i]=i;
    }
    for(int i=1;i<=m;i++){
        int x,y;
        cin>>x>>y;
        p[x]=y;
    }
    return 0;
}
```

4.3.2 带权并查集

题意：N 个人在玩剪刀石头布的游戏，其中有一个人是裁判，其余人随机被分到了三个阵营，即剪刀、石头、布。现在有 M 轮游戏，$x<,>,=y$ 表示 x 与 y 之间的关系，其中裁判可以自由变换阵营。问是否可以根据这 M 轮游戏，判断出谁是裁判，输出裁判是谁以及最早在哪一轮可以找到裁判。如果无法判断，则输出 $Can\ not\ determine$，如果游戏的情况不符合题意，则输出 $Impossible$ 。$(N \le 500, M \le 2000)$

思路：一开始做这题的时候，的确有些懵，只能想到如何发现有人变换了阵营，但是不知道如何找到这个人，并且确定这种情况是唯一的。

所以我们可以发现直接考虑整个问题会非常困难，再加上此题数据范围很小，直接做的话复杂度肯定很少，太对不起这个数据范围了。因此我们考虑 N^2 做法，即枚举每个人为裁判。枚举 x 为裁判时，如果 x 恰好为裁判，则并查集合并时不会发生矛盾。我们统计有多少个人为裁判时，并查集合并会发生矛盾，记人数为 cnt，并且统计每个人为裁判时，发生矛盾的轮数 ri 。

现在我们来考虑输出答案的所有情况。
1. 如果 $cnt = n - 1$，则可以唯一确定裁判，并且最早发现该裁判的轮数为 $max(ri)$。
2. 如果 $cnt = n$，则输出 $impossible$。
3. 其余情况则为 $Can\ not\ determine$。

至于带权并查集的合并类似于食物链问题，是个模 3 剩余系中的加减问题。

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl
    ;
typedef long long ll;
typedef double db;
const db EPS = 1e-9;
using namespace std;
const int N = 1000;
const int M  = 2000+100;
int n,m,fa[N],d[N],vis[N],error[N],cnt;

int find(int x){
    if(x == fa[x]) return x;
    int root = find(fa[x]);
    d[x] = (d[x]+d[fa[x]]+3)%3;
    return fa[x] = root;
}

struct Query{
    int x,y,cc;
}q[M];

int main()
{
    while(~scanf("%d%d",&n,&m))
    {
        memset(vis,0,sizeof vis);
        memset(error,0,sizeof error);
        cnt = 0;
        rep(i,1,m){
            int x,y; char cc;
            scanf("%d%c%d",&x,&cc,&y);
            q[i].x = x, q[i].y = y;
            if(cc == '=') q[i].cc = 0;
            else if(cc == '<') q[i].cc = 1;
            else q[i].cc = 2;
        }
        rep(i,0,n-1){
            rep(k,0,n) fa[k] = k, d[k] = 0;
            rep(j,1,m){
```

```
                int x = q[j].x, y = q[j].y, cc = q[j].cc;
                if(x == i || y == i) continue;
                int fx = find(x), fy = find(y);
                if(fx != fy){
                    fa[fx] = fy;
                    d[fx] = (d[y]-d[x]+cc+3)%3;
                }
                else{
                    int ttp = (d[x]-d[y]+3)%3;
                    if(ttp != cc){
                        cnt++;      // 产生矛盾个数
                        error[i] = j;
                        vis[i] = 1;
                        break;
                    }
                }
            }
        }
        // impossible
        // not determine
        // determine
        if(cnt == (n-1)) { //determine
            int ans = 0, hm;
            rep(i,0,n-1){
                ans = max(ans,error[i]);
                if(!vis[i]) hm = i;
            }
            printf("Player %d can be determined to be the judge after %d lines\n",hm,
    ans);
        }
        else if(cnt == n){ // impossible
            printf("Impossible\n");
        }
        else{  // not determine
            printf("Can not determine\n");
        }
    }
    return 0;
}
```

4.3.3 带权并查集与背包

题意：给出 $p1$ 个天使，$p2$ 个魔鬼，一共有 n 个问题。每个问题的格式为 x y (yes or no) 表示问 x y 是否为天使，如果 x 为天使，则会说真话，如果 x 为魔鬼，则会说假话。问根据这 n 个问题，是否可以确定哪些人为天使，如果可以确定，按编号大小输出天使编号。($n \leq 1000$，$p1, p2 \leq 300$)

思路：遇到这样的 yes or no 问题，显然我们需要对于可能出现的情况进行模拟。
1. 假如 x 为天使，y 为天使，则回答 yes 。
2. 假如 x 为天使，y 为魔鬼，则回答 no 。
3. 假如 x 为魔鬼，y 为天使，则回答 no 。
4. 假如 x 为魔鬼，y 为魔鬼，则回答 yes 。

可以发现，如果回答是 yes，则 x 与 y 属于同一类。如果回答 no，则 x 与 y 类别相反。因此一个模 2 剩余系的带权并查集合并就可以维护所有人之间的关系。

因此不难发现，处理完 n 个问题之后，我们会拥有若干个并查集，每个并查集中都会分为两批人，两批人类别不同。

因此问题变成了，从这若干个并查集中随机选一部分，使得最后选中的人数恰好为 $p1$，并且这种选择方式唯一，则我们可以确定哪些人为天使。因此本题就变成了一个类似背包的问题，$dp[i][j]$ 表示前 i 个并查集选取人数为 j 一共有多少种选择方案，$pre[i][j] = x$ 表示 $dp[i][j]$ 由 $dp[i-1][x]$ 更新而来。到此，本题即可顺利解决。

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <map>
#include <algorithm>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl;
typedef long long ll;
typedef double db;
const db EPS = 1e-9;
using namespace std;
const int N = 700;

int n,p1,p2,fa[N],d[N],base[N][2],tot,dp[N][600],pre[N][600],vis[N];
map<int,int> mp;

int find(int x)
{
    if(x == fa[x]) return x;
    int root = find(fa[x]);
    d[x] = (d[x]+d[fa[x]]+2)%2;
    return fa[x] = root;
}

void solve()
{
    memset(base,0,sizeof base);
    memset(vis,0,sizeof vis);
    mp.clear();
    tot = 0;
    rep(i,1,p1+p2){
        fa[i] = find(i);
        if(mp.find(fa[i]) == mp.end()) mp[fa[i]] = ++tot;
        int pos = mp[fa[i]];
        base[pos][d[i]]++;  //0: 相同   1: 不同
    }
    memset(dp,0,sizeof dp);
    memset(pre,0,sizeof pre);
    dp[1][base[1][1]]++;
    dp[1][base[1][0]]++;
    rep(i,2,tot){
        int minn = min(base[i][0],base[i][1]);
        rep(j,minn,p1){
            if(j >= base[i][0] && dp[i-1][j-base[i][0]] != 0)
                dp[i][j] += dp[i-1][j-base[i][0]], pre[i][j] = j-base[i][0];
            if(j >= base[i][1] && dp[i-1][j-base[i][1]] != 0)
                dp[i][j] += dp[i-1][j-base[i][1]], pre[i][j] = j-base[i][1];
        }
    }
    if(dp[tot][p1] != 1) printf("no\n");
    else{
        int xx = tot, yy = p1;
```

```
        while(xx > 1){
            if(pre[xx][yy] == yy-base[xx][0]) vis[xx] = 0, yy -= base[xx][0];
            else if(pre[xx][yy] == yy-base[xx][1]) vis[xx] = 1, yy -= base[xx][1];
            xx--;
        }
        if(xx == 1){
            if(base[1][0] == yy) vis[1] = 0;
            else vis[1] = 1;
        }
        rep(i,1,p1+p2){
            int pos = mp[fa[i]];
            int dd = d[i];
            if(vis[pos] == 1 && dd == 1) printf("%d\n",i);
            else if(vis[pos] == 0 && dd == 0) printf("%d\n",i);
        }
        printf("end\n");
    }
}

int main()
{
    while(~scanf("%d%d%d",&n,&p1,&p2))
    {
        if((n+p1+p2) == 0) break;
        rep(i,0,p1+p2) fa[i] = i, d[i] = 0;
        rep(i,1,n){
            int x,y; char op[10];
            scanf("%d%d",&x,&y);
            scanf("%s",op);
            int fx = find(x), fy = find(y);
            // LOG2("x",x,"y",y);
            // LOG2("fx",fx,"fy",fy);
            if(fx != fy){
                fa[fx] = fy;
                if(op[0] == 'n') d[fx] = (2+1+d[y]-d[x])%2;
                else d[fx] = (d[y]-d[x]+2)%2;
            }
        }
        solve();
    }
    return 0;
}
```

4.3.4 按秩合并并查集

题意：
n 个点，m 个操作，操作共两类。 $1\ u\ v$ 表示在图中加一条边连接 $u\ v$， $2\ u\ v$ 表示查询 u 与 v 最早是在哪一次加边操作后连通，不连通输 -1。$(1 \le n, m \le 10^5)$

思路：
维护连通性，最直观的想法就是用并查集来维护连通性。但是如何通过并查集来查看两点最早什么时候连通呢?

首先可以知道并查集维护的其实是一个森林，假如我们不破坏树的结构，即不进行路径压缩，则并查集每次加边，则将边权定义为操作编号，那么两点树上边权最大值就是两点最早连通的加边操作。

因此问题变成如何维护树的结构进行并查集合并，方法就是按秩合并，将小的树合并到大的树上，这样可以保证每个节点最多被合并 $log(n)$ 次，因此每个节点的高度最多为 $log(n)$。因此对于每次查询，我们可以将两个点到根节点的路

径直接取出，然后查询第一次遇到的位置，输出达到这个位置之前的边权最大值即可。

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const int N = 1e5+100;
const int M = 1e5+100;
const db EPS = 1e-9;
using namespace std;

int f[N],n,m,siz[N],d[N],flag[N],vis[N];

int find(int x){
    if(x == f[x]) return x;
    else return find(f[x]);
}

int solve(int u,int v){
    vis[u] = 1; flag[u] = 0; int w = 0;
    while(f[u] != u){
        w = max(w,d[u]);
        vis[f[u]] = 1; flag[f[u]] = w;
        u = f[u];
    }
    vis[u] = 1, flag[u] = w;
    w = 0;
    if(vis[v] == 1) return flag[v];
    while(f[v] != v){
        w = max(w,d[v]);
        if(vis[f[v]] == 1) return max(w,flag[f[v]]);
        v = f[v];
    }
    if(vis[v] == 1) return max(flag[v],w);
}

void clear(int u){
    while(f[u] != u) vis[u] = 0, flag[u] = 0, u = f[u];
    vis[u] = 0, flag[u] = 0;
}

int main()
{
    int _; scanf("%d",&_);
    while(_--){
        scanf("%d%d",&n,&m);
        rep(i,0,n) f[i] = i, d[i] = 0, siz[i] = 1;
        rep(i,1,m){
            int op,u,v; scanf("%d%d%d",&op,&u,&v);
            int xu = find(u), xv = find(v);
```

```
            if(op == 1 && xu != xv){
                if(siz[xu] < siz[xv]) f[xu] = xv, d[xu] = i, siz[xv] += siz[xu];
                else f[xv] = xu, d[xv] = i, siz[xu] += siz[xv];
            }
            else if(op == 2){
                if(xu != xv) printf("-1\n");
                else{
                    printf("%d\n",solve(u,v));
                    clear(u);
                }
            }
        }
    }
    return 0;
}
```

4.4 Hash

题意：
给出 n 个串，m 组询问。每组询问均为一个字符串，询问在初始 n 个串中是否存在一个串与该询问串恰好只有一个位置不相同。输出 YES or NO。

Hash 思路：
首先先讲讲 Hash 的算法，利用 bkdr 算法将每个字符串 hash 成一个数值，hash 函数如下：

hash: abccac
hash[1] = 0
hash[6] $= 0 * 131^5 + 1 * 131^4 + 2 * 131^3 + 2 * 131^2 + 0 * 131 + 2$

此处 hash[6] 即是这个字符串的 hash 值，131 为 seed，即 hash 种子，然后还要取一个模数，即 mod。由此可以发现 hash 值即是字符串中每一个位置的贡献，所以本题要求恰好只有一个位置不相同，即可以枚举不相同位置，减去原有贡献，加上新贡献即可。然后本题就可以解决，但是由于此题的数据卡的很 e xin，所以 seed 和 mod 取的不好的话，会被卡成 zz。

这里补充一下常见的 seed 和 mod，seed 一般取质数，3、5、7、13、131、13131 这些均可，mod 一般也取质数，1e9+7，1e11+7，1e13+7，1e18+7 均可，也可以直接将数据类型取为 unsigned long long，即可对 2^64-1 直接取模。

有一个注意点，在计算过程中，seed*mod 不能超过数据类型的最大值，否则相当于在计算过程中又模上了一个不是 mod 的数，会导致结果错误。

由于本题只有三个字符，所以可以将 seed 定为 3，mod 定一个很大的数，类似于三进制。

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <map>
#include <algorithm>
#define rep(i,a,b) for(int i = a; i <= b; i++)
using namespace std;
typedef long long ull;
const int N = 6*1e5+1000;
const ull mod = 1e11+7;
const ull ttp1 = 3;
const ull ttp2 = 5;
int n,m;
ull base[5] = {0,1,2,131313,123};
ull seed1[N],seed2[N];
```

```
char s[N];
struct Node{
    ull a,b;
}gn;
map<Node,int> mp;

bool operator == (Node x, Node y)
{
    if(x.a == y.a && x.b == y.b) return true;
    else return false;
}

bool operator < (Node x, Node y)
{
    return x.a < y.a;
}

int solve(ull hash1, ull hash2)
{
    int len = strlen(s);
    // printf("s:%s,len:%d\n",s,len);
    rep(j,0,len-1)
    {
        rep(k,0,2)
        {
            if(s[j]-'a' == k) continue;
            ull tmp1 = hash1;
            tmp1 = (tmp1+(-base[s[j]-'a']+base[k])*seed1[len-1-j]%mod)%mod;
            if(tmp1 < 0) tmp1 += mod;

            ull tmp2 = hash2;
            tmp2 = (tmp2+(-base[s[j]-'a']+base[k])*seed2[len-1-j]%mod)%mod;
            if(tmp2 < 0) tmp2 += mod;

            gn.a = tmp1, gn.b = tmp2;
            // printf("k:%d,tmp:%llu\n",k,tmp);
            if(mp[gn] == 1) return 1;
        }
    }
    return 0;
}

int main()
{
    // printf("mod:%lld\n",mod);
    mp.clear();
    seed1[0] = 1;
    seed1[1] = ttp1;
    seed2[0] = 1;
    seed2[1] = ttp2;
    int _ = 6*1e5+100;
    rep(i,2,_)
    {
        seed1[i] = (seed1[i-1]*ttp1)%mod;
        if(seed1[i] < 0) seed1[i] += mod;

        seed2[i] = (seed2[i-1]*ttp2)%mod;
        if(seed2[i] < 0) seed2[i] += mod;
    }
```

```
    scanf("%d%d",&n,&m);
    rep(i,1,n)
    {
        scanf("%s",s);
        int len = strlen(s);
        ull hash1 = 0;
        ull hash2 = 0;
        rep(j,0,len-1)
        {
            hash1 = (hash1*ttp1%mod+base[s[j]-'a'])%mod;
            if(hash1 < 0) hash1+=mod;

            hash2 = (hash2*ttp2%mod+base[s[j]-'a'])%mod;
            if(hash2 < 0) hash2+=mod;
        }
        // printf("s:%s,hash:%llu\n",s,hash);
        gn.a = hash1, gn.b = hash2;
        mp[gn] = 1;
    }
    rep(i,1,m)
    {
        scanf("%s",s);
        int len = strlen(s);
        ull hash1 = 0;
        ull hash2 = 0;
        rep(j,0,len-1)
        {
            hash1 = (hash1*ttp1%mod+base[s[j]-'a'])%mod;
            if(hash1 < 0) hash1+=mod;

            hash2 = (hash2*ttp2%mod+base[s[j]-'a'])%mod;
            if(hash2 < 0) hash2+=mod;
        }
        if(solve(hash1,hash2))
            printf("YES\n");
        else printf("NO\n");
    }
    return 0;
}

//mod: 1e9+7, seed: 257
//mod: 1e18+3, seed: 3、5
//mod: 1e11+7, seed: 3、5、1313、...
//如果mod*seed会越界的话，那么结果就会错误，因为计算过程中出现了两个seed，多了自动溢出的那个seed
```

4.5 KMP

一、KMP 求解过程

next 数组——最长前缀后缀匹配长度，需要小于串长度

$ababa$，$next[1] = 0, next[2] = 0, next[3] = 1, next[4] = 2, next[5] = 3$

求解问题：模式串在匹配串的哪些位置出现了，模式串是小串。

二、例题

(1) 求取 num 数组，表示最长的，不超过一半的，前缀与后缀相同的长度。$(1 \le n \le 10^6)$

一个显然的性质：$num[i] \le num[i-1]+1$
因此求取 $num[i]$ 时，先令 $num[i]=num[i-1]+1$，如果 $num[i]$ 超过一半的长度或者在 $num[i]$ 处发生失配，则直接跳 $next$ 数组。

(2) 给出 n 个串，每个串的长度为 m_i，询问这 n 个串的最长公共子串。此处两个子串相同的条件为，长度相同且其中一个子串加上一个数即可得到另一个子串。$(1 \le n \le 1000, 1 \le m_i \le 100)$

1. 在差分序列上进行 kmp 匹配。
2. 这里由于数据范围比较小，因此可以直接枚举后缀进行 kmp 匹配求最小值。
3. 但是多个串的最长公共子串其实是一个经典问题，可以用后缀数组或后缀自动机解决。

```
//预处理next数组，这里注意不要反复求strlen，这个函数是O(n)复杂度的，反复求会将复杂度变成 O(n^2)
next[0] = -1;
for(int i = 1, j; i <= n; i++){
  for(j = next[i-1]; j >= 0 && s1[j+1] != s1[i]; j = next[j]);
  next[i] = j+1;
}
//next[j] <= next[j-1]+1, 不会比前一位+1更大
//匹配过程
for(int i=1, j=0; i <= m; i++){
  for(; s2[i] != s1[j+1] && j >= 0; j = next[j]);
  j++;
  if(j==n) ans++;
}
```

4.6 字典树

```
#include <cstdio>
#include <iostream>
#include <cstring>
using namespace std;
const int charset = 26;
const int max_node = 100000+10;

struct Trie
{
    int tot,root,child[max_node][charset];  //root:根节点  tot:节点编号
    //child[i][j]=k, 表示编号为i的节点的第j个孩子是编号为k的节点
    bool flag[max_node];   //是否以某一个字符为结束
    Trie()
    {
//      memset(child,0,sizeof(child));
        memset(flag,0,sizeof(flag));
        root = tot = 1;  //根节点编号为1
    }
    void mem()  //将字典树初始化
    {
        memset(child,0,sizeof(child));
        memset(flag,0,sizeof(flag));
        root = tot = 1;  //根节点编号为1
    }
    void insert(const char *str)
    {
        int cur = root;
        for(int i = 0;str[i];i++)
        {
            int x = str[i]-'a';
```

```
            if(child[cur][x] == 0)
                child[cur][x] = ++tot;
            cur = child[cur][x];
        }
    //  flag[cur] = true; 记录单词以该点结束
    }
    bool query(const char *str)
    {
        int cur = root;
        for(int i = 0;str[i];i++)
        {
            int x = str[i]-'a';
            if(child[cur][x] == 0) return false;
            cur = child[cur][x];
        }
        return true;
        //查询单词时应该 return flag[cur];
    }
}tre;
```

4.7 ST 表

```
int st[N][25];

//数组范围为[0,len-1]
void init(){
    for(int i = 0; i < len; i++) st[i][0] = a[i];
    for(int j = 1; (1<<j) <= len; j++){
        for(int i = 0; i + (1<<j) - 1 < len; i++)
            st[i][j] = min(st[i][j-1],st[i+(1<<(j-1))][j-1]);
    }
}

//求[l,r]之间的最小值
int query(int l,int r){
    int k = (int)(log((double)(r - l + 1)) / log(2.0));
    return min(st[l][k],st[r-(1<<k)+1][k]);
}
```

4.8 树状数组

```
int c[N];

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

inline void update(int x, int c){
    for(int i = x; i <= k; i += lowbit(i)) t[i] += c;
}

inline int ask(int x){
    int tp = 0;
    for(int i = x; i; i -= lowbit(i)) tp += t[i];
    return tp;
}
```

4.9 线段树

4.9.1 动态开点与区间修改 lazy

```
#include <cstdio>
#include <algorithm>
#define rep(i,a,b) for(int i = a; i <= b; i++)
const int N = 1e5+100;
using namespace std;

int ls[2*N],rs[2*N],maxn[2*N],lazy[2*N],rt,sz,a[N],n; //rt: root, sz: 当前节点编号

void push_down(int now){
    if(ls[now] == 0) ls[now] = ++sz;
    if(rs[now] == 0) rs[now] = ++sz;
    maxn[ls[now]] += lazy[now], maxn[rs[now]] += lazy[now];
    lazy[ls[now]] += lazy[now], lazy[rs[now]] += lazy[now];
    lazy[now] = 0;
}
void update(int& now, int l, int r, int lx, int rx, int c){ //区间修改
    if(!now) now = ++sz;
    if(lx <= l && rx >= r){
        maxn[now] += c; lazy[now] += c;
        return;
    }
    if(lazy[now] != 0) push_down(now);
    int mid = (l+r)>>1;
    if(lx <= mid) update(ls[now],l,mid,lx,rx,c);
    if(rx > mid) update(rs[now],mid+1,r,lx,rx,c);
    maxn[now] = max(maxn[ls[now]],maxn[rs[now]]);
}
int query(int& now, int l, int r){  //查询最右边第一个值大于0的点
    if(!now) now = ++sz;
    if(maxn[now] <= 0) return -1;
    if(l == r) return a[l];
    if(lazy[now] != 0) push_down(now);
    int mid = (l+r)>>1;
    if(maxn[rs[now]] > 0) return query(rs[now],mid+1,r);
    else return query(ls[now],l,mid);
}

int main()
{
    scanf("%d",&n);
    rep(i,1,n){
        int pos,op; scanf("%d%d",&pos,&op);
        if(op == 1){ //push
            int xx; scanf("%d",&xx); a[pos] = xx;
            update(rt,1,n,1,pos,1);
        }
        else update(rt,1,n,1,pos,-1); //pop
        printf("%d\n",query(rt,1,n));
    }
    return 0;
}
```

4.9.2 区间递减序列和

题意：与楼房重建题意类似，但是求的是递减序列，而且询问的是区间 $[l, r]$ 的递减序列和。

思路：只需将维护内容的 $maxn$ 改为 $minn$ 即可，然后解决一下区间查询的问题。继续使用刚才的 $calc(now, tp)$ 函数，计算节点 now 在最小值为 tp 下的贡献。依然是左右区间二分，如果左区间最小值小于 tp，则直接计算右区间贡献，然后递归左区间。如果左区间最小值大于 tp，则直接递归右区间。

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const int N = 1e5+100;
const int M = 3*1e5+100;
const ll inf = 1e13;
const db EPS = 1e-9;
using namespace std;

int n,q,rt,sz,ls[M],rs[M];
ll a[N],sum[M],minn[M],ans;

ll calc(int &now,int l,int r,ll tp){
    if(!now) now = ++sz;
    if(l == r) return (minn[now]<tp?minn[now]:0ll);
    int mid = (l+r)>>1;
    if(minn[ls[now]] < tp) return (sum[now]-sum[ls[now]]+calc(ls[now],l,mid,tp));
    else return calc(rs[now],mid+1,r,tp);
}

void update(int &now,int l,int r,int pos,ll w){
    if(!now) now = ++sz;
    if(l == r){
        sum[now] = minn[now] = w;
        return;
    }
    int mid = (l+r)>>1;
    if(pos <= mid) update(ls[now],l,mid,pos,w);
    else update(rs[now],mid+1,r,pos,w);
    minn[now] = min(minn[rs[now]],minn[ls[now]]);
    sum[now] = sum[ls[now]]+calc(rs[now],mid+1,r,minn[ls[now]]);
}

ll query(int &now,int l,int r,int pos1,int pos2,ll w){
    if(!now) now = ++sz;
    if(pos1 <= l && pos2 >= r){
        ans += calc(now,l,r,w);
        return minn[now];
    }
    int mid = (l+r)>>1;
    if(pos1 <= mid) w = min(w,query(ls[now],l,mid,pos1,pos2,w));
    if(pos2 > mid) w = min(w,query(rs[now],mid+1,r,pos1,pos2,w));
    return w;
```

```
}

int main()
{
    int _; scanf("%d",&_);
    while(_--){
        rt = sz = 0;
        memset(ls,0,sizeof ls);
        memset(rs,0,sizeof rs);
        scanf("%d%d",&n,&q);
        rep(i,1,n){
            scanf("%lld",&a[i]);
            update(rt,1,n,i,a[i]);
        }
        rep(i,1,q){
            int l,r,p,c; scanf("%d%d%d%d",&l,&r,&p,&c);
            ans = 0, query(rt,1,n,l,r,inf);
            printf("%lld\n",ans);
            if(p != 0 || c != 0) update(rt,1,n,p,c);
        }
    }
    return 0;
}
```

4.9.3 线段树维护图连通

题意：n 个点，m 条边，每条边有一个区间 $[l_i, r_i]$，表示只有体重在这个范围内的人才能通过这条边，问有多少种不同的体重可以从起点到达终点。$(1 \le n, m \le 10^5, 1 \le l_i, r_i \le 10^9)$

思路：这应该是一类经典题目，即按照每条边的权重建线段树，这次是题目给出的上下界，下次可能是到达时间等。

然后每个节点维护该权重下有哪些边是连通的。因此每个节点均维护一个 $vector$，存储当前权重下哪些边连通。然后线段树分治的时候，将当前权重涉及到的边全部用并查集连起来。

每到达线段树的一个节点，就将节点的边连通，然后判断当前 1 是否和 n 可达，如果可达，则返回。否则继续二分区间，向下继续分治。没做过这类题的话，可能不太好理解，建议直接看代码，代码还是很好理解的。

这里还涉及到一个问题，就是每条边的权重是一个区间，如果我们直接按照闭区间往线段树中添加，是会出错的。具体出错原因可以手跑一下下面这个数据，大致来说就是权重若为闭合区间，在线段树分治的过程中，会丢失很多相邻区间之间的答案。

因此离散化时，需要将右端点 $+1$，插入一个区间时，需要将右端点离散化后的值再 -1。分治时，查询到一段可行区间后，对答案的贡献应该是 $b[r+1]-b[l]$，这样就不会忽略掉 $[b[r], b[r+1])$ 之间的一段了。而开区间的话，会忽略掉这一段。

```
#include <bits/stdc++.h>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl
    ;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2
    << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const db EPS = 1e-9;
const int N = 2*1e5+100;
```

```
using namespace std;

int n,m,tot,sz[N],fa[N];
struct Node{
    int u,v;
    ll l,r;
}e[N];
ll b[N],ans;
vector<int> t[2*N];

int findx(ll x) {return lower_bound(b+1,b+1+tot,x)-b;}
int find(int x) {return x==fa[x]?x:find(fa[x]);}
inline int get_id(int l,int r) {return (l+r)|(l!=r);}

void update(int l,int r,int ls,int rs,int id){
    int now = get_id(l,r);
    if(ls <= l && r <= rs){
        t[now].push_back(id);
        return;
    }
    int mid = (l+r)>>1;
    if(ls <= mid) update(l,mid,ls,rs,id);
    if(rs > mid) update(mid+1,r,ls,rs,id);
}

void dfs(int l,int r){
    int now = get_id(l,r);
    int len = t[now].size();
    vector<int> back; back.clear();
    rep(i,0,len-1){
        int u = find(e[t[now][i]].u), v = find(e[t[now][i]].v);
        if(u != v){
            if(sz[u] > sz[v]) swap(u,v);
            fa[u] = v; sz[v] += sz[u];
            back.push_back(u); back.push_back(v);
        }
    }
    if(find(1) == find(n)) ans += b[r+1]-b[l];
    else if(l != r){
        int mid = (l+r)>>1;
        dfs(l,mid); dfs(mid+1,r);
    }
    len = back.size();
    rep(i,0,len-1){
        fa[back[i]] = back[i], sz[back[i]] = 1;
    }
    back.clear();
}

int main()
{
    scanf("%d%d",&n,&m);
    rep(i,1,n) fa[i] = i, sz[i] = 1;
    rep(i,1,m){
        scanf("%d%d%lld%lld",&e[i].u,&e[i].v,&e[i].l,&e[i].r);
        e[i].r++;
        b[++tot] = e[i].l; b[++tot] = e[i].r;
    }
    sort(b+1,b+1+tot);
```

```
    tot = unique(b+1,b+1+tot)-b-1;
    rep(i,1,m) update(1,tot,findx(e[i].l),findx(e[i].r)-1,i);
    dfs(1,tot);
    printf("%lld\n",ans);
    return 0;
}
```

4.9.4 线段树维护树直径

题意：n 个点的一棵树，每个点初始为 0，支持两种操作，第一种操作 $C\ x$，表示将第 x 个点取反，即 1 变 0，0 变 1。第二种操作为 G，表示查询两个相距最远的 0 点距离。$(1 \le n \le 10^5, 1 \le m \le 5 * 10^5)$

思路：这道题的做法有括号序列、动态点分治、线段树维护直径。此处只介绍线段树维护直径的做法。

首先我们求一个 dfs 序，然后在 dfs 序上建线段树，对于线段树的每个区间，我们维护两个点表示在这个区间中相距最远的两个 0 点。区间合并时我们只需要取出这两个区间所维护的点，然后对这四个点两两求距离更新答案即可。

这样做的原因在于对于两个子树来说，每个子树都有一条属于该子树的直径，则两个子树合并后的新直径必定是从原来两个子树中的 4 个点中选取 2 个点作为答案。

大致思路就是这样，具体细节见代码。

```
#include <bits/stdc++.h>
#define rep(i,a,b) for(int i = a; i <= b; i++)
typedef long long ll;
const int N = 1e5+100;
using namespace std;

int n,m,dis[N],tot,head[N],f[N][25],dfn[N],rk[N],flag[N],T;
pair<int,int> sgt[4*N];
struct Edge{
    int to,next;
}e[2*N];

inline void add(int x,int y){
    e[++tot].to = y, e[tot].next = head[x], head[x] = tot;
}

void dfs(int x,int fa){
    dis[x] = dis[fa]+1; f[x][0] = fa; dfn[x] = ++tot; rk[tot] = x;
    for(int i = 1; (1<<i) <= dis[x]; i++)
        f[x][i] = f[f[x][i-1]][i-1];
    for(int i = head[x]; i; i = e[i].next){
        int y = e[i].to;
        if(y == fa) continue;
        dis[y] = dis[x]+1; dfs(y,x);
    }
}

inline int lca(int x,int y){
    if(dis[x] > dis[y]) swap(x,y);
    for(int i = T; i >= 0; i--)
        if(dis[f[y][i]] >= dis[x]) y = f[y][i];
    if(x == y) return x;
    for(int i = T; i >= 0; i--)
        if(f[x][i] != f[y][i]) x = f[x][i], y = f[y][i];
    return f[x][0];
```

```
}

inline int dist(int x,int y){
    if(x == 0 && y == 0) return -1;
    if(x == 0 || y == 0) return 0;
    return dis[x]+dis[y]-2*dis[lca(x,y)];
}

inline pair<int,int> pushUp(pair<int,int> x,pair<int,int> y){
    int a[5],res = 0,cnt = 0,tmp; pair<int,int> base = make_pair(0,0); a[0] = 0;
    if(x.first && flag[x.first]) a[++cnt] = x.first;
    if(x.second && flag[x.second]) a[++cnt] = x.second;
    if(y.first && flag[y.first]) a[++cnt] = y.first;
    if(y.second && flag[y.second]) a[++cnt] = y.second;
    rep(i,1,cnt-1)
        rep(j,i+1,cnt)
            if((tmp=dist(a[i],a[j])) > res)
                res = tmp, base = make_pair(a[i],a[j]);
    if(res == 0) base = make_pair(a[cnt],a[cnt]);
    return base;
}

inline void build(int now,int l,int r){
    if(l == r) sgt[now] = make_pair(rk[l],rk[l]), flag[rk[l]] = 1;
    else{
        int mid = (l+r)>>1;
        build(now<<1,l,mid); build(now<<1|1,mid+1,r);
        sgt[now] = pushUp(sgt[now<<1],sgt[now<<1|1]);
    }
}

inline void update(int now,int l,int r,int pos){
    if(l == r){
        flag[rk[l]] ^= 1;
        if(flag[rk[l]]) sgt[now] = make_pair(rk[l],rk[l]);
        else sgt[now] = make_pair(0,0);
        return;
    }
    int mid = (l+r)>>1;
    if(pos <= mid) update(now<<1,l,mid,pos);
    else update(now<<1|1,mid+1,r,pos);
    sgt[now] = pushUp(sgt[now<<1],sgt[now<<1|1]);
}

int main()
{
    scanf("%d",&n);
    tot = 1; T = (int)(log(n)/log(2))+1;
    rep(i,1,n-1){
        int a,b; scanf("%d%d",&a,&b);
        add(a,b); add(b,a);
    }
    tot = 0; dfs(1,0); build(1,1,n);
    scanf("%d",&m);
    while(m--){
        char op[10]; scanf("%s",op);
        if(op[0] == 'C'){
            int x; scanf("%d",&x);
            update(1,1,n,dfn[x]);
```

```
        }
        else printf("%d\n",dist(sgt[1].first,sgt[1].second));
    }
    return 0;
}
```

4.9.5 区间不同 gcd 个数

题意：给出 n 个数字，一共 q 次查询，每次询问一个 l、r，查询区间 $[l,r]$ 中有多少个不同的 gcd，其中一个子区间代表一个 gcd。$(1 \le n, q \le 10^5, 1 \le a_i \le 10^6)$

思路：区间查询不同 gcd 的个数，这类题像一类套路问题，主要要抓住 gcd 的几个性质。

1. 固定右端点，移动左端点，gcd 的值从 a_r 不断变化，要么不变，要么至少除 2（因为 gcd 最小值为 2）。因此固定右端点之后，只会存在至多 log 个不同的 gcd，我们对于相同的 gcd 仅保留最靠右的位置。
2. 因此对于每个右端点，我们记录一个 $vector$，存储对于这个右端点的所有不同的 gcd 值。我们可以根据 $vector[i-1]$ 来更新 $vector[i]$。

处理完上述操作之后，我们得到了 $n * logn$ 个 (l, r, gcd) 三元组，然后我们将所有查询按照右端点排序。

记录一个 pos，不断右移到查询的右端点位置，每次移动时将 $vector[pos]$ 内的信息存储到树状数组中，即对于每个三元组 (l, pos, gcd)，如果该 gcd 未出现过，则在树状数组的 l 位置 $+1$，并设置 $vis[gcd] = l$。如果该 gcd 出现过，则比较 l 是否比 $vis[gcd]$ 更大，如果更大，则修改 gcd 在树状数组中的位置。上述操作即不断维护 gcd 最靠右的出现位置。然后对于每个查询，直接在树状数组中区间查询即可。

总结：此题最关键的在于发现区间 gcd 不断除 2 的性质，然后将查询离线利用树状数组不断维护每个 gcd 最后出现的位置即可完成。

```
#include <bits/stdc++.h>
#define mem(a,b) memset(a,b,sizeof a);
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define per(i,a,b) for(int i = a; i >= b; i--)
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
typedef long long ll;
typedef double db;
const int N = 1e6+100;
const db EPS = 1e-9;
using namespace std;

void dbg() {cout << "\n";}
template<typename T, typename... A> void dbg(T a, A... x) {cout << a << ' '; dbg(x...)
    ;}
#define logs(x...) {cout << #x << " -> "; dbg(x);}

int n,q,a[N],vis[N];
ll ans[N],c[N];
struct Node{
    int l,r,id;
    bool operator < (Node xx) const {
        return r < xx.r;
    }
}Q[N];
vector<pair<int,int> > base[N];

inline int lowbit(int x) {return x&(~x+1);}
inline void update(int x,ll v) {for(;x<=n;x+=lowbit(x)) c[x] += v;}
inline ll ask(int x){
```

```
    ll tp = 0;
    while(x) tp += c[x], x -= lowbit(x);
    return tp;
}

int gcd(int a,int b){
    return b == 0 ? a:gcd(b,a%b);
}

int main()
{
    while(~scanf("%d%d",&n,&q)){
        rep(i,1,n) scanf("%d",&a[i]);
        rep(i,1,n) base[i].clear();
        rep(i,0,n) c[i] = 0;
        //更新每个点的vector
        rep(i,1,n){
            base[i].push_back(make_pair(i,a[i])); vis[a[i]] = 1;
            for(auto &v:base[i-1]){
                int tp = gcd(v.second,a[i]);
                if(!vis[tp]){
                    base[i].push_back(make_pair(v.first,tp));
                    vis[tp] = 1;
                }
            }
            for(auto &v:base[i])
                vis[v.second] = 0;
        }
        rep(i,1,q) scanf("%d%d",&Q[i].l,&Q[i].r), Q[i].id = i;
        sort(Q+1,Q+1+q);
        int pos = 0;
        rep(i,1,q){
            while(pos <= Q[i].r){
                for(auto &v:base[pos]){
                    int hp = v.first;
                    if(hp > vis[v.second]){
                        if(vis[v.second]) update(vis[v.second],-1);
                        vis[v.second] = hp;
                        update(vis[v.second],1);
                    }
                }
                pos++;
            }
            ans[Q[i].id] = ask(Q[i].r)-ask(Q[i].l-1);
        }
        rep(i,1,q) printf("%lld\n",ans[i]);
        memset(vis,0,sizeof vis);
    }
    return 0;
}
```

4.10 扫描线

4.10.1 面积并

只要矩形一条线一条线扫描的时候，右边的线没有被读进来，则右边的线一直在扫描面积

举个简单的例子，本题中的扫描线只是一个消除矩阵面积重复计算部分的一个方法

```
#include <iostream>
#include <cstdio>
#include <algorithm>
using namespace std;
const int SIZE = 300+10;
struct Line{
    double x,y1,y2;
    int flag;
}line[SIZE];

bool cmp(Line a,Line b)
{
    return a.x<b.x;
}
struct tree{
    int l,r;
    double ml,mr;
    int s;   //s始终 >= 0
    double len;
}t[SIZE*4];
int n;
double y[SIZE];
void build(int p,int l,int r)
{
    t[p].l = l; t[p].r = r; t[p].ml = y[l]; t[p].mr = y[r];
    t[p].s = 0;
    t[p].len = 0;
    if(t[p].l+1 == t[p].r) return;
    int mid = (l+r)>>1;
    build(p*2,l,mid);
    build(p*2+1,mid,r);
}

void callen(int p)
{
    if(t[p].s > 0)
        t[p].len = t[p].mr-t[p].ml;
    else if(t[p].l == (t[p].r-1)) //所以s == 0的点，最终长度会被赋成0
        t[p].len = 0;
    else
        t[p].len = t[p*2].len+t[p*2+1].len;
}

void change(int p,Line tmp)
{
    if(t[p].ml == tmp.y1 && t[p].mr == tmp.y2)
    {
        t[p].s += tmp.flag;
        callen(p);
        return;
    }
    if(tmp.y2 <= t[p*2].mr) change(p*2,tmp);
    else if(tmp.y1 >= t[p*2+1].ml) change(p*2+1,tmp);
    else
    {
        Line tp = tmp;
        tp.y2 = t[p*2].mr;
```

```
        change(p*2,tp);
        tp = tmp;
        tp.y1 = t[p*2+1].ml;
        change(p*2+1,tp);
    }
    callen(p);
}

int main()
{
    int cnt = 1;
    while(scanf("%d",&n) && n!=0)
    {
        int num = 1;
        double x1,x2,y1,y2;
        for(int i = 0;i < n;i++)
        {
            scanf("%lf%lf%lf%lf",&x1,&y1,&x2,&y2);
            line[num].x = x1; line[num].y1 = y1; line[num].y2 = y2;
            line[num].flag = 1;
            //保存的是线，flag == 1 表示该线在左边
            y[num++] = y1; //将所有的y都读入数组中，进行离散化
            line[num].x = x2; line[num].y1 = y1; line[num].y2 = y2;
            line[num].flag = -1;
            //flag == -1 表示该线在右边
            y[num++] = y2;
        }
        sort(line+1,line+num,cmp); //按照横坐标进行排序
        //对纵坐标进行离散化
        sort(y+1,y+num);
        int cm = unique(y+1,y+num)-y-1;
        //在y轴上建立线段树
        build(1,1,cm);
        change(1,line[1]);
        double ans = 0;
        for(int i = 2;i < num;i++)
        {
            ans += t[1].len*(line[i].x-line[i-1].x);
            change(1,line[i]);
        }
        printf("Test case #%d\n",cnt++);
        printf("Total explored area: %.2f\n\n",ans);
    }
    return 0;
}
```

4.10.2 面积交

只要矩形一条线一条线扫描的时候，右边的线没有被读进来，则右边的线一直在扫描面积

举个简单的例子，本题中的扫描线只是一个消除矩阵面积重复计算部分的一个方法

```
#include <iostream>
#include <cstdio>
#include <algorithm>
using namespace std;
const int SIZE = 300+10;
```

```
struct Line{
    double x,y1,y2;
    int flag;
}line[SIZE];

bool cmp(Line a,Line b)
{
    return a.x<b.x;
}
struct tree{
    int l,r;
    double ml,mr;
    int s;   //s始终 >= 0
    double len;
}t[SIZE*4];
int n;
double y[SIZE];
void build(int p,int l,int r)
{
    t[p].l = l; t[p].r = r; t[p].ml = y[l]; t[p].mr = y[r];
    t[p].s = 0;
    t[p].len = 0;
    if(t[p].l+1 == t[p].r) return;
    int mid = (l+r)>>1;
    build(p*2,l,mid);
    build(p*2+1,mid,r);
}

void callen(int p)
{
    if(t[p].s > 0)
        t[p].len = t[p].mr-t[p].ml;
    else if(t[p].l == (t[p].r-1)) //所以s == 0的点，最终长度会被赋成0
        t[p].len = 0;
    else
        t[p].len = t[p*2].len+t[p*2+1].len;
}

void change(int p,Line tmp)
{
    if(t[p].ml == tmp.y1 && t[p].mr == tmp.y2)
    {
        t[p].s += tmp.flag;
        callen(p);
        return;
    }
    if(tmp.y2 <= t[p*2].mr) change(p*2,tmp);
    else if(tmp.y1 >= t[p*2+1].ml) change(p*2+1,tmp);
    else
    {
        Line tp = tmp;
        tp.y2 = t[p*2].mr;
        change(p*2,tp);
        tp = tmp;
        tp.y1 = t[p*2+1].ml;
        change(p*2+1,tp);
    }
    callen(p);
}
```

```

int main()
{
    int cnt = 1;
    while(scanf("%d",&n) && n!=0)
    {
        int num = 1;
        double x1,x2,y1,y2;
        for(int i = 0;i < n;i++)
        {
            scanf("%lf%lf%lf%lf",&x1,&y1,&x2,&y2);
            line[num].x = x1; line[num].y1 = y1; line[num].y2 = y2;
            line[num].flag = 1;
            //保存的是线，flag == 1 表示该线在左边
            y[num++] = y1; //将所有的y都读入数组中，进行离散化
            line[num].x = x2; line[num].y1 = y1; line[num].y2 = y2;
            line[num].flag = -1;
            //flag == -1 表示该线在右边
            y[num++] = y2;
        }
        sort(line+1,line+num,cmp); //按照横坐标进行排序
        //对纵坐标进行离散化
        sort(y+1,y+num);
        int cm = unique(y+1,y+num)-y-1;
        //在y轴上建立线段树
        build(1,1,cm);
        change(1,line[1]);
        double ans = 0;
        for(int i = 2;i < num;i++)
        {
            ans += t[1].len*(line[i].x-line[i-1].x);
            change(1,line[i]);
        }
        printf("Test case #%d\n",cnt++);
        printf("Total explored area: %.2f\n\n",ans);
    }
    return 0;
}
```

4.10.3 周长并

题意：
给一堆矩阵，求出所有矩阵拼起来，求出矩阵并起来的总周长。

思路：
其实与面积并差不多，就是求 ans 的时候，ans += abs(last-t[1].len)，每次插入一根线段，对答案的贡献值为使得 t[1].len 增加或减少的长度。

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#include <cmath>
#define rep(i,a,b) for(int i = a; i <= b; i++)
using namespace std;
const int N = 10000+100;

```

```
int n;
struct Line{
    int x,y1,y2;
    int flag;
}s1[N],s2[N];

bool cmp(Line a,Line b)
{
    return a.x < b.x;
}

struct Tree{
    int l,r,s;
    int ml,mr,len;
}t[N*4];

int y[4][N],ans;
int num1,num2;

void build(int p,int l,int r,int idx)
{
    t[p].l = l, t[p].r = r, t[p].s = 0, t[p].ml = y[idx][l], t[p].mr = y[idx][r];
    if(l == (r-1)) return;
    int mid = (l+r)>>1;
    build(p*2,l,mid,idx);
    build(p*2+1,mid,r,idx);
}

void calc(int p)
{
    if(t[p].s >= 1) t[p].len = t[p].mr-t[p].ml;
    else if(t[p].l == (t[p].r-1)) t[p].len = 0;
    else{
        t[p].len = t[p*2].len+t[p*2+1].len;
    }
}

void update(int p, Line tp)
{
    if(tp.y1 <= t[p].ml && t[p].mr <= tp.y2)
    {
        t[p].s += tp.flag;
        calc(p);
        return;
    }
    if(t[p*2].mr >= tp.y2) update(p*2,tp);
    else if(t[p*2+1].ml <= tp.y1) update(p*2+1,tp);
    else{
        update(p*2,tp);
        update(p*2+1,tp);
    }
    calc(p);
}

int main()
{
    while(~scanf("%d",&n))
    {
        num1 = 0, num2 = 0, ans = 0;
```

```
        rep(i,1,n)
        {
            int x1,y1,x2,y2;
            scanf("%d%d%d%d",&x1,&y1,&x2,&y2);
            s1[++num1].x = x1, s1[num1].y1 = y1, s1[num1].y2 = y2, y[1][num1] = y1, s1[num1].flag = 1;
            s1[++num1].x = x2, s1[num1].y1 = y1, s1[num1].y2 = y2, y[1][num1] = y2, s1[num1].flag = -1;

            s2[++num2].x = y1, s2[num2].y1 = x1, s2[num2].y2 = x2, y[2][num2] = x1, s2[num2].flag = 1;
            s2[++num2].x = y2, s2[num2].y1 = x1, s2[num2].y2 = x2, y[2][num2] = x2, s2[num2].flag = -1;
        }
        sort(s1+1,s1+1+num1,cmp);
        sort(s2+1,s2+1+num2,cmp);
        sort(y[1]+1,y[1]+1+num1);
        sort(y[2]+1,y[2]+1+num2);

        int scr1 = unique(y[1]+1,y[1]+1+num1)-y[1]-1;
        int scr2 = unique(y[2]+1,y[2]+1+num2)-y[2]-1;

        build(1,1,scr1,1);
        rep(i,1,num1)
        {
            int last = t[1].len;
            update(1,s1[i]);
            ans += abs(last-t[1].len);
        }

        build(1,1,scr2,2);
        rep(i,1,num2)
        {
            int last = t[1].len;
            update(1,s2[i]);
            ans += abs(last-t[1].len);
        }

        printf("%d\n",ans);
    }
    return 0;
}
```

4.10.4 包星星问题

题意：
给出一大堆星星的坐标，给出每个星星的亮度。然后给出一个矩形，要求用这个矩形包住的星星的亮度最大。注意：如果星星在矩形边界上，则不计算这个星星的亮度。

思路：
我们来思考一下，一个矩形的位置是不是由这个矩形右上角这个点所决定的，所以我们可以把考虑矩形的位置改为考虑右上角这个点所在的位置。

然后我们可以发现，对于一颗星星，(x,y)，只要右上角这个点在（x+0.1，y+0.1）~（x+w-0.1，y+h-0.1）这个范围内，即可包住这颗星星。此处取 0.1 的原因是星星不能在矩形边界上。

因此一个星星就可以确定一个矩形，那么本题就变成了给出一大堆矩形，每个矩形都有一个权值，问其中哪一个区域矩形值之和最大。

因此我们可以将每个矩形的左右边界抽离出来，然后就变成了区间覆盖问题。

询问在线段树维护下的这根扫描线上亮度最大的值是多少，所以线段树上只需要维护一个最大值，再加上一个 lazy 标记，然后边插入边，边更新 ans，就可以通过此题。

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#define rep(i,a,b) for(int i = a; i <= b; i++)
using namespace std;
const int N = 100000;

struct Line{
    double x,y1,y2;
    int flag;
}line[N];

bool cmp(Line a,Line b)
{
    return a.x < b.x;
}

int n,w,h,num,ans;
double y[N];

struct Tree{
    int l,r,lazy;
    double ml,mr;
    int maxn;
}t[N*4];

void build(int p,int l,int r)
{
    t[p].l = l, t[p].r = r, t[p].ml = y[l], t[p].mr = y[r], t[p].maxn = 0, t[p].lazy =
    0;
    if(l == r) return;
    int mid = (l+r)>>1;
    build(p*2,l,mid);
    build(p*2+1,mid+1,r);
}

void pushup(int p)
{
    if(t[p].lazy != 0)
    {
        t[p*2].lazy += t[p].lazy;
        t[p*2+1].lazy += t[p].lazy;
        t[p*2].maxn += t[p].lazy;
        t[p*2+1].maxn += t[p].lazy;
        t[p].lazy = 0;
    }
}

void change(int p, Line a)
{
//  cout << a.y1 << " " << a.y2 << endl;
    if(a.y1 <= t[p].ml && t[p].mr <= a.y2)
```

```
    {
    //  t[p].s += a.flag;
        t[p].lazy += a.flag;
        t[p].maxn += a.flag;
        return;
    }
    pushup(p);
    if(t[p*2].mr >= a.y2) change(p*2,a);
    else if(t[p*2+1].ml <= a.y1) change(p*2+1,a);
    else{
        change(p*2,a);
        change(p*2+1,a);
    }
    t[p].maxn = max(t[p*2].maxn,t[p*2+1].maxn);
}

int main()
{
    while(~scanf("%d%d%d",&n,&w,&h))
    {
        ans = 0, num = 0;
        rep(i,1,n)
        {
            double x1,y1,z1;
            scanf("%lf%lf%lf",&x1,&y1,&z1);
            line[++num].x = x1+0.1, line[num].y1 = y1+0.1, line[num].y2 = y1+h-0.1, y[num] = y1+0.1, line[num].flag = z1;
            line[++num].x = x1+w-0.1, line[num].y1 = y1+0.1, line[num].y2 = y1+h-0.1, y[num] = y1+h-0.1, line[num].flag = -z1;
        }
        sort(line+1,line+1+num,cmp);
        sort(y+1,y+1+num);
        int scr = unique(y+1,y+1+num)-y-1;
        build(1,1,scr);

        rep(i,1,num)
        {
            change(1,line[i]);
            ans = max(ans,t[1].maxn);
        }
        printf("%d\n",ans);
    }
    return 0;
}
```

4.10.5 覆盖奇数次的面积

题意：
给出 n 个矩形，求被覆盖区域为奇数次的总面积。

思路：
扫描线有很多种写法，可以打 lazy 更新到底，也可以不打 lazy，只是单纯对目标边进行更新，然后再区间合并上去。

本题问的是被覆盖区域为奇数次的总面积。因此线段树每个节点记录被覆盖的次数，被覆盖奇数次的长度，被覆盖偶数次的长度。每次加入一条边，只对被覆盖的那个最大的区间，覆盖次数 +1，对于该区间下面的区间不再进行更新，之后也不会更新。

因此假如第一次加入的线段是 (1,4)，第二次加入的线段是 (1,2)，因此第一次 (1,4) 区间覆盖次数变为 1，第二次

(1,2) 区间覆盖次数变为 1。然后区间合并的时候，len1 表示覆盖奇数次的长度，len2 表示覆盖偶数次的长度，(1,2) 区间 len1 = 1，len2 = 0. (1,4) 区间覆盖次数为奇数，因此 len1 = (1,2) 与 (3,4) 区间被覆盖偶数次的长度，因为偶 + 奇 = 奇，而 len2 = (1,2) 与 (3,4) 区间被覆盖奇数次的长度。因此可以发现，虽然在加入直线的时候，没有将更新次数一次性更新到底，但是在区间合并的时候，会将之前覆盖的长度一并算入。

由于每次插入直线的时候，最后都会合并到整根扫描线上，因此询问整根扫描线的奇偶长度得到的答案是正确的。但是如果询问某一个区间被覆盖奇数次的长度则会得到错误答案，因为这个区间被覆盖的次数还取决于这个区间之上的区间。

```cpp
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#define rep(i,a,b) for(int i = a; i <= b; i++)
using namespace std;
const int N = 2*1e5+1000;
typedef long long ll;

struct Line{
    ll x,y1,y2;
    int flag;
}line[N];
int n, num;
ll y[N], ans;
struct Tree{
    int l,r,s;
    ll ml,mr,len1,len2; //记录覆盖偶数次的长度 和 覆盖奇数次的长度
}t[N*4];

bool cmp(Line a,Line b)
{
    return a.x < b.x;
}

void build(int p,int l,int r)
{
    t[p].l = l, t[p].r = r, t[p].ml = y[l], t[p].mr = y[r], t[p].s = 0, t[p].len1 = y[r]-y[l], t[p].len2 = 0;
//  cout << t[p].ml << " " << t[p].mr << endl;
    if(l == (r-1)) return;
    int mid = (l+r)>>1;
    build(p*2,l,mid);
    build(p*2+1,mid,r);
}

void calc(int p)
{
    //len1: 覆盖偶数次, len2: 覆盖奇数次
    if(t[p].s % 2){ //覆盖奇数次
        if(t[p].r == (t[p].l+1)) t[p].len2 = t[p].mr-t[p].ml, t[p].len1 = 0;
        else{
            t[p].len1 = t[p*2].len2+t[p*2+1].len2;
            t[p].len2 = t[p*2].len1+t[p*2+1].len1;
            // t[p].len2 = t[p].mr-t[p].ml-t[p*2].len2-t[p*2+1].len2;
        }
    }
    else if(t[p].s % 2 == 0){   //覆盖偶数次
        if(t[p].r == (t[p].l+1)) t[p].len1 = t[p].mr-t[p].ml, t[p].len2 = 0;
        else{
```

```
            // t[p].len1 = t[p].mr-t[p].ml-t[p*2].len2-t[p*2+1].len2;
            t[p].len1 = t[p*2].len1+t[p*2+1].len1;
            t[p].len2 = t[p*2].len2+t[p*2+1].len2;
        }
    }
}

void change(int p, Line tp)
{
    if(tp.y1 <= t[p].ml && tp.y2 >= t[p].mr)
    {
        t[p].s += tp.flag;
        calc(p);
        return;
    }
    if(t[p*2].mr >= tp.y2) change(p*2,tp);
    else if(t[p*2+1].ml <= tp.y1) change(p*2+1,tp);
    else{
        change(p*2,tp);
        change(p*2+1,tp);
    }
    calc(p);
}

int main()
{
    ans = 0, num = 0;
    scanf("%d",&n);
    rep(i,1,n){
        ll x1,x2,y1,y2;
        scanf("%lld%lld%lld%lld",&x1,&y1,&x2,&y2);
        line[++num].x = x1, line[num].y1 = y1, line[num].y2 = y2, line[num].flag = 1;
        y[num] = y1;
        line[++num].x = x2, line[num].y1 = y1, line[num].y2 = y2, line[num].flag = -1;

        y[num] = y2;
    }
    sort(line+1,line+1+num,cmp);
    sort(y+1,y+1+num);
    int sc = unique(y+1,y+num+1)-y-1;
    build(1,1,sc);
    change(1,line[1]);
    rep(i,2,num)
    {
        // printf("%f\n",t[1].len2);
        ans += t[1].len2*(line[i].x-line[i-1].x);
        change(1,line[i]);
    }
    cout << ans << endl;
    return 0;
}
```

4.11 主席树

4.11.1 静态主席树

新建多个权值线段树副本，记录只考虑 1-i 个数时，每个数出现在各个区间的个数是多少，类似于建多棵权值线段树然后第 i 棵线段树，参考第 i-1 棵线段树，优化空间空间为 4*n*log(n)

```cpp
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#define rep(i,a,b) for(int i = a; i <= b; i++)
using namespace std;
const int N = 1e5+100;

struct Tree{
    int lc,rc; //左右节点 t数组 编号
    int l,r; //节点的左右端点
    int sum;
}t[N*20];
int n,m,a[N],num,b[N],root[N],tot;

int build(int l,int r)
{
    int p = ++tot; // 新建一个节点，编号为p，代表当前区间[l,r]
    t[p].l = l, t[p].r = r, t[p].sum = 0;
    if(l == r) return p;
    int mid = (l+r)>>1;
    t[p].lc = build(l,mid);
    t[p].rc = build(mid+1,r);
    return p;
}

int insert(int now,int pos,int k)
{
    int p = ++tot;
    t[p] = t[now]; //建立副本
    if(t[p].l == t[p].r){
        t[p].sum += k; //在副本上修改
        return p;
    }
    int mid = (t[p].l+t[p].r)>>1;
    if(pos <= mid) t[p].lc = insert(t[p].lc,pos,k); //保留右儿子部分，把左儿子更新
    else t[p].rc = insert(t[p].rc,pos,k);
    t[p].sum = t[t[p].lc].sum + t[t[p].rc].sum;
    return p;
}

int ask(int lp,int rp,int k) //lp和rp所代表的区间是相同的，他们只不过是在不同状态下的副本
{
    if(t[lp].l == t[lp].r) return t[lp].l; //找到答案
    int cnt = t[t[rp].lc].sum-t[t[lp].lc].sum; // 值在[l,mid]中的数有多少个
    if(cnt >= k) return ask(t[lp].lc,t[rp].lc,k);
    else return ask(t[lp].rc,t[rp].rc,k-cnt);
}

int main()
{
    num = tot = 0;
    scanf("%d%d",&n,&m);
    rep(i,1,n){
        scanf("%d",&a[i]);
        b[++num] = a[i];
    }
    sort(b+1,b+1+num); //离散化
    num = unique(b+1,b+1+num)-b-1;
```

```
    root[0] = build(1,num); //root[0]这颗树是一棵空树，关于离散化后的值域建树
    rep(i,1,n)
    {
        int x = lower_bound(b+1,b+1+num,a[i])-b; //离散化后的值
        root[i] = insert(root[i-1],x,1); //值为x的数增加1个
    }
    rep(i,1,m)
    {
        int x,y,z;
        scanf("%d%d%d",&x,&y,&z);
        int ans = ask(root[x-1],root[y],z);
        printf("%d\n",b[ans]); //从离散化后的值变回原值
    }
    return 0;
    //root[i]:表示只考虑1-i这些数时候建树的情况，这颗树树根的编号
}
```

4.11.2 区间中不同数的个数

题意：
长度为 n 的序列，q 次询问，每次给出 l、r，返回序列 $[l,r]$ 中不同数的个数。$(1 \le n \le 3*10^4, 1 \le q \le 2*10^5)$

思路：
与之前主席树的权值线段树思路不同，此题的思路是建立 n 颗线段树，第 i 颗线段树存储区间 $[1,i]$ 的信息。其中每个节点维护 sum，表示节点对应区间中数的个数，因此每棵线段树中只保留每个数最后出现的位置。

举个例子，序列为 5 5 5 5 5，则第 1 颗线段树中只有第一个位置 sum 为 1，然后第二颗线段树从第一颗线段树继承过来，由于 5 这个数字之前出现过，因此在第二颗线段树中令第一个位置的 sum 为 0，令第二个位置的 sum 为 1，来保存每个数字最后出现的位置。

因此查询区间 $[l,r]$ 时，就在第 r 颗线段树中查询区间 $[l,n]$ 中数的个数即可。

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl
    ;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2
    << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const int N = 3*1e4+100;
const int M = 1e6+100;
const db EPS = 1e-9;
using namespace std;

int n,a[N],q;
int ls[40*N],rs[40*N],sum[40*N],root[N],sz;
int pos[M]; //记录每个数的位置

void build(int &now,int l,int r){
    if(!now) now = ++sz;
    if(l == r){
```

```
        sum[now] = 0; return;
    }
    build(ls[now],l,(l+r)>>1);
    build(rs[now],((l+r)>>1)+1,r);
    sum[now] = sum[ls[now]]+sum[rs[now]];
}

void update(int x,int &now,int l,int r,int p1,int ct){
    if(!now) now = ++sz;
    if(l == r){
        sum[now] = sum[x]+ct; return;
    }
    int mid = (l+r)>>1;
    if(p1 <= mid){
        if(!rs[now]) rs[now] = rs[x]; //继承前一个备份的节点
        if(ls[now] <= now) ls[now] = 0; //因为现在要修改ls[now], 因此要重新开空间
        //如果ls[now] <= now, 表示这个节点是之前继承的, 因此要赋为0, 重新开空间
        update(ls[x],ls[now],l,mid,p1,ct);
    }
    else{
        if(!ls[now]) ls[now] = ls[x];
        if(rs[now] <= now) rs[now] = 0;
        update(rs[x],rs[now],mid+1,r,p1,ct);
    }
    sum[now] = sum[ls[now]]+sum[rs[now]];
}

int ask(int &now,int l,int r,int p1){
    if(!now) now = ++sz;
    if(l >= p1) return sum[now];
    else if(r < p1) return 0;
    int mid = (l+r)>>1, ans = 0;
    if(mid >= p1) ans += ask(ls[now],l,mid,p1)+sum[rs[now]];
    else ans += ask(rs[now],mid+1,r,p1);
    return ans;
}

int main()
{
    scanf("%d",&n);
    rep(i,1,n) scanf("%d",&a[i]);
    build(root[0],1,n);
    rep(i,1,n){
        if(pos[a[i]]){
            update(root[i-1],root[i],1,n,pos[a[i]],-1);
            update(root[i-1],root[i],1,n,i,1);
            pos[a[i]] = i;
        }
        else update(root[i-1],root[i],1,n,i,1), pos[a[i]] = i;
    }
    scanf("%d",&q);
    rep(i,1,q){
        int xx,yy; scanf("%d%d",&xx,&yy);
        printf("%d\n",ask(root[yy],1,n,xx));
    }
    return 0;
}
```

4.11.3 单点修改主席树

题意：

给定一个区间，求这个区间第 k 小的数，支持单点修改。

思路：

动态主席树裸题。

我们先来回顾一下静态主席树的做法，对于数组中每一个位置都维护一棵权值线段树，该权值线段树保存的是区间 [1,x] 的信息。因此我想要求区间 [l,r] 之间第 k 大的时候，只需要将 root[r]-root[l-1] 就是维护区间 [l,r] 信息的权值线段树，因此就可以快速直接求出这个区间中第 k 大的元素是多少。

现在我们来看看单点修改的操作。

如果我现在要将 a[pos] 修改为 x，那么最暴力的做法就是对于 root[pos] ~ root[n] 中的每一颗权值线段树都进行修改，即将 a[pos] 这个点的值减 1，将 x 这个点的值 +1。

最暴力的做法显然是无法通过此题的，因此我们可以想到有没有一种 logn 的方法，可以只修改 logn 个节点，就可以对于每一个线段树记录修改信息，于是我们想到了树状数组。

我们来回忆一下树状数组，每一个节点记录区间 [x-lowbit(x)+1, x] 的所有信息，因此当需要求 [1,x] 内维护的信息的时候，只需要从节点 x 出发，每次进行 x-=lowbit(x) 的操作，即可求出 [1,x] 内维护的所有信息。

每次对 x 节点进行修改的时候，只需要不断进行 x+=lowbit(x) 的操作，就可以访问到所有存储 x 节点信息的节点，因此实现了 logn 的查询。

因此本题也维护一个树状数组。

节点 x 维护的是区间 [x-lowbit(x)+1, x] 的权值线段树，因此当我需要访问 [l, r] 区间信息的时候，只需要将 root[r]-root[l-1]+getsum(r)-getsum(l-1) 这里面维护的便是区间 [l, r] 的权值线段树。

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const int N = 6*1e4+100;
const int M = 32*N+100;
const db EPS = 1e-9;
using namespace std;

int rt[N],root[N],ls[M],rs[M],sz,n,m,tot,h1,h2,L[N],R[N],sum[M],a[N],b[N];
struct Node{
    int op,xx,yy,k; //保存修改信息
}t[N];

int find(ll x) {return (lower_bound(b+1,b+1+tot,x)-b);}
int lowbit(int x) {return x&(-x);}

void update(int pre,int &now,int l,int r,int pos,int c){
```

```
    if(!now) now = ++sz;
    sum[now] = sum[pre]+c;
    if(l == r) return;
    int mid = (l+r)>>1;
    if(pos <= mid){rs[now] = rs[pre]; update(ls[pre],ls[now],l,mid,pos,c);}
    else{ls[now] = ls[pre]; update(rs[pre],rs[now],mid+1,r,pos,c);}
}

int query(int pre,int now,int l,int r,int k){
    int sum1 = 0, sum2 = 0;
    if(l == r) return l;
    rep(i,1,h1) sum1 += sum[ls[L[i]]]; //左边树状数组节点累加值
    rep(i,1,h2) sum2 += sum[ls[R[i]]]; //右边树状数组节点累加值
    int temp = sum[ls[now]]-sum[ls[pre]]+sum2-sum1;
    int mid = (l+r)>>1;
    if(temp >= k){
        rep(i,1,h1) L[i] = ls[L[i]];
        rep(i,1,h2) R[i] = ls[R[i]];
        return query(ls[pre],ls[now],l,mid,k);
    }
    else{
        rep(i,1,h1) L[i] = rs[L[i]];
        rep(i,1,h2) R[i] = rs[R[i]];
        return query(rs[pre],rs[now],mid+1,r,k-temp);
    }
}

void init(){
    scanf("%d%d",&n,&m);
    rep(i,1,n){
        scanf("%d",&a[i]); b[++tot] = a[i];
    }
    rep(i,1,m){
        char op[10]; scanf("%s",op);
        if(op[0] == 'Q'){
            t[i].op = 1; scanf("%d%d%d",&t[i].xx,&t[i].yy,&t[i].k);
        }
        else{
            t[i].op = 2; scanf("%d%d",&t[i].xx,&t[i].yy);
            b[++tot] = t[i].yy;
        }
    }
    sort(b+1,b+1+tot);
    tot = unique(b+1,b+1+tot)-b-1;
    rep(i,1,n)
        update(rt[i-1],rt[i],1,tot,find(a[i]),1);
}

void solve(){
    rep(i,1,m){
        if(t[i].op == 1){ //查询
            h1 = h2 = 0;
            for(int j = t[i].xx-1; j; j -= lowbit(j)) L[++h1] = root[j]; //记录树状数组要计算的节点
            for(int j = t[i].yy; j; j -= lowbit(j)) R[++h2] = root[j];
            int pos = query(rt[t[i].xx-1],rt[t[i].yy],1,tot,t[i].k);
            printf("%d\n",b[pos]);
        }
        else{ //修改
```

```
            int pos1 = find(a[t[i].xx]), pos2 = find(t[i].yy);
            for(int j = t[i].xx; j <= n; j += lowbit(j)) update(root[j],root[j],1,tot,pos1,-1);
            for(int j = t[i].xx; j <= n; j += lowbit(j)) update(root[j],root[j],1,tot,pos2,1);
            a[t[i].xx] = t[i].yy;
        }
    }
}

int main()
{
    int _; scanf("%d",&_);
    while(_--){
        init();
        solve();
        rep(i,0,sz) ls[i] = rs[i] = sum[i] = 0;
        rep(i,0,n) root[i] = rt[i] = 0;
        sz = 0, tot = 0;
    }
    return 0;
}
```

4.12 李超树

4.12.1 李超树模板题

题意：
两种操作。第一种操作查询第 T 天的最大收益，第二种操作给出一个设计方案，该方案第一天收益为 S，之后每一天比上一天多收益 P，注意设计方案收益不可叠加。$(1 \le T \le 5 * 10^4)$

思路：
这是一道模板题，每次动态插入一条直线，单点查询最大值即可。

```
#include <bits/stdc++.h>
#define rep(i,a,b) for(int i = a; i <= b; i++)
typedef long long ll;
typedef double db;
const int N = 5*1e4+100;
const int M = 1e5+100;
const db EPS = 1e-12;
using namespace std;

struct line{
    db k,b;
    int l,r,flag;
    line() {}
    line(db a1,db a2,int l1,int r1,int f1) : k(a1),b(a2),l(l1),r(r1),flag(f1) {}
}sgt[2*N];
int n;

inline int get_id(int l,int r) {return (l+r)|(l!=r);}
//计算某条线段在某一个横坐标的纵坐标值
inline db calc(line a,int pos) {return a.k*pos+a.b;}

void modify(int l,int r,line k){
    int now = get_id(l,r);
    if(k.l <= l && k.r >= r){
```

```
        if(!sgt[now].flag) sgt[now] = k, sgt[now].flag = 1; //原区间没有优势线段
        else if(calc(k,l)-calc(sgt[now],l) > EPS && calc(k,r)-calc(sgt[now],r) > EPS) sgt[now] = k; //原区间优势线段被覆盖
        else if(calc(k,l)-calc(sgt[now],l) > EPS || calc(k,r)-calc(sgt[now],r) > EPS){ //有一端比原线段大
            int mid = (l+r)>>1;
            //中点坐标处，新线段比原来优势线段更优
            if(calc(k,mid)-calc(sgt[now],mid) > EPS) swap(k,sgt[now]);
            if(calc(k,l)-calc(sgt[now],l) > EPS) modify(l,mid,k);
            else modify(mid+1,r,k);
        }
    }
    else{
        int mid = (l+r)>>1;
        if(k.l <= mid) modify(l,mid,k); //涉及了左区间
        if(mid < k.r) modify(mid+1,r,k); //涉及了右区间
    }
}

db query(int l,int r,int x){
    int now = get_id(l,r);
    if(l == r) return calc(sgt[now],x);
    else{
        int mid = (l+r)>>1;
        db ans = calc(sgt[now],x);
        if(x <= mid) return max(ans,query(l,mid,x));
        else return max(ans,query(mid+1,r,x));
    }
}

int main()
{
    scanf("%d",&n);
    rep(i,1,n){
        char op[10]; scanf("%s",op);
        if(op[0] == 'P'){
            db k,b; scanf("%lf%lf",&b,&k);
            b = b-k;
            line tmp(k,b,1,5*1e4,1);
            modify(1,5*1e4,tmp);
        }
        else{
            int x; scanf("%d",&x);
            printf("%lld\n",(ll)floor(query(1,50000,x)/100));
        }
    }
    return 0;
}
```

4.12.2 离散化 + 维护最大最小直线

题意：
现有 n 个机器人，第 i 个机器人初始在 a_i 位置。一共有两个操作。第一种操作为 T_i command k_i x_i，表示 T_i 时间，第 k_i 个机器人以 x_i 的速度开始移动，x_i 可以为负值。第二种操作为 T_i query，表示查询 T_i 时，离原点最远的机器人的坐标。$(0 \le T_i \le 10^9)$

思路：
对于每一个机器人来说，其速度变化都是一个分段函数，只需要先离线，将时间离散化，处理出每一个机器人各速度下

的分段直线。然后在第一种操作时，动态插入一条直线，第二种操作时，单点查询最大值与最小值即可完成此题。

本题主要特点就是对时间离散化，然后在离散化后的时间上插入直线。每一个节点需要同时维护最大值和最小值直线。

```
#include <bits/stdc++.h>
#define rep(i,a,b) for(int i = a; i <= b; i++)
typedef long long ll;
const int N = 1e5+100;
const int M = 6*1e5+100;
using namespace std;

int n,m,tot,rp[N];
ll a[N],b[M];
struct Query{
    ll t,x; //x是速度
    int k,op; //k是第k个机器人
}q[M];
struct NP{
    ll t,x,pos,t2;
    NP() {}
    NP(ll _a,ll _b,ll _c,ll _d) : t(_a),x(_b),pos(_c),t2(_d) {}
};
vector<NP> hp[N];
struct line{
    ll k,b;
    int l,r;
    line() {}
    line(ll _a, ll _b, int _c, int _d) : k(_a), b(_b), l(_c), r(_d) {}
};
struct Node{
    line minn,maxn;
    int flag;
    Node() {flag = 0;}
}sgt[2*M];

inline int get_id(int l,int r) {return (l+r)|(l!=r);}
//计算某线段在某一个横坐标的纵坐标值
inline ll calc(line a,ll pos) {return a.k*b[pos]+a.b;}
//离散化
int find(ll x) {return lower_bound(b+1,b+1+tot,x)-b;}

void modify(int l,int r,line k){
    int now = get_id(l,r);
    if(k.l <= l && k.r >= r){
        if(!sgt[now].flag) sgt[now].minn = sgt[now].maxn = k, sgt[now].flag = 1;
        else{
            line tp = k;
            if(calc(k,l) >= calc(sgt[now].maxn,l) && calc(k,r) >= calc(sgt[now].maxn,r)) sgt[now].maxn = k;
            else if(calc(k,l) > calc(sgt[now].maxn,l) || calc(k,r) > calc(sgt[now].maxn,r)){
                int mid = (l+r)>>1;
                //让非优势线段向下更新
                if(calc(k,mid) > calc(sgt[now].maxn,mid)) swap(k,sgt[now].maxn);
                //只需要判断两条直线左右端点大小
                if(calc(k,l) > calc(sgt[now].maxn,l)) modify(l,mid,k);
                else modify(mid+1,r,k);
            }
```

```

            k = tp;
            if(calc(k,l) <= calc(sgt[now].minn,l) && calc(k,r) <= calc(sgt[now].minn,r)) sgt[now].minn = k;
            else if(calc(k,l) < calc(sgt[now].minn,l) || calc(k,r) < calc(sgt[now].minn,r)){
                int mid = (l+r)>>1;
                //让非优势线段向下更新
                if(calc(k,mid) < calc(sgt[now].minn,mid)) swap(k,sgt[now].minn);
                //只需要判断两条直线左右端点大小
                if(calc(k,l) < calc(sgt[now].minn,l)) modify(l,mid,k);
                else modify(mid+1,r,k);
            }
        }
    }
    else{
        int mid = (l+r)>>1;
        if(k.l <= mid) modify(l,mid,k);
        if(k.r > mid) modify(mid+1,r,k);
    }
}

ll Query(int l,int r,int x){
    int now = get_id(l,r);
    ll ans = max(calc(sgt[now].maxn,x),-calc(sgt[now].minn,x));
    if(l == r) return ans;
    else{
        int mid = (l+r)>>1;
        if(x <= mid) return max(ans,Query(l,mid,x));
        else return max(ans,Query(mid+1,r,x));
        return ans;
    }
}

int main()
{
    scanf("%d%d",&n,&m);
    rep(i,1,n) scanf("%lld",&a[i]);
    rep(i,1,n) hp[i].push_back({1,0,a[i],0});
    b[++tot] = 0;
    rep(i,1,m){
        ll t; scanf("%lld",&t); b[++tot] = t;
        char op[10]; scanf("%s",op+1);
        if(op[1] == 'c'){
            int k; ll x; scanf("%d%lld",&k,&x);
            q[i].t = t; q[i].x = x; q[i].k = k; q[i].op = 0;
        }
        else q[i].op = 1, q[i].t = t;
    }
    tot = unique(b+1,b+1+tot)-b-1;
    rep(i,1,m){
        if(q[i].op == 0){
            q[i].t = find(q[i].t);
            int len = hp[q[i].k].size();
            ll pos = hp[q[i].k][len-1].x*b[q[i].t]+hp[q[i].k][len-1].pos;
            ll now = -b[q[i].t]*q[i].x+pos;
            hp[q[i].k][len-1].t2 = q[i].t;
            hp[q[i].k].push_back({q[i].t,q[i].x,now,0});
        }
```

```
        else q[i].t = find(q[i].t);
    }
    rep(i,1,n){
        int len = hp[i].size();
        hp[i][len-1].t2 = tot;
    }
    rep(i,1,n){
        line tmp; tmp.l = hp[i][rp[i]].t;
        tmp.r = hp[i][rp[i]].t2;
        tmp.k = hp[i][rp[i]].x;
        tmp.b = hp[i][rp[i]].pos;
        rp[i]++;
        modify(1,tot,tmp);
    }
    rep(i,1,m){
        if(q[i].op == 0){
            int xp = q[i].k;
            line tmp; tmp.l = hp[xp][rp[xp]].t;
            tmp.r = hp[xp][rp[xp]].t2;
            tmp.k = hp[xp][rp[xp]].x;
            tmp.b = hp[xp][rp[xp]].pos;
            rp[xp]++;
            modify(1,tot,tmp);
        }
        else{
            ll ans = Query(1,tot,q[i].t);
            printf("%lld\n",ans);
        }
    }
    return 0;
}
```

4.12.3 李超树 + 两个 lazy

题意:
一共 N 个商店，$1-N$ 依次编号。购买一件纪念品的花费为纪念品的花费加上税费。初始没有纪念品，税费为 0。现有三种操作。
1 u v a b: 新增纪念品, 编号为 x 的商店，新增 b+(x-u)*a 的商品，$u \le x \le v$
2 u v a b: 税费调整操作, 表示编号为 x 的商店，税费增加 b+(x-u)*a，$u \le x \le v$
3 i: 询问操作，询问顾客在编号为 i 的商店，购买一件纪念品最高的花费是多少。

思路:
第一种操作就是李超树基础操作。第二种操作可以将 b 和 a 分开维护，对于 b 的维护只需打 $lazy$ 区间更新即可，对于 a 的维护，我们可以这样思考，对于编号为 x 的商店，税费增加 $b+(x-u)*a=(b-u*a)+x*a$，因此我们将 $b-u*a$ 统一维护，再单独维护 a。最后每个点的税费即为 $x*sum(a)+sum(b)$。

此题还需注意将物品离散化，维护线段的同时维护两个 sum，两个 $lazy$，每次对税费和物品分开求最大值，累加到答案即可。当然维护 sum 的过程也可以用树状数组完成。

```
#include <bits/stdc++.h>
#define rep(i,a,b) for(int i = a; i <= b; i++)
typedef long long ll;
const int N = 6*1e5+100;
const int M = 1e5+100;
const ll inf = -5*1e18;
using namespace std;

```

```
int n,m;
struct line{
    ll k,b;
    int l,r;
};
struct Node{
    line t;
    int flag;
    Node() {flag = 0;}
}sgt[2*N];
int b[N],tot;
ll sum1[2*N],sum2[2*N],lazy1[2*N],lazy2[2*N];
struct query{
    int op; int u,v,a,b;
}q[N];
inline int get_id(int l,int r) {return (l+r)|(l!=r);}
//计算坐标
inline ll calc(line t,ll pos) {return (t.k*b[pos]+t.b);}

void modify(int l,int r,line k){
    int now = get_id(l,r);
    if(k.l <= l && r <= k.r){
        if(!sgt[now].flag) sgt[now].flag = 1, sgt[now].t = k;
        else if(calc(k,l) >= calc(sgt[now].t,l) && calc(k,r) >= calc(sgt[now].t,r)) sgt[now].t = k;
        else if(calc(k,l) > calc(sgt[now].t,l) || calc(k,r) > calc(sgt[now].t,r)){
            int mid = (l+r)>>1;
            if(calc(k,mid) > calc(sgt[now].t,mid)) swap(k,sgt[now].t);
            if(calc(k,l) > calc(sgt[now].t,l)) modify(l,mid,k);
            else modify(mid+1,r,k);
        }
    }
    else{
        int mid = (l+r)>>1;
        if(k.l <= mid) modify(l,mid,k);
        if(k.r > mid) modify(mid+1,r,k);
    }
}

void push_down(int l,int r){
    int now = get_id(l,r), mid = (l+r)>>1, ls = get_id(l,mid), rs = get_id(mid+1,r);
    if(lazy1[now]){
        sum1[ls] += lazy1[now]; sum1[rs] += lazy1[now];
        lazy1[ls] += lazy1[now]; lazy1[rs] += lazy1[now];
        lazy1[now] = 0;
    }
    if(lazy2[now]){
        sum2[ls] += lazy2[now]; sum2[rs] += lazy2[now];
        lazy2[ls] += lazy2[now]; lazy2[rs] += lazy2[now];
        lazy2[now] = 0;
    }
}

void update(int l,int r,int L,int R,ll a,ll b){
    int now = get_id(l,r);
    if(L <= l && r <= R){
        sum1[now] += a;
        sum2[now] += b;
        lazy1[now] += a;
```

```
        lazy2[now] += b;
        return;
    }
    push_down(l,r);
    int mid = (l+r)>>1;
    if(L <= mid) update(l,mid,L,R,a,b);
    if(R > mid) update(mid+1,r,L,R,a,b);
    sum1[now] = sum1[get_id(l,mid)]+sum1[get_id(mid+1,r)];
    sum2[now] = sum2[get_id(l,mid)]+sum2[get_id(mid+1,r)];
}

ll query1(int l,int r,int pos){
    int now = get_id(l,r);
    if(l == r){
        if((sgt[now].t).l <= pos && (sgt[now].t).r >= pos) return calc(sgt[now].t,pos);
        else return inf;
    }
    int mid = (l+r)>>1;
    ll ans = inf;
    if((sgt[now].t).l <= pos && (sgt[now].t).r >= pos) ans = calc(sgt[now].t,pos);
    if(pos <= mid) return max(query1(l,mid,pos),ans);
    else return max(query1(mid+1,r,pos),ans);
}

pair<ll,ll> query2(int l,int r,int pos){
    int now = get_id(l,r);
    if(l == r) return make_pair(sum1[now],sum2[now]);
    int mid = (l+r)>>1;
    push_down(l,r);
    if(pos <= mid) return query2(l,mid,pos);
    else return query2(mid+1,r,pos);
}

int main()
{
    scanf("%d%d",&n,&m);
    rep(i,1,m){
        scanf("%d",&q[i].op);
        if(q[i].op != 3){
            scanf("%d%d%d%d",&q[i].u,&q[i].v,&q[i].a,&q[i].b);
            b[++tot] = q[i].u; b[++tot] = q[i].v;
        }
        else{
            scanf("%d",&q[i].u);
            b[++tot] = q[i].u;
        }
    }
    sort(b+1,b+1+tot);
    tot = unique(b+1,b+1+tot)-b-1;
    rep(i,1,m){
        if(q[i].op != 3){
            q[i].u = lower_bound(b+1,b+1+tot,q[i].u)-b;
            q[i].v = lower_bound(b+1,b+1+tot,q[i].v)-b;
        }
        else q[i].u = lower_bound(b+1,b+1+tot,q[i].u)-b;
    }
    rep(i,1,m){
        if(q[i].op == 1){
            line tmp; tmp.l = q[i].u, tmp.r = q[i].v;
```

```
            tmp.k = q[i].a, tmp.b = (ll)q[i].b-(ll)q[i].a*(ll)b[q[i].u];
            modify(1,tot,tmp);
        }
        else if(q[i].op == 2){
            update(1,tot,q[i].u,q[i].v,q[i].a,(ll)q[i].b-(ll)q[i].a*(ll)b[q[i].u]);
        }
        else{
            ll ans1 = query1(1,tot,q[i].u);
            pair<ll,ll> ans2 = query2(1,tot,q[i].u);
            if(ans1 == inf) printf("NA\n");
            else{
                ll ans = ans1+ans2.second+ans2.first*(ll)b[q[i].u];
                printf("%lld\n",ans);
            }
        }
    }
    return 0;
}
```

4.12.4 树剖 + 李超树 + 区间直线最大值

题意：
一棵 n 个点的树，两个操作。操作 1，选择两个点 s 和 t，对于该路径上的一个点 r，在该点上添加的数字是 $a*dis+b$，dis 为点 r 到点 s 的距离。操作 2，给出两个点 s 和 t，在 s 到 t 路径上选择一个点，再在这个点上选择一个数字，要求选的数字最小。每个点都有一个初始数字，为 123456789123456789。$(n \le 10^5, m \le 10^5)$

思路：
树链剖分后维护李超树。第二种操作好处理，只需在线段树每个节点上维护一个最小值，每次插入直线都动态维护即可。第一种操作需要分类讨论，$lca = lca(s,t)$。对于 s 到 lca 上的节点 x，每次添加数字为 $a*(dis[s]-dis[x])+b$，因此 $dis[x]$ 为横坐标，k 为 $-a$，b 为 $a*dis[s]+b$。对于 lca 到 t 上的节点 x，每次添加数字为 $a*(dis[s]+dis[x]-2*dis[lca])+b$，因此 $dis[x]$ 为横坐标，k 为 a，b 为 $a*(dis[s]-2*dis[lca])+b$。

因此插入直线时对两种情况分类讨论插入直线即可完成。

```
#include <bits/stdc++.h>
#define rep(i,a,b) for(int i = a; i <= b; i++)
typedef long long ll;
const int N = 1e5+100;
const ll inf = 1234567891234567 89ll;
using namespace std;

int n,m,tot,head[N],fa[N],d[N],son[N],siz[N],top[N],id[N],rk[N];
ll dis[N];
struct Edge{
    int to,next;
    ll w;
}e[2*N];
struct line { ll k,b; int l,r;};
struct Node{
    line t;
    ll mn;
}sgt[2*N];
inline ll calc(line t, ll pos) {return t.k*pos+t.b;}
inline int get_id(int l, int r) { return (l + r) | (l != r); } //将叶子结点赋成偶数，其余节点赋成奇数，编号最大为2n

void add(int x,int y,ll w){
```

```
    e[++tot].to = y, e[tot].next = head[x], head[x] = tot, e[tot].w = w;
}

void dfs1(int x){
    siz[x] = 1, d[x] = d[fa[x]]+1, son[x] = 0;
    for(int v,i = head[x]; i; i = e[i].next){
        if((v = e[i].to) != fa[x]){
            dis[v] = dis[x]+e[i].w; fa[v] = x; dfs1(v); siz[x] += siz[v];
            if(siz[son[x]] < siz[v]) son[x] = v;
        }
    }
}

void dfs2(int x,int tp){
    top[x] = tp; id[x] = ++tot; rk[tot] = x;
    if(son[x]) dfs2(son[x],tp);
    for(int v,i = head[x]; i; i = e[i].next)
        if((v = e[i].to) != fa[x] && v != son[x]) dfs2(v,v);
}

void build(int l,int r){
    int now = get_id(l,r);
    sgt[now].t = {0,inf,l,r}; sgt[now].mn = inf;
    if(l < r){
        int mid = (l+r)>>1;
        build(l,mid); build(mid+1,r);
    }
}

void insert(int l,int r,line k){
    int now = get_id(l,r);
    if(k.l <= l && r <= k.r){
        if(calc(k,dis[rk[l]]) <= calc(sgt[now].t,dis[rk[l]]) && calc(k,dis[rk[r]]) <= calc(sgt[now].t,dis[rk[r]])){
            sgt[now].t = k;
            sgt[now].mn = min(calc(k,dis[rk[l]]),calc(k,dis[rk[r]]));
            int mid = (l+r)>>1;
            if(l < r) sgt[now].mn = min(sgt[now].mn,min(sgt[get_id(l,mid)].mn,sgt[get_id(mid+1,r)].mn));
        }
        else if(calc(k,dis[rk[l]]) < calc(sgt[now].t,dis[rk[l]]) || calc(k,dis[rk[r]]) < calc(sgt[now].t,dis[rk[r]])){
            int mid = (l+r)>>1;
            if(calc(k,dis[rk[mid]]) < calc(sgt[now].t,dis[rk[mid]])) swap(k,sgt[now].t);
            if(calc(k,dis[rk[l]]) < calc(sgt[now].t,dis[rk[l]])) insert(l,mid,k);
            else insert(mid+1,r,k);
            sgt[now].mn = min(sgt[get_id(l,mid)].mn,sgt[get_id(mid+1,r)].mn);
            sgt[now].mn = min(sgt[now].mn,min(calc(sgt[now].t,dis[rk[l]]),calc(sgt[now].t,dis[rk[r]])));
        }
    }
    else{
        int mid = (l+r)>>1;
        if(k.l <= mid) insert(l,mid,k);
        if(mid < k.r) insert(mid+1,r,k);
        sgt[now].mn = min(sgt[get_id(l,mid)].mn,sgt[get_id(mid+1,r)].mn);
        sgt[now].mn = min(sgt[now].mn,min(calc(sgt[now].t,dis[rk[l]]),calc(sgt[now].t,dis[rk[r]])));
```

```
    }
}

int getlca(int x,int y){
    while(top[x] != top[y]){
        if(d[top[x]] < d[top[y]]) swap(x,y);
        x = fa[top[x]];
    }
    return (d[x]<d[y]?x:y);
}

void modify(int x,int y,ll a,ll b){
    int u = x, v = y, lca = getlca(x,y);
    while(d[top[x]] > d[lca]){
        line tmp = {-a,a*dis[u]+b,id[top[x]],id[x]};
        insert(1,n,tmp);
        x = fa[top[x]];
    }
    while(d[top[y]] > d[lca]){
        line tmp = {a,a*(dis[u]-2ll*dis[lca])+b,id[top[y]],id[y]};
        insert(1,n,tmp);
        y = fa[top[y]];
    }
    if(x == lca){
        line tmp = {a,a*(dis[u]-2ll*dis[lca])+b,id[x],id[y]};
        insert(1,n,tmp);
    }
    else{
        line tmp = {-a,a*dis[u]+b,id[y],id[x]};
        insert(1,n,tmp);
    }
}

ll getmin(int l,int r,int x,int y){
    int now = get_id(l,r);
    if(x <= l && r <= y) return sgt[now].mn;
    else{
        int mid = (l+r)>>1;
        ll ans = min(calc(sgt[now].t,dis[rk[max(l,x)]]),calc(sgt[now].t,dis[rk[min(y,r)]]));
        if(x <= mid) ans = min(ans,getmin(l,mid,x,y));
        if(mid < y) ans = min(ans,getmin(mid+1,r,x,y));
        return ans;
    }
}

ll query(int u,int v){
    ll ans = inf;
    while(top[u] != top[v]){
        if(d[top[u]] < d[top[v]]) swap(u,v);
        ans = min(ans,getmin(1,n,id[top[u]],id[u]));
        u = fa[top[u]];
    }
    if(d[u] < d[v]) ans = min(ans,getmin(1,n,id[u],id[v]));
    else ans = min(ans,getmin(1,n,id[v],id[u]));
    return ans;
}

int main()
```

```
{
    tot = 1;
    scanf("%d%d",&n,&m);
    rep(i,1,n-1){
        int u,v; ll w; scanf("%d%d%lld",&u,&v,&w);
        add(u,v,w); add(v,u,w);
    }
    tot = 0; dfs1(1); dfs2(1,1);
    build(1,n);
    rep(i,1,m){
        int op; scanf("%d",&op);
        if(op == 1){
            int s,t; ll a,b; scanf("%d%d%lld%lld",&s,&t,&a,&b);
            modify(s,t,a,b);
        }
        else{
            int s,t; scanf("%d%d",&s,&t);
            printf("%lld\n",query(s,t));
        }
    }
    return 0;
}
```

4.13 启发式合并

4.13.1 DSU 模板

适用问题:
树上启发式合并作为树上问题三剑客之一（点分治、长链剖分），以其优雅的暴力而闻名于江湖之中。

通常来说，如果一个问题可以被划分为一个个子树进行求解的问题，而且各个子儿子对答案的贡献容易添加与删除，就可以考虑使用树上启发式合并来求解。

本文主要介绍树上启发式合并的一些习题，可以从习题中仔细感受该算法的一系列特点。

算法介绍:
树上启发式合并需要两次 dfs，第一次 dfs 进行重链剖分，第二次 dfs 进行求解。

通常有一个全局的数组用于信息记录。dfs 之前，需要将这个数组赋初值。dfs 时，先递归处理轻儿子，处理完之后清空轻儿子，最后再处理重儿子，处理完之后不清空。

计算完儿子的答案之后，再递归所有轻儿子，边递归边计算答案，并将轻儿子的信息添加到全局数组中。

这个做法的时间复杂度是 $O(nlogn)$，因为每个节点直接继承了其子树中的重儿子，即每次只有轻儿子会被重复访问，访问完之后，轻儿子即会和重儿子进行合并，每次合并 sz 至少乘 2，因此每个点最多被重复访问 $logn$ 次，即总时间复杂度为 $O(nlogn)$。

算法模板:
其实树上启发式合并并没有什么模板，只需要处理好两次 dfs 的过程，然后实现插入、删除、更新三个函数即可。

以下面第一题的代码为例，给出一个大致的模板。

```
int sz[N],son[N];

void dfs1(int x){
    /* 求解重儿子 */
    sz[x] = 1;
    for(int i = head[x]; i; i = e[i].next){
```

```
        int y = e[i].to;
        dfs1(y); sz[x] += sz[y];
        if(sz[y] > sz[son[x]]) son[x] = y;
    }
}

void Delete(int x){
    /* 删除的内容 */
    for(int i = head[x]; i; i = e[i].next) Delete(e[i].to);
}

void modify(int x,int fa){
    /* 更新的内容 */
    for(int i = head[x]; i; i = e[i].next) modify(e[i].to,fa);
}

void ins(int x){
    /* 插入的内容 */
    for(int i = head[x]; i; i = e[i].next) ins(e[i].to);
}

void dfs2(int x){
    /* 求解轻儿子并清空 */
    for(int i = head[x]; i; i = e[i].next)
        if(e[i].to != son[x]) dfs2(e[i].to), Delete(e[i].to);

    /* 求解重儿子并保留 */
    if(son[x]) dfs2(son[x]);
    /* 用重儿子更新答案 */

    /* 枚举轻儿子更新答案，并加入轻儿子 */
    for(int i = head[x]; i; i = e[i].next)
        if(e[i].to != son[x]) modify(e[i].to,x), ins(e[i].to);

    /* 用所有儿子更新答案 */
}
```

4.13.2 例题 1

题意：
给定一棵 n 个点的树，以及一张 k 个点的图。树中每一个节点都控制图中的一条边，问：对于树中每一个节点，将其子树 (包括自己) 中所有节点控制的边加到图中，图中连通块个数即为这个节点的答案。

思路：
树上启发式合并裸题，但是比赛的时候没有写过启发式合并以为这样写会 T... 就没有尝试...

先具体讲一下这题的解法，再分析一下复杂度。‘用并查集维护图中的连通关系，对于当前节点，最后求其重儿子子树所形成的并查集，并将重儿子的并查集直接复用在求父节点上。对于非重儿子节点再对其子树中的边进行添加。

非常暴力的做法... 但是为什么能够快速通过呢... 我们先考虑一下序列上的启发式合并，由于每次选择将小的合并进大的，所以每个元素的大小每次至少 $*2$，因此最多被合并 log 次就可以达到 n 的大小，所以复杂度是 $nlogn$。

而对于树上的启发式合并，每次将小的子树合并进大的子树中，因此子树的大小每次至少 $*2$，因此最多 $logn$ 次子树的大小就可以达到 n，因此复杂度也是 $nlogn$。

```
#include <cstdio>
#include <iostream>
```

```
#include <cstring>
#include <vector>
#include <algorithm>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const int N = 1e5+100;
const int M = 1e5+100;
const db EPS = 1e-9;
using namespace std;

int sz[N],n,k,l[N],r[N],ans[N],f[N],num;
vector<int> p[N];

int find(int x){
    return x == f[x] ? x : (f[x] = find(f[x]));
}

void clear(int x){
    f[l[x]] = l[x], f[r[x]] = r[x];
    for(int u : p[x]) clear(u);
}

void unite(int x,int y){
    int fx = find(x), fy = find(y);
    if(fx != fy) f[fx] = fy, num--;
}

void solve(int x){
    unite(l[x],r[x]);
    for(int u : p[x]) solve(u);
}

void dfs(int x){
    int y = -1;
    for(int u : p[x])
        if(y == -1 || sz[y] < sz[u]) y = u;
    for(int u : p[x]){
        if(u == y) continue;
        num = k, dfs(u), clear(u);
    }
    num = k;
    if(y != -1) dfs(y);
    for(int u : p[x]){
        if(u != y) solve(u);
    }
    unite(l[x],r[x]);
    ans[x] = num;
}

int main()
{
    scanf("%d%d",&n,&k);
```

```
    rep(i,2,n){
        int pp; scanf("%d",&pp);
        p[pp].push_back(i);
    }
    rep(i,1,n) scanf("%d%d",&l[i],&r[i]);
    for(int i = n; i >= 1; i--){
        for(int y : p[i]) sz[i] += sz[y];
        sz[i]++;
    }
    rep(i,0,k) f[i] = i;
    dfs(1);
    rep(i,1,n) printf("%d\n",ans[i]);
    return 0;
}
```

4.13.3 例题 2

题意:
给定一棵有根树，每条边的权值是 $[a, v]$ 的一个字母。现对于每个树上点，求出最长的一条“回文”路径。“回文”路径的含义是将路径上所有的字母取出，可以组成一个回文串。$(1 \le n \le 5 * 10^5)$

思路:
树上的这类问题，我们可以依次思考点分治、树上启发式合并、长链剖分，根据该题题意，不难识别这是一道树上启发式合并问题。

确认是树上启发式合并之后，我们需要定状态。由于“回文”路径只要求所有字母可以组成一个回文串，因此我们可以利用状压的思想给每一个字母进行赋值，然后对于每个点求一个从根节点到当前点的异或和。

令 $f[i]$ 表示对于当前点 now，其子树中存在一个点 x，$value[x] = i, f[i] = dep[x]$，$x$ 为此中情况下深度最深的点。然后计算点 now 答案时，我们只需考虑三种情况。
1. now 为最长路径的一个端点
2. 最长路径经过 now
3. 最长路径在 now 子树中，不经过 now

三种情况在代码中有比较清晰的注释，不太清楚细节的朋友可以看看代码。总体来说，这题应该属于树上启发式合并的经典问题。

总结:
此题有几个思想比较可取。
1. 对每一个字母进行状压编码
2. 每一个点维护的权值是从根到该点的异或和
3. 分 3 种情况对答案进行了枚举

```
#include <bits/stdc++.h>
#define mem(a,b) memset(a,b,sizeof a);
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define per(i,a,b) for(int i = a; i >= b; i--)
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
typedef long long ll;
typedef double db;
const int inf = 1e8;
const int N = 5e5+10;
const db EPS = 1e-9;
using namespace std;

void dbg() {cout << "\n";}
template<typename T, typename... A> void dbg(T a, A... x) {cout << a << ' '; dbg(x...)
    ;}
```

```
#define logs(x...) {cout << #x << " -> "; dbg(x);}

int n,a[N],tot,head[N],sz[N],son[N],f[1<<22],dep[N],ans[N]; //f[i]:表示子树中异或和为f[i]的最大深度
struct Node{
    int to,next;
}e[N];

void add(int x,int y){
    e[++tot].to = y, e[tot].next = head[x], head[x] = tot;
}

void dfs1(int x){
    ans[x] = -inf; sz[x] = 1;
    for(int i = head[x]; i; i = e[i].next){
        int y = e[i].to;
        dep[y] = dep[x]+1; a[y] ^= a[x];
        dfs1(y); sz[x] += sz[y];
        if(sz[y] > sz[son[x]]) son[x] = y;
    }
}

void Delete(int x){
    f[a[x]] = -inf;
    for(int i = head[x]; i; i = e[i].next) Delete(e[i].to);
}

void modify(int x,int fa){
    ans[fa] = max(ans[fa],f[a[x]]+dep[x]-2*dep[fa]);
    for(int i = 0; i < 22; i++)
        ans[fa] = max(ans[fa],f[a[x]^(1<<i)]+dep[x]-2*dep[fa]);
    for(int i = head[x]; i; i = e[i].next) modify(e[i].to,fa);
}

void ins(int x){
    f[a[x]] = max(f[a[x]],dep[x]);
    for(int i = head[x]; i; i = e[i].next) ins(e[i].to);
}

void dfs2(int x){
    ans[x] = 0;
    for(int i = head[x]; i; i = e[i].next)
        if(e[i].to != son[x]) dfs2(e[i].to), Delete(e[i].to);
    if(son[x]) dfs2(son[x]);
    f[a[x]] = max(f[a[x]],dep[x]);
    //路径经过x
    for(int i = head[x]; i; i = e[i].next)
        if(e[i].to != son[x]) modify(e[i].to,x), ins(e[i].to);
    //x为路径端点
    ans[x] = max(ans[x],f[a[x]]-dep[x]);
    for(int i = 0; i < 22; i++)
        ans[x] = max(ans[x],f[a[x]^(1<<i)]-dep[x]);
    //路径不经过x
    for(int i = head[x]; i; i = e[i].next) ans[x] = max(ans[x],ans[e[i].to]);
}

int main()
{
    scanf("%d",&n); tot = 1;
```

```
    rep(i,0,(1<<22)-1) f[i] = -inf; //不要忘记赋初值
    rep(i,2,n){
        int p; char s[10];
        scanf("%d%s",&p,s);
        add(p,i); a[i] = 1<<(s[0]-'a');
    }
    dfs1(1); dfs2(1);
    rep(i,1,n) printf("%d%c",ans[i]," \n"[i==n]);
    return 0;
}
```

4.14 Splay

4.14.1 普通 Splay

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const int N = 1e6+100;
const int M = 1e5+100;
const db EPS = 1e-9;
using namespace std;

int f[N],cnt[N],ch[N][2],siz[N],key[N],sz,rt;
//f[x]-x的父节点, ch[x][0]表示x的左儿子, ch[x][1]表示x的右儿子, key[x]表示x的关键点,
//cnt[x]表示x节点关键字出现的次数(权值), siz[x]表示包括x的这个子树的大小, sz为整棵树的大小, rt为整棵树的根
//空间大小为所有插入点个数, 点被删除之后空间不能复用

void init(){
    memset(f,0,sizeof f); memset(cnt,0,sizeof cnt);
    memset(ch,0,sizeof ch); memset(siz,0,sizeof siz);
    memset(key,0,sizeof key); sz = rt = 0;

}
void clear(int x) { f[x] = cnt[x] = ch[x][0] = ch[x][1] = siz[x] = key[x] = 0; }
bool get(int x) { return ch[f[x]][1] == x;}
void push_up(int x) { siz[x] = siz[ch[x][0]]+siz[ch[x][1]]+cnt[x]; } //更新节点信息, 此处为siz
void rotate(int x){
    int old = f[x], oldf = f[old], which = get(x);
    ch[old][which] = ch[x][which^1]; f[ch[old][which]] = old; //把儿子过继给爸爸, 同时处理父子两个方向上的信息
    ch[x][which^1] = old; f[old] = x; //我给我爸爸当爹, 我爸爸管我叫爸爸
    f[x] = oldf; //我的爷爷成了我的爸爸
    if(oldf) ch[oldf][ch[oldf][1]==old] = x;
    push_up(old); push_up(x); //分别维护信息
}
void splay(int x){ //将x旋为根
```

```
    for(int fa; (fa = f[x]); rotate(x))
        if(f[fa]) rotate((get(x) == get(fa))?fa:x); //如果祖父三代连成一线，则旋转父亲，否则旋转自己
    rt = x;
}

void insert(int x){ //x为权值
    if(rt == 0) {
        key[++sz] = x, rt = sz;
        cnt[sz] = siz[sz] = 1; f[sz] = ch[sz][0] = ch[sz][1] = 0;
        return;
    } //空树
    int now = rt, fa = 0;
    while(1){
        if(x == key[now]){ //这个数出现过
            cnt[now]++, push_up(now), push_up(fa), splay(now); return;
        }
        fa = now, now = ch[now][key[now]<x];
        if(now == 0){
            ++sz; siz[sz] = cnt[sz] = 1;
            ch[sz][0] = ch[sz][1] = 0;
            ch[fa][x>key[fa]] = sz; //根据加入点顺序重新编号
            f[sz] = fa, key[sz] = x, push_up(fa), splay(sz); return;
        }
    }
}

int rnk(int x){ //查询x的排名
    int now = rt, ans = 0;
    while(now){
        if(x < key[now]) now = ch[now][0];
        else{
            ans += siz[ch[now][0]];
            if(x == key[now]) { splay(now); return (ans+1);} //x在树中节点的位置
            ans += cnt[now], now = ch[now][1]; //到达右孩子处
        }
    }
    return -1; //找不到
}
int kth(int x){ //查询排名为x的数
    int now = rt;
    while(1){
        if(ch[now][0] && x <= siz[ch[now][0]]) now = ch[now][0];
        else{
            int tmp = siz[ch[now][0]]+cnt[now];
            if(x <= tmp) return key[now];
            x -= tmp, now = ch[now][1];
        }
    }
}
int pre(){ //进行splay后，x已经在根节点了，因此只用找左子树最大节点即可，返回节点编号
    int now = ch[rt][0];
    while(ch[now][1]) now = ch[now][1];
    return now;
}
int next(){ //找右子树最小
    int now = ch[rt][1];
    while(ch[now][0]) now = ch[now][0];
    return now;
```

```
}

void del(int x){
    rnk(x); //把x对应节点转到了根
    if(cnt[rt] > 1) {cnt[rt]--; push_up(rt); return;} //有多个相同的数
    if(!ch[rt][0] && !ch[rt][1]) {clear(rt); rt = 0; return;}
    if(!ch[rt][0] || !ch[rt][1]){ //只有一个儿子
        int oldrt = rt; rt = ch[rt][1]+ch[rt][0]; f[rt] = 0; clear(oldrt); return;
    }
    int oldrt = rt, leftbig = pre();
    splay(leftbig);
    ch[rt][1] = ch[oldrt][1];
    f[ch[oldrt][1]] = rt;
    clear(oldrt);
    push_up(rt);
}

int main()
{
    int m; scanf("%d",&m);
    rep(i,1,m){
        // LOG1("i",i);
        int op,x; scanf("%d%d",&op,&x);
        if(op == 1) insert(x);
        else if(op == 2) del(x);
        else if(op == 3) printf("%d\n",rnk(x));
        else if(op == 4) printf("%d\n",kth(x));
        else if(op == 5){
            insert(x);
            printf("%d\n",key[pre()]);
            del(x);
        }
        else if(op == 6){
            insert(x);
            printf("%d\n",key[next()]);
            del(x);
        }
    }
    return 0;
}
```

4.14.2 区间 Splay

```
#include<iostream>
#include<cstdio>
using namespace std;
struct{int l,r,size,dat,bit,add,rev;}a[200010];
int L[200010],R[200010],n,m,tot,root,i,x,y,z;
char str[10];
//size-子树大小, dat-节点真实值, bit-子树中最小值, add-子树区间加标记, rev-子树翻转标记

void spreadadd(int x,int y)
{
    if(!x||!y) return;
    a[x].add+=y; a[x].dat+=y; a[x].bit+=y;
}
//add是lazy标记

```

```
void spreadrev(int x)
{
    if(!a[x].rev) return;
    swap(a[x].l,a[x].r);
    if(a[x].l) a[a[x].l].rev^=1; //翻转标记下传
    if(a[x].r) a[a[x].r].rev^=1;
}

void spread(int x)
{
    spreadadd(a[x].l,a[x].add); //标记下传
    spreadadd(a[x].r,a[x].add);
    spreadrev(x); //rev表示是否要翻转
    a[x].add=a[x].rev=0;
}

inline void update(int x) //更换左右儿子后，更新其siz及bit
{
    a[x].size=a[a[x].l].size+a[a[x].r].size+1;
    a[x].bit=min(a[x].dat,min(a[a[x].l].bit,a[a[x].r].bit));
}

void turnleft(int &x) //x左旋，即x的右儿子左旋到x位置
{
    int y=a[x].r; a[x].r=a[y].l; a[y].l=x;
    update(x); update(y);   x=y;
}

void turnright(int &x) //x右旋，即x的左儿子右旋到x位置
{
    int y=a[x].l; a[x].l=a[y].r; a[y].r=x;
    update(x); update(y);   x=y;
    //先更新x，因为x是儿子
}

void splay(int &x,int y) //将x子树中第y小的数转为根
{
    if(!x) return;
    L[0]=R[0]=0; //L存储了y转到x之后，左子树中的各个节点
    //R存储y转到x之后，右子树中的各个节点
    while(1)
    {
        //lazy、旋转标记下传
        spread(x),spread(a[x].l),spread(a[x].r);
        int temp=a[a[x].l].size+1; //<=x的数的个数
        //完成了目标
        if(y==temp||(y<temp&&!a[x].l)||(y>temp&&!a[x].r)) break;
        if(y<temp) //让y的左儿子旋转上来
        {
            if(a[a[x].l].l&&y<=a[a[a[x].l].l].size) {turnright(x); if(!a[x].l) break;}
            R[++R[0]]=x; x=a[x].l;
        }
        else //让y的右儿子旋转上来
        {
            y-=temp;
            int temp=a[a[a[x].r].l].size+1;
            if(a[a[x].r].r&&y>temp) {y-=temp; turnleft(x); if(!a[x].r) break;}
            L[++L[0]]=x; x=a[x].r;
        }
```

```
    }
    L[++L[0]]=a[x].l; R[++R[0]]=a[x].r; //此处的x已经成为y所对应的节点
    //将左右子树一一连接
    for(int i=L[0]-1;i>0;i--) {a[L[i]].r=L[i+1]; update(L[i]);}
    for(int i=R[0]-1;i>0;i--) {a[R[i]].l=R[i+1]; update(R[i]);}
    a[x].l=L[1]; a[x].r=R[1]; update(x);
}

void ADD(int x,int y,int z)
{
    splay(root,y+1); //把后继转到树根
    splay(a[root].l,x-1); //把前驱转到树根左儿子
    spreadadd(a[a[root].l].r,z); //区间+, 打lazy标记
}

void INSERT(int x,int y) //将y插入到a[x]之后
{
    splay(root,x); //令x为根
    a[++tot].dat=y; a[tot].r=a[root].r; a[root].r=tot; //令y为x的右儿子, x原右儿子为y右儿子
    update(tot); update(root);
}

void DELETE(int x) //删除数组中第x个数
{
    splay(root,x); //将数组中第x个数旋转到根
    splay(a[root].r,1); //将子树中最小的数转到根, 则根节点的右儿子没有左儿子
    a[a[root].r].l=a[root].l; root=a[root].r;
    update(root);
}

void REVERSE(int x,int y)
{
    splay(root,y+1);
    splay(a[root].l,x-1);
    a[a[a[root].l].r].rev^=1; //子树中所有点都左右翻转
    //此处的操作与add处不同, 原因在于区间反转不会导致该区间维护的min发生变化, 因此可以这样操作
    //但是正常打标记的操作应该是与ADD函数一致
}

void REVOLVE(int x,int y,int z) //将[x,y-len]与[y-len+1,y]互换位置, len为z%(y-x+1)
{
    z%=y-x+1;
    if(!z)return;
    int mid=y-z; //找到右区间的第一个数
    splay(root,mid); //将mid转到根
    splay(a[root].l,x-1); //根左儿子的右儿子对应区间[x,y-len] (不包括根左儿子的右儿子)
    splay(a[root].r,y-a[a[root].l].size); //根右儿子的左儿子对应区间[y-len+2,y] (不包括根右儿子的左儿子)
    z=a[root].l;
    a[root].l=a[z].r;
    a[z].r=a[a[root].r].l;
    a[a[root].r].l=0;
    update(z); update(a[root].r); update(root);
    splay(root,1); //空出根的左儿子
    a[root].l=z;
    update(root);
}

void MIN(int x,int y)
```

```
{
    splay(root,y+1); //将根节点中第y+1小的数转为根
    splay(a[root].l,x-1); //将根节点左儿子子树中第x-1小的数转为根节点左儿子
    printf("%d\n",a[a[a[root].l].r].bit);
}

// splay(root,n+2);
// splay(a[root].l,0);
// Print(a[a[root].l].r);
void Print(int x){ //打印区间[1,n]的数，输出之前要先把这个区间的数旋转到一个子树中
    spread(x); //标记下传
    if(a[x].l) Print(a[x].l);
    printf("%d ",a[x].dat); //中序遍历
    if(a[x].r) Print(a[x].r);
}

int main()
{
    cin>>n;
    a[1].size=1; a[n+2].l=n+1; a[n+2].size=n+2;
    a[1].dat=a[n+2].dat=a[1].bit=a[0].bit=0x3fffffff;
    //所有数右移一格，2表示1，n+1表示n
    for(i=2;i<=n+1;i++) //一开始构建出一根链
    {
        scanf("%d",&a[i].dat); //dat为真实值
        a[i].l=i-1; a[i].size=i; //l表示左儿子，siz表示子树大小
        a[i].bit=min(a[i-1].bit,a[i].dat);  //bit表示最小值
    }
    a[n+2].bit=a[n+1].bit;
    root=tot=n+2; //一共n+2个节点，加了两个前驱和后续
    cin>>m;
    for(i=0;i<m;i++)
    {
        scanf("%s",&str);
        //ADD x y z 区间[x,y]每个数+z
        if(str[0]=='A') {scanf("%d%d%d",&x,&y,&z); ADD(++x,++y,z);}
        //INSERT x y 在序列第x个数之后插入y
        if(str[0]=='I') {scanf("%d%d",&x,&y); INSERT(++x,y);}
        //DELETE x 删除序列第x个数
        if(str[0]=='D') {scanf("%d",&x); DELETE(++x);}
        //MIN x y 查询区间[x,y]的最小值
        if(str[0]=='M') {scanf("%d%d",&x,&y);   MIN(++x,++y);}
        //REVERSE x y 将区间[x,y]中的数翻转，如原序列1 2 3 4 5，REVERSE 3 5，得到1 2 5 4 3
        if(str[0]=='R'&&str[3]=='E') {scanf("%d%d",&x,&y);  REVERSE(++x,++y);}
        //REVOLVE x y z 将区间[x,y]旋转z次，即将最后一个数移到第一个位置，如原序列1 2 3 4 5，REVOLVE 2 4 2，得到1 3 4 2 5
        //区间左右旋转可以互相转换
        if(str[0]=='R'&&str[3]=='O') {scanf("%d%d%d",&x,&y,&z); REVOLVE(++x,++y,z);}

    }
    return 0;
}
```

4.14.3 Splay (三个 lazy 标记)

Replace a b c, 将 [a,b] 中所有括号改为 c Swap a b, 将 [a,b] 中所有字符翻转 Invert a b, 将 [a,b] 中所有'(' 变为')', ')' 变为'(' Query a b, 询问 [a,b] 之间的字符至少要改变多少位才能变成合法的括号序列

```cpp
#include <iostream>
#include <cstdio>
#define rep(i,a,b) for(int i = a; i <= b; i++)
const int N = 1e5+100;
using namespace std;
struct Node{int l,r,size,dat,add,rev,res,l1,l2,r1,r2,sum;}a[N];
int L[N],R[N],n,m,tot,root;
char str[10];
//size-子树大小, dat-节点真实值, add-子树区间赋值标记, rev-子树翻转标记, res-子树取反标记
//l1、l2-最大、小前缀和, r1、r2-最大、小后缀和
void spreadadd(int x,int y)
{
    if(!x||!y) return;
    a[x].add=y; a[x].dat=y;
    a[x].rev = 0, a[x].res = 0;
    if(y < 0) a[x].sum = -a[x].size;
    else a[x].sum = a[x].size;
    a[x].r1 = a[x].l1 = (y==1?a[x].sum:0);
    a[x].r2 = a[x].l2 = (y==1?0:a[x].sum);
}
//add是lazy标记

void spreadrev(int x) //翻转
{
    if(!x) return;
    swap(a[x].l1,a[x].r1);
    swap(a[x].l2,a[x].r2);
    swap(a[x].l,a[x].r);
    a[x].rev ^= 1;
}

void spreadres(int x) //取反
{
    if(!x) return;
    swap(a[x].l1,a[x].l2);
    swap(a[x].r1,a[x].r2);
    a[x].sum = -a[x].sum; a[x].dat = -a[x].dat;
    a[x].l1 = -a[x].l1, a[x].l2 = -a[x].l2;
    a[x].r1 = -a[x].r1, a[x].r2 = -a[x].r2;
    a[x].res ^= 1;
}

void spread(int x)
{
    if(a[x].rev){
        spreadrev(a[x].l);
        spreadrev(a[x].r);
    }
    if(a[x].add){
        spreadadd(a[x].l,a[x].add); //标记下传
        spreadadd(a[x].r,a[x].add);
    }
    if(a[x].res){
        spreadres(a[x].l);
        spreadres(a[x].r);
    }
    a[x].add=a[x].rev=a[x].res=0;
}

```

```
inline void update(int x)
{
    int L = a[x].l, R = a[x].r;
    a[x].size=a[L].size+a[R].size+1;
    a[x].sum = a[L].sum+a[R].sum+a[x].dat;
    a[x].l1 = max(a[L].l1,a[L].sum+a[R].l1+a[x].dat);
    a[x].l2 = min(a[L].l2,a[L].sum+a[R].l2+a[x].dat);
    a[x].r1 = max(a[R].r1,a[R].sum+a[L].r1+a[x].dat);
    a[x].r2 = min(a[R].r2,a[R].sum+a[L].r2+a[x].dat);
}

void turnleft(int &x) //x左旋，即x的右儿子左旋到x位置
{
    int y=a[x].r; a[x].r=a[y].l; a[y].l=x;
    update(x); update(y);   x=y;
}

void turnright(int &x) //x右旋，即x的左儿子右旋到x位置
{
    int y=a[x].l; a[x].l=a[y].r; a[y].r=x;
    update(x); update(y);   x=y;
    //先更新x，因为x是儿子
}

void splay(int &x,int y) //将x子树中第y小的数转为根
{
    if(!x) return;
    L[0]=R[0]=0; //L存储了y转到x之后，左子树中的各个节点
    //R存储y转到x之后，右子树中的各个节点
    while(1)
    {
        //lazy、旋转标记下传
        spread(x),spread(a[x].l),spread(a[x].r);
        int temp=a[a[x].l].size+1; //<=x的数的个数
        //完成了目标
        if(y==temp||(y<temp&&!a[x].l)||(y>temp&&!a[x].r)) break;
        if(y<temp) //让y的左儿子旋转上来
        {
            if(a[a[x].l].l&&y<=a[a[a[x].l].l].size) {turnright(x); if(!a[x].l) break;}
            R[++R[0]]=x; x=a[x].l;
        }
        else //让y的右儿子旋转上来
        {
            y-=temp;
            int temp=a[a[a[x].r].l].size+1;
            if(a[a[x].r].r&&y>temp) {y-=temp; turnleft(x); if(!a[x].r) break;}
            L[++L[0]]=x; x=a[x].r;
        }
    }
    L[++L[0]]=a[x].l; R[++R[0]]=a[x].r; //此处的x已经成为y所对应的节点
    //将左右子树一一连接
    for(int i=L[0]-1;i>0;i--) {a[L[i]].r=L[i+1]; update(L[i]);}
    for(int i=R[0]-1;i>0;i--) {a[R[i]].l=R[i+1]; update(R[i]);}
    a[x].l=L[1]; a[x].r=R[1]; update(x);
}

void ADD(int x,int y,int z)
{
    splay(root,y+1); //把后继转到树根
```

```
    splay(a[root].l,x-1); //把前驱转到树根左儿子
    spreadadd(a[a[root].l].r,z); //区间+, 打lazy标记
}

void SWAP(int x,int y)
{
    splay(root,y+1);
    splay(a[root].l,x-1);
    spreadrev(a[a[root].l].r);
}

void Invert(int x,int y)
{
    splay(root,y+1);
    splay(a[root].l,x-1);
    spreadres(a[a[root].l].r);
}

void Query(int x,int y)
{
    splay(root,y+1);
    splay(a[root].l,x-1);
    int x1 = a[a[a[root].l].r].l2;
    int x2 = a[a[a[root].l].r].r1;
    x1 = -x1;
    int ans = ((x1+1)/2)+((x2+1)/2); //左端最小前缀和/2上取整+右端最大前缀和上取整
    //abs(左端最小前缀和)=x,表示左端至少有x个'('未匹配
    //abs(右端最大前缀和)=x,表示右端至少有x个')'未匹配
    printf("%d\n",ans);
}

char s[N];

int main()
{
    cin>>n>>m;
    a[1].size=1; a[n+2].l=n+1; a[n+2].size=n+2;
    a[1].dat = a[n+2].dat; update(1);
    scanf("%s",s+1);
    rep(i,2,n+1) a[i].dat = (s[i-1]=='('?1:-1);
    rep(i,2,n+2)
    {
        a[i].l=i-1; a[i].size=i; //l表示左儿子, siz表示子树大小
        update(i);
    }
    root=tot=n+2; //一共n+2个节点, 加了两个前驱和后续
    rep(i,1,m)
    {
        int x,y,z; char op[5];
        scanf("%s",str);
        if(str[0]=='R'){
            scanf("%d%d",&x,&y); scanf("%s",op);
            if(op[0] == '(') z = 1;
            else z = -1;
            ADD(++x,++y,z);
        }
        else if(str[0]=='S'){
            scanf("%d%d",&x,&y);
            SWAP(++x,++y);
```

```
        }
        else if(str[0] == 'I'){
            scanf("%d%d",&x,&y);
            Invert(++x,++y);
        }
        else if(str[0] == 'Q'){
            scanf("%d%d",&x,&y);
            Query(++x,++y);
        }
    }
    return 0;
}
```

4.15 LCT

4.15.1 LCT 模板题

一、适用问题:
动态树主要用于解决操作中带有加边、删边、换根的一系列问题，即树结构发生变化的问题，理论上来说，树链剖分的问题都能用 LCT 进行解决。

二、LCT 函数解析
LCT 本质是上对树进行实链剖分，实链剖分的意思就是将一棵树分成多条链，链中的边称为实边，链与链之间的边则称为虚边，每条链都是一个 $splay$，在 $splay$ 中进行中序遍历即可还原原来的树结构。而 LCT 就是不断进行虚边、实边转换的一个算法。

LCT 中一共有 $clear$、$pushUp$、$pushDown$、$update$、$rotate$、$splay$、$access$、$makeRoot$、$link$、cut、$find$、$split$ 等函数，下面大致介绍一下每个函数的具体作用以及一些坑点，更多的是提纲挈领的作用，想要从最基础的地方开始学的话，推荐 [oiwiki](https://oi-wiki.org/ds/lct/)。

(1) 简单函数（仅操作单个 $splay$ 的函数）
1. $clear(x)$：清除一个点的信息，如父亲、左右儿子、标记、维护信息等信息。
2. $pushUp(x)$：由左右儿子的信息更新父节点的信息，与线段树的 $pushUp()$ 函数没有太大差别。
3. $pushDown(x)$：将当前节点的标记下放到儿子节点，如加、减、翻转等标记。
4. $update(x)$：一直递归到根节点，然后把标记信息不断下放，没有涉及任何虚实边的转换。

```
void update(int p){ //递归地从上到下pushDown信息
  if(!isRoot(p)) update(f[p]);
  pushDown(p);
}
```

5. $rotate(x)$：将当前节点向上旋转一层，可以自己模拟一下。此处改变了 $splay$ 的内部结构，即子节点发生了改变，因此需要进行 $pushUp$，但是仍然没有进行任何虚实边的转换。

```
inline void rotate(int x){ //将x向上旋转一层的操作
  int y = f[x], z = f[y], k = Get(x);
  if(!isRoot(y)) ch[z][ch[z][1] == y] = x;
  ch[y][k] = ch[x][!k], f[ch[y][k]] = y;
  ch[x][!k] = y, f[y] = x, f[x] = z;
  pushUp(y); //要先pushUp(y)
  pushUp(x);
}
```

6. $splay(x)$：将当前点旋转到 $splay$ 的根节点，$splay$ 到根节点作用在于不需要在向上进行更新。比如你现在要修改 x 的点权，但是每个节点还要维护子树 sum 的信息，如果 x 不是其所在 $splay$ 的根节点，那么修改 x 的点权势必影响到其祖先节点的 sum 信息，因此需要将 x 旋转为其所在 $splay$ 的根后再进行单点修改。注意 $splay$ 函数也没有进行实边和虚边的转换。

```
 inline void splay(int x){ //把x旋转到当前splay的根
   update(x); //将上面的标记完全下放
   for(int fa; fa = f[x], !isRoot(x); rotate(x)){
     if(!isRoot(fa)) rotate(Get(fa) == Get(x) ? fa : x);
   }
 }
```

以上函数都属于 LCT 函数中的简单函数，因为这些函数都只是在单个 $splay$ 中进行操作，不涉及任何虚实边的转换。

(2) 复杂函数（涉及多个 $splay$ 的操作，进行虚实边转换）
1. $access(x)$：将点 x 到根的路径经过的点放入同一个 $splay$ 中，且这个 $splay$ 中仅包含从 x 到根路径上经过的点。具体操作即是将 x 点不断转成其所在 $splay$ 的根，然后再进行虚实边转换一直到根。此处 $access$ 函数有返回值，返回值为最后构成的 $splay$ 的根节点。

```
inline int access(int x){ //把从根到x的所有点放在一条实链里，返回这个splay的根
    int p; //每次改变右儿子的值，因为整棵树是中序遍历，放入右儿子才能保证先遍历父亲再遍历儿子
    for(p = 0; x; p = x, x = f[x]){
      splay(x), ch[x][1] = p, pushUp(x);
    }
    return p;
}
```

2. $makeRoot(x)$：换根操作，将点 x 变成当前树的根。具体过程为先 $access$ 点 x，然后再将点 x 旋转为其 $splay$ 所在根，然后将所有节点的左右儿子翻转即可。

```
inline void makeRoot(int p){ //使x点成为整棵树的根
    access(p); splay(p);
    swap(ch[p][0],ch[p][1]); //把整条链反向
    rev[p] ^= 1;
}
```

3. $split(x, y)$：从树中拎出 $x \to y$ 的路径，返回该路径的 $splay$ 根节点，可以查询路径最大值、点权和、边权和等信息。

```
inline int split(int x,int y){
    makeRoot(x);
    return access(y);
}
```

4. $find(x)$：即返回点 x 所在树的根节点，不是所在 $splay$ 中的根节点，用于判断两点是否连通。

```
 inline int find(int p){ //找到x所在树的根节点编号
   access(p), splay(p);
   while(ls) pushDown(p), p = ls;
   return p;
 }
```

5. $link(x, y)$：连接树中 x、y 两点之间的边，无边变虚边，如果题目中没有保证操作一定合法，则需要自行判断 x、y 是否已经连通。

```
inline void link(int x,int y){ //在x、y两点间连一条边，连接了虚边
   if (find(x) != find(y)) makeRoot(x), f[x] = y;
 }
```

6. $cut(x, y)$：断开树中 x、y 两点之间的实边，两个点同时断开即可。

```
inline void cut(int x,int p){ //把x、y两点间边删掉，此处删除的是实边，注意实边和虚边的区别
   makeRoot(x), access(p), splay(p);
   if (ls == x && !rs) ls = f[x] = 0;
 }
```

三、具体细节

(1) 单点修改
由于 LCT 中维护了多个 $splay$，因此单点修改需要把该点修改的信息不断上传，所以我们需要先将点 x 旋转到 $splay$ 的根或者整棵树的根，然后再进行单点修改。

如果题中只需要维护实链信息，则只需要旋转到 $splay$ 的根，如果需要同时维护实链和虚链信息，即整棵子树的信息的话，则需要令该点为树根，即调用 $makeRoot()$ 函数。

(2) 维护边权
由于 LCT 是不断地进行虚边、实边转换，因此没有固定的边结构，所以直接维护边权十分困难，因此我们将边转成点，边 (a,b) 成为一个点 x，$link(a,x)$、$link(x,b)$ 即可。

(3) 维护子树信息
普通 LCT 只能维护具有可减性的子树信息，比如子树大小，子树贡献等，而子树 max、min 等问题则不具有可减性，难以维护。

维护子树信息主要在于维护实边信息和虚边信息，而进行实虚转换的函数只有 $makeRoot()$、$access()$、$link()$ 三个函数，只需要在该三个函数进行一定的修改即可，下面习题中包含了该问题可供参考。

四、例题

题意:
n 个点一棵树，支持四种操作。$(1 \le n, q \le 10^5, 0 \le c \le 10^4)$
1. $+\ u\ v\ c$，将 u 到 v 的路径上的点的权值都加上 c。
2. $-\ u_1\ v_1\ u_2\ v_2$，将树中原有的边 (u_1, v_1) 删除，加入一条新边 (u_2, v_2)，保证操作完之后仍然是一颗树。
3. $*\ u\ v\ c$，将 u 到 v 的路径上的点的权值都乘上 c。
4. $/\ u\ v$，询问 u 到 v 的路径上的点的权值和，答案 mod 51061。

思路:
三个涉及到路径的操作，都是先把 u 变成树根，然后 $access(v)$，即拉起一条 u 到 v 的路径，使得 u 到 v 路径上的点都在一个 $splay$ 中，然后获得这个 $splay$ 的根节点，即可对根节点打标记完成。

删边则是令 u 为根，拉起 u 到 v 的路径，将 v 旋转成 $splay$ 的根，然后儿子与父亲双向断开联系。加边则是令 u 为根，然后使 u 的父亲变成 v。

总结:
这题应该算是 LCT 的模板题，涉及的操作都是基础操作，没有太多思维上的难点。

```
#include <bits/stdc++.h>
#define rep(i,a,b) for(int i = a; i <= b; i++)
using namespace std;
#define int long long
const int N = 100010;
const int mod = 51061;
int n, q, u, v, c;
char op;

struct LCT{
  #define ls ch[p][0]
  #define rs ch[p][1]
  #define Get(p) (ch[f[p]][1] == p)
  int ch[N][2], f[N], sum[N], val[N], siz[N], rev[N], add[N], mul[N];

  inline void clear(int p){ //清除这个点的信息
    ch[p][0] = ch[p][1] = f[p] = siz[p] = val[p] = sum[p] = rev[p] = add[p] = 0;
    mul[p] = 1;
  }

```

```
inline int isRoot(int p){
  clear(0);
  return ch[f[p]][0] != p && ch[f[p]][1] != p;
}

inline void pushUp(int p){
  clear(0);
  siz[p] = (siz[ls] + 1 + siz[rs]) % mod;
  sum[p] = (sum[ls] + val[p] + sum[rs]) % mod;
}

inline void pushDown(int p){
  clear(0);
  if(mul[p] != 1){ //乘法
    if(ls){ //左儿子
      mul[ls] = (mul[ls] * mul[p]) % mod;
      val[ls] = (val[ls] * mul[p]) % mod;
      sum[ls] = (sum[ls] * mul[p]) % mod;
      add[ls] = (add[ls] * mul[p]) % mod;
    }
    if(rs){ //右儿子
      mul[rs] = (mul[rs] * mul[p]) % mod;
      val[rs] = (val[rs] * mul[p]) % mod;
      sum[rs] = (sum[rs] * mul[p]) % mod;
      add[rs] = (add[rs] * mul[p]) % mod;
    }
    mul[p] = 1;
  }
  if(add[p]){
    if(ls){
      add[ls] = (add[ls] + add[p]) % mod;
      val[ls] = (val[ls] + add[p]) % mod;
      sum[ls] = (sum[ls] + add[p] * siz[ls] % mod) % mod;
    }
    if(rs){
      add[rs] = (add[rs] + add[p]) % mod;
      val[rs] = (val[rs] + add[p]) % mod;
      sum[rs] = (sum[rs] + add[p] * siz[rs] % mod) % mod;
    }
    add[p] = 0;
  }
  if(rev[p]){
    if(ls) rev[ls] ^= 1, swap(ch[ls][0],ch[ls][1]);
    if(rs) rev[rs] ^= 1, swap(ch[rs][0],ch[rs][1]);
    rev[p] = 0;
  }
}

void update(int p){ //递归地从上到下pushDown信息
  //没有将实边变成虚边
  if(!isRoot(p)) update(f[p]);
  pushDown(p);
}

inline void rotate(int x){ //将x向上旋转一层的操作
  //没有将实边变成虚边
  int y = f[x], z = f[y], k = Get(x);
  if(!isRoot(y)) ch[z][ch[z][1] == y] = x;
  ch[y][k] = ch[x][!k], f[ch[y][k]] = y;
```

```
    ch[x][!k] = y, f[y] = x, f[x] = z;
    pushUp(y); //要先pushUp(y)
    pushUp(x);
  }

  inline void splay(int x){ //把x旋转到当前splay的根
    //没有将实边变成虚边
    update(x); //将上面的标记完全下放
    for(int fa; fa = f[x], !isRoot(x); rotate(x)){
      if(!isRoot(fa)) rotate(Get(fa) == Get(x) ? fa : x);
    }
  }

  inline int access(int x){ //把从根到x的所有点放在一条实链里，返回这个splay的根
    //进行了边的虚实变换
    int p; //每次改变右儿子的值，因为整棵树是中序遍历，放入右儿子才能保证先遍历父亲再遍历儿子
    for(p = 0; x; p = x, x = f[x]){
      splay(x), ch[x][1] = p, pushUp(x);
    }
    return p;
  }

  inline void makeRoot(int p){ //使x点成为整棵树的根
    access(p); splay(p);
    swap(ch[p][0],ch[p][1]); //把整条链反向
    rev[p] ^= 1;
  }

  inline void link(int x,int y){ //在x、y两点间连一条边，连接了虚边
    if (find(x) != find(y)) makeRoot(x), f[x] = y;
  }

  inline void cut(int x,int p){ //把x、y两点间边删掉，此处删除的是实边，注意实边和虚边的区别
    makeRoot(x), access(p), splay(p);
    if (ls == x && !rs) ls = f[x] = 0;
  }

  inline int find(int p){ //找到x所在树的根节点编号
    access(p), splay(p);
    while(ls) pushDown(p), p = ls;
    return p;
  }
  //中序遍历即可还原树结构
  void print(int p){
    if(!p) return;
    pushDown(p);
    print(ls);
    printf("%lld ",p);
    print(rs);
  }
}st;

signed main() {
  scanf("%lld%lld", &n, &q);
  for (int i = 1; i <= n; i++) st.val[i] = 1;
  for (int i = 1; i < n; i++) {
    scanf("%lld%lld", &u, &v);
    st.link(u,v);
  }
```

```
  while (q--) {
    scanf(" %c%lld%lld", &op, &u, &v);
    if (op == '+') { //+ u v c, u到v的路径上的点权值+c
      scanf("%lld", &c);
      //u变成树根，拉起v到u的链，把v旋到splay的根
      st.makeRoot(u); v = st.access(v);
      st.val[v] = (st.val[v] + c) % mod;
      st.sum[v] = (st.sum[v] + st.siz[v] * c % mod) % mod;
      st.add[v] = (st.add[v] + c) % mod;
    }
    if (op == '-') { //- u1 v1 u2 v2, 删除(u1,v1), 加上(u2,v2)
      st.cut(u,v);
      scanf("%lld%lld", &u, &v);
      st.link(u,v);
    }
    if (op == '*') { //* u v c, u到v的路径乘上c
      scanf("%lld", &c);
      st.makeRoot(u); v = st.access(v);
      st.val[v] = st.val[v] * c % mod;
      st.sum[v] = st.sum[v] * c % mod;
      st.mul[v] = st.mul[v] * c % mod;
    }
    if (op == '/'){ //u v, 询问u到v路径权值和
      st.makeRoot(u); v = st.access(v);
      printf("%lld\n", st.sum[v]);
    }
  }
  return 0;
}
```

4.15.2 边权 LCT

题意:
n 个点一棵树，支持两种操作。$(1 \le n, q \le 10^4)$
1. CHANGE i t_i，将第 i 条边的边权改为 t_i。
2. QUERY a b，查询树中点 a 到点 b 的路径中的边权最大值。

思路:
边权 LCT，需要对于每一条边建一个节点，即树中一共有 $2*n-1$ 个节点，每个边节点与上下两个节点连边。

建边需要先确定每个节点的边权之后再进行 $link$，因为点修改会对该节点的祖先节点产生影响，需要将该点旋至 $splay$ 端点后才能进行修改。

总结:
总结一下构建 LCT 构建的关键，构建 LCT 需要先对各个顶点赋值然后再进行 $link$ 操作，若是 $link$ 之后再赋值相当于点修改，而点修改需要将点旋为 $splay$ 根之后才能进行更改。

```
#include <bits/stdc++.h>
#define rep(i,a,b) for(int i = a; i <= b; i++)
using namespace std;
const int N = 20000+10;
int n,val[N];

struct LCT{
  #define ls ch[p][0]
  #define rs ch[p][1]
  #define Get(p) (ch[f[p]][1] == p)
```

```
int ch[N][2], f[N], maxn[N], val[N], siz[N], rev[N];

inline void clear(int p){ //清除这个点的信息
  ch[p][0] = ch[p][1] = f[p] = siz[p] = val[p] = maxn[p] = 0;
}

inline int isRoot(int p){
  clear(0);
  return ch[f[p]][0] != p && ch[f[p]][1] != p;
}

inline void pushUp(int p){
  clear(0);
  siz[p] = siz[ls] + 1 + siz[rs];
  maxn[p] = max(val[p],max(maxn[ls],maxn[rs]));
}

inline void pushDown(int p){
  clear(0);
  if(rev[p]){
    if(ls) rev[ls] ^= 1, swap(ch[ls][0],ch[ls][1]);
    if(rs) rev[rs] ^= 1, swap(ch[rs][0],ch[rs][1]);
    rev[p] = 0;
  }
}

void update(int p){ //递归地从上到下pushDown信息
  if(!isRoot(p)) update(f[p]);
  pushDown(p);
}

inline void rotate(int x){ //将x向上旋转一层的操作
  int y = f[x], z = f[y], k = Get(x);
  if(!isRoot(y)) ch[z][ch[z][1] == y] = x;
  ch[y][k] = ch[x][!k], f[ch[y][k]] = y;
  ch[x][!k] = y, f[y] = x, f[x] = z;
  pushUp(y); //要先pushUp(y)
  pushUp(x);
}

inline void splay(int x){ //把x旋转到当前splay的根
  update(x); //将上面的标记完全下放
  for(int fa; fa = f[x], !isRoot(x); rotate(x)){
    if(!isRoot(fa)) rotate(Get(fa) == Get(x) ? fa : x);
  }
}

inline int access(int x){ //把从根到x的所有点放在一条实链里，返回这个splay的根
  int p;
  for(p = 0; x; p = x, x = f[x]){
    splay(x), ch[x][1] = p, pushUp(x);
  }
  return p;
}

inline void makeRoot(int p){ //使x点成为整棵树的根
  access(p); splay(p);
  swap(ch[p][0],ch[p][1]); //把整条链反向
  rev[p] ^= 1;
```

```
  }

  inline void link(int x,int y){ //在x、y两点间连一条边
    makeRoot(x), f[x] = y; //dfs建树，每条边都是有效的，因此不需要判断是否有效
  }
}st;

int main(){
  int _; scanf("%d",&_);
  while(_--){
    scanf("%d",&n);
    rep(i,0,2*n) st.clear(i);
    rep(i,1,n-1){
      int a,b,c; scanf("%d%d%d",&a,&b,&c);
      st.val[i+n] = c;
      st.link(a,i+n);
      st.link(i+n,b);
    }
    while(1){
      char s[20]; scanf("%s",s);
      if(s[0] == 'D') break;
      else if(s[0] == 'C'){
        int a,b; scanf("%d%d",&a,&b);
        st.splay(a+n); //先转为splay根节点
        st.val[a+n] = b;
      }
      else{
        int a,b; scanf("%d%d",&a,&b);
        st.makeRoot(a);
        b = st.access(b);
        printf("%d\n",st.maxn[b]);
      }
    }
  }
  return 0;
}
```

4.15.3 LCT 最大连续和

题意:
n 个点一棵树，每个节点有一个值，支持两种操作。$(1 \le n, q \le 10^5)$
1 a b，查询树中点 a 到点 b 的路径中最大连续和
2 a b c，将树中点 a 到点 b 路径中所有点的值改为 c

思路:
对每一个点维护一个 $lc[i]$、$rc[i]$、$maxn[i]$ 表示点 i 子树中左连续的最大值、右连续的最大值以及整棵子树中的最大连续值。

需要注意一点，交换左右儿子的时候，还需要把每个节点的 lc 和 rc 进行交换，其余细节见代码。

```
#include <bits/stdc++.h>
#define rep(i,a,b) for(int i = a; i <= b; i++)
using namespace std;
typedef long long ll;
const ll inf = 1e9+100;
const int N = 1e5+10;
int n,Q;

```

```
struct LCT{
  #define ls ch[p][0]
  #define rs ch[p][1]
  #define Get(p) (ch[f[p]][1] == p)
  int ch[N][2], f[N];
  ll maxn[N], sum[N], lc[N], rc[N], val[N], siz[N], lazy[N];
  bool rev[N];

  inline void clear(int p){ //清除这个点的信息
    ch[p][0] = ch[p][1] = f[p] = val[p] = maxn[p] = lc[p] = sum[p] = rc[p] = siz[p] = 0;
  }

  inline int isRoot(int p){
    return ch[f[p]][0] != p && ch[f[p]][1] != p;
  }

  inline void pushUp(int p){
    siz[p] = siz[ls] + 1 + siz[rs];
    sum[p] = val[p] + sum[ls] + sum[rs]; //ls、rs可能为0
    maxn[p] = max(maxn[ls],max(maxn[rs],rc[ls]+lc[rs]+val[p]));
    lc[p] = max(lc[ls],sum[ls]+val[p]+lc[rs]);
    rc[p] = max(rc[rs],sum[rs]+val[p]+rc[ls]);
  }

  inline void pushDown(int p){
    if(rev[p]){
      if(ls) rev[ls] ^= 1, swap(ch[ls][0],ch[ls][1]);
      if(rs) rev[rs] ^= 1, swap(ch[rs][0],ch[rs][1]);
      swap(lc[ls],rc[ls]); swap(lc[rs],rc[rs]); //交换左右儿子时还要交换左右连续最大值
      rev[p] = 0;
    }
    if(lazy[p] != -inf){
      if(ls){
        sum[ls] = siz[ls]*lazy[p];
        val[ls] = lazy[ls] = lazy[p];
        lc[ls] = rc[ls] = maxn[ls] = lazy[p] > 0 ? sum[ls]:0;
      }
      if(rs){
        sum[rs] = siz[rs]*lazy[p];
        val[rs] = lazy[rs] = lazy[p];
        lc[rs] = rc[rs] = maxn[rs] = lazy[p] > 0 ? sum[rs]:0;
      }
      lazy[p] = -inf;
    }
  }

  void update(int p){ //递归地从上到下pushDown信息
    if(!isRoot(p)) update(f[p]);
    pushDown(p);
  }

  inline void rotate(int x){ //将x向上旋转一层的操作
    int y = f[x], z = f[y], k = Get(x);
    if(!isRoot(y)) ch[z][ch[z][1] == y] = x;
    ch[y][k] = ch[x][!k], f[ch[y][k]] = y;
    ch[x][!k] = y, f[y] = x, f[x] = z;
    pushUp(y); //要先pushUp(y)
    pushUp(x);
```

```
  }

  inline void splay(int x){ //把x旋转到当前splay的根
    update(x); //将上面的标记完全下放
    for(int fa; fa = f[x], !isRoot(x); rotate(x)){
      if(!isRoot(fa)) rotate(Get(fa) == Get(x) ? fa : x);
    }
  }

  inline int access(int x){ //把从根到x的所有点放在一条实链里, 返回这个splay的根
    int p;
    for(p = 0; x; p = x, x = f[x]){
      splay(x), ch[x][1] = p, pushUp(x);
    }
    return p;
  }

  inline void makeRoot(int p){ //使x点成为整棵树的根
    access(p);
    splay(p);
    swap(ch[p][0],ch[p][1]); //把整条链反向
    rev[p] ^= 1;
  }

  inline void link(int x,int y){ //在x、y两点间连一条边
    makeRoot(x), f[x] = y; //dfs建树, 每条边都是有效的, 因此不需要判断是否有效
  }
}st;

int main(){
  scanf("%d",&n);
  rep(i,1,n){
    scanf("%lld",&st.val[i]);
    st.siz[i] = 1; st.sum[i] = st.val[i];
    st.lazy[i] = -inf;
    st.lc[i] = st.rc[i] = st.maxn[i] = st.val[i] > 0 ? st.val[i]:0;
  }
  rep(i,1,n-1){
    int a,b; scanf("%d%d",&a,&b);
    st.link(a,b);
  }
  scanf("%d",&Q);
  while(Q--){
    int op; scanf("%d",&op);
    if(op == 1){ //a->b max
      int a,b; scanf("%d%d",&a,&b);
      st.makeRoot(a);
      b = st.access(b);
      printf("%lld\n",st.maxn[b]);
    }
    else{ //a->b to c
      int a,b; ll c; scanf("%d%d%lld",&a,&b,&c);
      st.makeRoot(a);
      b = st.access(b);
      st.val[b] = st.lazy[b] = c;
      st.sum[b] = st.siz[b]*c;
      st.lc[b] = st.rc[b] = st.maxn[b] = c > 0 ? st.sum[b]:0;
    }
  }
```

```
  return 0;
}
```

4.15.4 最小差值生成树

题意:
n 个点，m 条边的一个无向图，求边权最大值与最小值的差值最小的生成树。$(1 \le n \le 5*10^4, 1 \le m \le 2*10^5)$

思路:
关于这类特殊生成树问题，一般考虑用 LCT 动态维护树结构然后更新答案。

此题也可以这样考虑。将边按边权从小到大排序，如果 (a,b) 两点不连通，则加上该边，如果 (a,b) 两点连通，则将 $a \to b$ 路径上边权最小的边去除，然后连上当前的边。维护过程不断更新最大值与最小值的差值，不断取 min 即可。

因此只需要维护一个边权 LCT，并且维护路径最小值以及最小值点的编号，然后动态加边删边即可。还需要对在树中的边打上标记，去除的时候删去标记，用于查找整棵树中的最小边权。

```
#include <bits/stdc++.h>
#define rep(i,a,b) for(int i = a; i <= b; i++)
using namespace std;
const int N = 2e5+5e4+100;
const int M = 2e5+10;
int n, m, vis[N], pos = 1, num, ans = 1e9;
struct Edge{
  int a,b,w;
  bool operator < (Edge xx) const {
    return w < xx.w;
  }
}e[N];

void dbg() {cout << "\n";}
template<typename T, typename... A> void dbg(T a, A... x) {cout << a << ' '; dbg(x...)
    ;}
#define logs(x...) {cout << #x << " -> "; dbg(x);}

struct LCT{
  #define ls ch[p][0]
  #define rs ch[p][1]
  #define Get(p) (ch[f[p]][1] == p)
  int ch[N][2], f[N], val[N], minn[N], mpos[N], rev[N];

  inline void clear(int p){ //清除这个点的信息
    ch[p][0] = ch[p][1] = f[p] = val[p] = mpos[p] = minn[p] = 0;
  }

  inline int isRoot(int p){
    return ch[f[p]][0] != p && ch[f[p]][1] != p;
  }

  inline void pushUp(int p){
    minn[p] = val[p]; mpos[p] = p;
    if(ls && minn[ls] < minn[p]) minn[p] = minn[ls], mpos[p] = mpos[ls];
    if(rs && minn[rs] < minn[p]) minn[p] = minn[rs], mpos[p] = mpos[rs];
  }

  inline void pushDown(int p){
    if(rev[p]){
```

```
      if(ls) rev[ls] ^= 1, swap(ch[ls][0],ch[ls][1]);
      if(rs) rev[rs] ^= 1, swap(ch[rs][0],ch[rs][1]);
      rev[p] = 0;
    }
  }

  void update(int p){ //递归地从上到下pushDown信息
    if(!isRoot(p)) update(f[p]);
    pushDown(p);
  }

  inline void rotate(int x){ //将x向上旋转一层的操作
    int y = f[x], z = f[y], k = Get(x);
    if(!isRoot(y)) ch[z][ch[z][1] == y] = x;
    ch[y][k] = ch[x][!k], f[ch[y][k]] = y;
    ch[x][!k] = y, f[y] = x, f[x] = z;
    pushUp(y); //要先pushUp(y)
    pushUp(x);
  }

  inline void splay(int x){ //把x旋转到当前splay的根
    update(x); //将上面的标记完全下放
    for(int fa; fa = f[x], !isRoot(x); rotate(x)){
      if(!isRoot(fa)) rotate(Get(fa) == Get(x) ? fa : x);
    }
  }

  inline int access(int x){ //把从根到x的所有点放在一条实链里，返回这个splay的根
    int p = 0;
    for(p = 0; x; p = x, x = f[x]){
      splay(x), ch[x][1] = p, pushUp(x);
    }
    return p;
  }

  inline void makeRoot(int p){ //使x点成为整棵树的根
    access(p); splay(p);
    swap(ch[p][0],ch[p][1]); //把整条链反向
    rev[p] ^= 1;
  }

  inline void link(int x,int y){ //在x、y两点间连一条边
    // if (find(x) != find(y))
    makeRoot(x), f[x] = y;
  }

  inline void cut(int x,int p){ //把x、y两点间边删掉
    makeRoot(x), access(p), splay(p);
    if (ls == x && !rs) ls = f[x] = 0;
  }

  inline int find(int p){ //找到x所在树的根节点编号
    access(p), splay(p);
    while(ls) pushDown(p), p = ls;
    return p;
  }
}st;

signed main() {
```

```
  scanf("%d%d", &n, &m);
  rep(i,0,n) st.val[i] = st.minn[i] = 1e5;
  rep(i,1,m) scanf("%d%d%d",&e[i].a,&e[i].b,&e[i].w);
  sort(e+1,e+1+m);
  rep(i,1,m) st.val[i+n] = st.minn[i+n] = e[i].w, st.mpos[i+n] = i+n;
  rep(i,1,m){
    if(e[i].a == e[i].b) continue;
    if(st.find(e[i].a) != st.find(e[i].b)){
      st.link(e[i].a,i+n); st.link(i+n,e[i].b);
      num++; vis[i] = 1;
      while(!vis[pos]) pos++;
      if(num == n-1) ans = min(ans,e[i].w-e[pos].w);
    }
    else{
      st.makeRoot(e[i].a);
      int p1 = st.access(e[i].b);
      p1 = st.mpos[p1];
      st.cut(e[p1-n].a,p1); st.cut(p1,e[p1-n].b); vis[p1-n] = 0;
      st.link(e[i].a,i+n); st.link(i+n,e[i].b); vis[i] = 1;
      while(!vis[pos]) pos++;
      if(num == n-1) ans = min(ans,e[i].w-e[pos].w);
    }
  }
  printf("%d\n",ans);
  return 0;
}
```

4.15.5 维护实边与虚边

题意:
n 个点，一共 q 次操作。一共有两种操作类型，$A\ x\ y$ 表示连通 (x,y)，保证操作合法，且始终是棵森林。$Q\ x\ y$ 表示查询去除 (x,y) 边之后，x 所在树的节点数 $*\ y$ 所在树的节点数。$(1 \le n, q \le 10^5)$

思路:
我们一般遇到的都是维护链上节点个数的问题，而此题要求这颗树上的节点个数，因此我们需要同时维护虚边和实边的信息。

我们令 $sz[x]$ 表示节点 x 子树中节点个数，$sz2[x]$ 表示节点 x 虚儿子的节点个数和。因此 $sz[x] = sz[ls]+sz[rs]+1+sz2[x]$，而这也正是 $pushUp$ 函数。

因此我们只需要维护 $sz2[x]$ 即可，然后观察哪些函数会改变 $sz2[x]$ 的值，不难发现，只有 $makeRoot$、$access$、$link$、cut 会改变边的虚实关系，其中 $makeRoot$ 主要修改在于调用了 $access$ 函数，而 cut 只是删除实边不会修改虚边，因此真正关键的函数即为 $link$ 和 $access$ 函数，具体操作见代码，不难思考。

这里主要讲解 $link$ 中修改 $sz2[y]$ 信息时为什么需要将节点 y $makeRoot(y)$，原因在于单点修改之后，其祖先节点维护的信息都会发生变化，因此一般的问题需要 $splay(y)$，因为一般问题只需要维护实链信息。然后在该问题中还维护了虚链信息，因此需要 $makeRoot(y)$ 而不是 $splay(y)$。

```
#include <bits/stdc++.h>
#define rep(i,a,b) for(int i = a; i <= b; i++)
using namespace std;
typedef long long ll;
const int N = 100010;
int n, q;

struct LCT{
  #define ls ch[p][0]
```

```
#define rs ch[p][1]
#define Get(p) (ch[f[p]][1] == p)
int ch[N][2], f[N], siz[N], siz2[N], rev[N];

inline void clear(int p){ //清除这个点的信息
  ch[p][0] = ch[p][1] = f[p] = siz[p] = siz2[p] = rev[p] = 0;
}

inline int isRoot(int p){
  clear(0);
  return ch[f[p]][0] != p && ch[f[p]][1] != p;
}

inline void pushUp(int p){
  clear(0);
  siz[p] = siz[ls] + 1 + siz[rs] + siz2[p];
}

inline void pushDown(int p){
  clear(0);
  if(rev[p]){
    if(ls) rev[ls] ^= 1, swap(ch[ls][0],ch[ls][1]);
    if(rs) rev[rs] ^= 1, swap(ch[rs][0],ch[rs][1]);
    rev[p] = 0;
  }
}

void update(int p){ //递归地从上到下pushDown信息
  if(!isRoot(p)) update(f[p]);
  pushDown(p);
}

inline void rotate(int x){ //将x向上旋转一层的操作
  int y = f[x], z = f[y], k = Get(x);
  if(!isRoot(y)) ch[z][ch[z][1] == y] = x;
  ch[y][k] = ch[x][!k], f[ch[y][k]] = y;
  ch[x][!k] = y, f[y] = x, f[x] = z;
  pushUp(y); //要先pushUp(y)
  pushUp(x);
}

inline void splay(int x){ //把x旋转到当前splay的根
  update(x); //将上面的标记完全下放
  for(int fa; fa = f[x], !isRoot(x); rotate(x)){
    if(!isRoot(fa)) rotate(Get(fa) == Get(x) ? fa : x);
  }
}

inline int access(int x){ //把从根到x的所有点放在一条实链里，返回这个splay的根
  int p; //每次改变右儿子的值，因为整棵树是中序遍历，放入右儿子才能保证先遍历父亲再遍历儿子
  for(p = 0; x; p = x, x = f[x]){
    splay(x), siz2[x] += siz[ch[x][1]]-siz[p], ch[x][1] = p, pushUp(x);
  }
  return p;
}

inline void makeRoot(int p){ //使x点成为整棵树的根
  access(p); splay(p);
  swap(ch[p][0],ch[p][1]); //把整条链反向
```

```
    rev[p] ^= 1;
  }

  inline void link(int x,int y){ //在x、y两点间连一条边
    //makeRoot(x)的作用是使得x无父亲
    //makeRoot(y)的作用是使得y无父亲，因此可以修改y的信息，不用去更新y的祖先
    makeRoot(x), makeRoot(y), f[x] = y, siz2[y] += siz[x]; pushUp(y);
  }

  inline void cut(int x,int p){ //把x、y两点间边删掉，此处删除的是实边，注意实边和虚边的区别
    makeRoot(x), access(p), splay(p);
    if (ls == x && !rs) ls = f[x] = 0;
  }

  inline int find(int p){ //找到x所在树的根节点编号
    access(p), splay(p);
    while(ls) pushDown(p), p = ls;
    return p;
  }
}st;

signed main() {
  scanf("%d%d", &n, &q);
  rep(i,1,n) st.siz[i] = 1;
  while (q--) {
    char op[10]; int x,y; scanf("%s%d%d",op,&x,&y);
    if(op[0] == 'A') st.link(x,y);
    else{
      st.cut(x,y);
      st.makeRoot(x); st.splay(x);
      int a1 = st.siz[x];
      st.makeRoot(y); st.splay(y);
      int a2 = st.siz[y];
      st.link(x,y);
      printf("%lld\n",(ll)a1*(ll)a2);
    }
  }
  return 0;
}
```

4.15.6 TopTree

题意:
支持 12 种操作，包括链 max、min、sum，子树 max、min、sum，换根，换边，子树和链的修改与加值。$(1 \le n, m \le 10^5)$

思路:
$TopTree$ 典型例题，主要思路是对于每个点维护了一个 $splay$，详情看 [$claris$ 的题解]

```
/*
Toptree即为可以维护子树信息的lct升级版
*/
#include<cstdio>
#define N 200010
const int inf=~0U>>1;
inline void swap(int&a,int&b){int c=a;a=b;b=c;}
inline int max(int a,int b){return a>b?a:b;}
inline int min(int a,int b){return a<b?a:b;}
```

```
inline void read(int&a){
  char c;bool f=0;a=0;
  while(!((((c=getchar())>='0')&&(c<='9'))||(c=='-')));
  if(c!='-')a=c-'0';else f=1;
  while(((c=getchar())>='0')&&(c<='9'))(a*=10)+=c-'0';
  if(f)a=-a;
}
struct tag{
  int a,b;//ax+b
  tag(){a=1,b=0;}
  tag(int x,int y){a=x,b=y;}
  inline bool ex(){return a!=1||b;}
  inline tag operator+(const tag&x){return tag(a*x.a,b*x.a+x.b);}
};
inline int atag(int x,tag y){return x*y.a+y.b;}
struct data{
  int sum,minv,maxv,size;
  data(){sum=size=0,minv=inf,maxv=-inf;}
  data(int x){sum=minv=maxv=x,size=1;}
  data(int a,int b,int c,int d){sum=a,minv=b,maxv=c,size=d;}
  inline data operator+(const data&x){return data(sum+x.sum,min(minv,x.minv),max(maxv,x
    .maxv),size+x.size);}
};
inline data operator+(const data&a,const tag&b){return a.size?data(a.sum*b.a+a.size*b.b
    ,atag(a.minv,b),atag(a.maxv,b),a.size):a;}
//son:0-1: 重链儿子, 2-3: AAA树儿子
int f[N],son[N][4],a[N],tot,rt,rub,ru[N];bool rev[N],in[N];
int val[N];
data csum[N],tsum[N],asum[N];
tag ctag[N],ttag[N];
inline bool isroot(int x,int t){
  if(t)return !f[x]||!in[f[x]]||!in[x];
  return !f[x]||(son[f[x]][0]!=x&&son[f[x]][1]!=x)||in[f[x]]||in[x];
}
inline void rev1(int x){
  if(!x)return;
  swap(son[x][0],son[x][1]);rev[x]^=1;
}
inline void tagchain(int x,tag p){
  if(!x)return;
  csum[x]=csum[x]+p;
  asum[x]=csum[x]+tsum[x];
  val[x]=atag(val[x],p);
  ctag[x]=ctag[x]+p;
}
inline void tagtree(int x,tag p,bool t){
  if(!x)return;
  tsum[x]=tsum[x]+p;
  ttag[x]=ttag[x]+p;
  if(!in[x]&&t)tagchain(x,p);else asum[x]=csum[x]+tsum[x];
}
inline void pb(int x){
  if(!x)return;
  if(rev[x])rev1(son[x][0]),rev1(son[x][1]),rev[x]=0;
  if(!in[x]&&ctag[x].ex())tagchain(son[x][0],ctag[x]),tagchain(son[x][1],ctag[x]),ctag[
    x]=tag();
  if(ttag[x].ex()){
    tagtree(son[x][0],ttag[x],0),tagtree(son[x][1],ttag[x],0);
    tagtree(son[x][2],ttag[x],1),tagtree(son[x][3],ttag[x],1);
```

```
    ttag[x]=tag();
  }
}
inline void up(int x){
  tsum[x]=data();
  for(int i=0;i<2;i++)if(son[x][i])tsum[x]=tsum[x]+tsum[son[x][i]];
  for(int i=2;i<4;i++)if(son[x][i])tsum[x]=tsum[x]+asum[son[x][i]];
  if(in[x]){
    csum[x]=data();
    asum[x]=tsum[x];
  }else{
    csum[x]=data(val[x]);
    for(int i=0;i<2;i++)if(son[x][i])csum[x]=csum[x]+csum[son[x][i]];
    asum[x]=csum[x]+tsum[x];
  }
}
inline int child(int x,int t){pb(son[x][t]);return son[x][t];}
inline void rotate(int x,int t){
  int y=f[x],w=(son[y][t+1]==x)+t;
  son[y][w]=son[x][w^1];
  if(son[x][w^1])f[son[x][w^1]]=y;
  if(f[y])for(int z=f[y],i=0;i<4;i++)if(son[z][i]==y)son[z][i]=x;
  f[x]=f[y];f[y]=x;son[x][w^1]=y;up(y);
}
inline void splay(int x,int t=0){
  int s=1,i=x,y;a[1]=i;
  while(!isroot(i,t))a[++s]=i=f[i];
  while(s)pb(a[s--]);
  while(!isroot(x,t)){
    y=f[x];
    if(!isroot(y,t)){if((son[f[y]][t]==y)^(son[y][t]==x))rotate(x,t);else rotate(y,t);}
    rotate(x,t);
  }
  up(x);
}
inline int newnode(){
  int x=rub?ru[rub--]:++tot;
  son[x][2]=son[x][3]=0;in[x]=1;
  return x;
}
inline void setson(int x,int t,int y){son[x][t]=y;f[y]=x;}
inline int pos(int x){for(int i=0;i<4;i++)if(son[f[x]][i]==x)return i;return 4;}
inline void add(int x,int y){//从x连出一条虚边到y
  if(!y)return;
  pb(x);
  for(int i=2;i<4;i++)if(!son[x][i]){
    setson(x,i,y);
    return;
  }
  while(son[x][2]&&in[son[x][2]])x=child(x,2);
  int z=newnode();
  setson(z,2,son[x][2]);
  setson(z,3,y);
  setson(x,2,z);
  splay(z,2);
}
inline void del(int x){//将x与其虚边上的父亲断开
  if(!x)return;
  splay(x);
```

```
  if(!f[x])return;
  int y=f[x];
  if(in[y]){
    int s=1,i=y,z=f[y];a[1]=i;
    while(!isroot(i,2))a[++s]=i=f[i];
    while(s)pb(a[s--]);
    if(z){
      setson(z,pos(y),child(y,pos(x)^1));
      splay(z,2);
    }
    ru[++rub]=y;
  }else{
    son[y][pos(x)]=0;
    splay(y);
  }
  f[x]=0;
}
inline int fa(int x){//x通过虚边的父亲
  splay(x);
  if(!f[x])return 0;
  if(!in[f[x]])return f[x];
  int t=f[x];
  splay(t,2);
  return f[t];
}
inline int access(int x){
  int y=0;
  for(;x;y=x,x=fa(x)){
    splay(x);
    del(y);
    add(x,son[x][1]);
    setson(x,1,y);
    up(x);
  }
  return y;
}
inline int lca(int x,int y){
  access(x);
  return access(y);
}
inline int root(int x){
  access(x);
  splay(x);
  while(son[x][0])x=son[x][0];
  return x;
}
inline void makeroot(int x){
  access(x);
  splay(x);
  rev1(x);
}
inline void link(int x,int y){
  makeroot(x);
  add(y,x);
  access(x);
}
inline void cut(int x){
  access(x);
  splay(x);
```

```
  f[son[x][0]]=0;
  son[x][0]=0;
  up(x);
}
inline void changechain(int x,int y,tag p){
  makeroot(x);
  access(y);
  splay(y);
  tagchain(y,p);
}
inline data askchain(int x,int y){
  makeroot(x);
  access(y);
  splay(y);
  return csum[y];
}
inline void changetree(int x,tag p){
  access(x);
  splay(x);
  val[x]=atag(val[x],p);
  for(int i=2;i<4;i++)if(son[x][i])tagtree(son[x][i],p,1);
  up(x);
  splay(x);
}
inline data asktree(int x){
  access(x);
  splay(x);
  data t=data(val[x]);
  for(int i=2;i<4;i++)if(son[x][i])t=t+asum[son[x][i]];
  return t;
}
int n,m,x,y,z,k,i,ed[N][2];
int main(){
  read(n);read(m);
  tot=n;
  for(i=1;i<n;i++)read(ed[i][0]),read(ed[i][1]); //连边
  for(i=1;i<=n;i++)read(val[i]),up(i); //先赋点权，再连边
  for(i=1;i<n;i++)link(ed[i][0],ed[i][1]); //每个点的权值
  read(rt); //给出根
  makeroot(rt);
  while(m--){
    read(k);
    if(k==1){//换根，x变成根
      read(rt);
      makeroot(rt);
    }
    if(k==9){//x的父亲变成y，x父亲换成y
      read(x),read(y);
      if(lca(x,y)==x)continue;
      cut(x);
      link(y,x);
      makeroot(rt);
    }
    if(k==0){//子树赋值，以x为根的子树点权值改为y
      read(x),read(y);
      changetree(x,tag(0,y));
    }
    if(k==5){//子树加，x为根子树点权值加上y
      read(x),read(y);
```

```
      changetree(x,tag(1,y));
    }
    if(k==3){//子树最小值，x为根子树中点权值求min
      read(x);
      printf("%d\n",asktree(x).minv);
    }
    if(k==4){//子树最大值，x为根子树中点权值求max
      read(x);
      printf("%d\n",asktree(x).maxv);
    }
    if(k==11){//子树和，x为根子树中点权sum
      read(x);
      printf("%d\n",asktree(x).sum);
    }
    if(k==2){//链赋值，x-y路径上点权值改为z
      read(x),read(y),read(z);
      changechain(x,y,tag(0,z));
      makeroot(rt);
    }
    if(k==6){//链加，x-y路径上点权值加上z
      read(x),read(y),read(z);
      changechain(x,y,tag(1,z));
      makeroot(rt);
    }
    if(k==7){//链最小值，x-y路径上点权值求min
      read(x),read(y);
      printf("%d\n",askchain(x,y).minv);
      makeroot(rt);
    }
    if(k==8){//链最大值，x-y路径上点权值求max
      read(x),read(y);
      printf("%d\n",askchain(x,y).maxv);
      makeroot(rt);
    }
    if(k==10){//链和，x-y路径上点权值求sum
      read(x),read(y);
      printf("%d\n",askchain(x,y).sum);
      makeroot(rt);
    }
  }
  return 0;
}
```

4.16 树链剖分

4.16.1 点剖模板

一颗 N 个节点的树，每个节点上都有初始权值。现在有四种操作：操作 1 ——1 x y z，表示将 x 到 y 节点最短路径上所有节点的值加 z 操作 2 ——2 x y，表示求 x 到 y 节点最短路径上所有节点值之和操作 3 ——3 x z，表示将以 x 为根节点的子树内所有节点值加 z 操作 4 ——4 x，表示求以 x 为根节点的子树内所有节点值之和

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
```

```cpp
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl;
typedef long long ll;
typedef double db;
#define int long long
const db EPS = 1e-9;
const int N = 1e5+100;
using namespace std;

struct Edge { int next,to;} e[2*N];
struct Node { int l,r,ls,rs,sum,lazy;} t[2*N];
int n,m,root,rt,mod,val[N],head[N],tot,fa[N],d[N],son[N],size[N],top[N],id[N],rk[N];
//top[x]: x节点所在链的顶端节点, id[x]: 节点dfs序, rk[x]: dfs序对应的节点
//val[x]: 每个点初始权值, fa[x]: 每个点父节点, d[x]: 节点深度, size[x]: 节点子树大小
//rt: 线段树根节点编号
void init(){
    memset(head,0,sizeof head);
    tot = 1, size[0] = 0;
}

void add(int x, int y){
    e[++tot].next = head[x], e[tot].to = y, head[x] = tot;
}

void dfs1(int x){   //求出每个点的子树大小、深度、重儿子
    size[x] = 1, d[x] = d[fa[x]]+1, son[x] = 0;
    for(int v,i = head[x]; i; i = e[i].next)
        if((v = e[i].to)!=fa[x]){
            fa[v] = x, dfs1(v), size[x] += size[v];
            if(size[son[x]] < size[v])
                son[x] = v;
        }
}

void dfs2(int x, int tp){   //求出每个节点的dfs序, dfs序对应的节点, 以及每个点所在链的顶端节点
    top[x] = tp, id[x] = ++tot, rk[tot] = x;
    if(son[x]) dfs2(son[x],tp);
    for(int v,i = head[x]; i; i = e[i].next)
        if((v = e[i].to)!=fa[x] && v!=son[x]) dfs2(v,v);
}

inline void pushup(int x){ //基础的线段树向上区间合并
    t[x].sum = (t[t[x].ls].sum+t[t[x].rs].sum)%mod;  //此题需要将sum和对mod取模
}

void build(int l, int r, int x){ //基础建树, 动态开点
    t[x].l = l, t[x].r = r, t[x].lazy = 0;
    if(l == r){
        t[x].sum = val[rk[l]]; return;
    }
    int mid = (l+r)>>1;
    t[x].ls = ++tot, t[x].rs = ++tot;
    build(l,mid,t[x].ls), build(mid+1,r,t[x].rs), pushup(x);
}

inline int len(int x) { return t[x].r-t[x].l+1; }

inline void pushdown(int x){ //基础的线段树标记下放
    if(t[x].lazy && t[x].l != t[x].r){
```

```
        int ls = t[x].ls, rs = t[x].rs, lz = t[x].lazy;
        (t[ls].lazy+=lz) %= mod, (t[rs].lazy+=lz) %= mod;
        (t[ls].sum+=lz*len(ls)) %= mod, (t[rs].sum+=lz*len(rs)) %= mod;
        t[x].lazy = 0;
    }
}

void update(int l, int r, int x, int c){ //基础的线段树更新
    if(t[x].l >= l && t[x].r <= r){
        (t[x].lazy += c) %= mod, (t[x].sum += len(x)*c) %= mod;
        return;
    }
    pushdown(x);
    int mid = (t[x].l+t[x].r)>>1;
    if(mid >= l) update(l,r,t[x].ls,c);
    if(mid < r) update(l,r,t[x].rs,c);
    pushup(x);
}

int query(int l, int r, int x){  //基础的线段树查询
    if(t[x].l >= l && t[x].r <= r) return t[x].sum;
    pushdown(x);
    int mid = (t[x].l+t[x].r)>>1, tp = 0;
    if(mid >= l) tp += query(l,r,t[x].ls);
    if(mid < r) tp += query(l,r,t[x].rs);
    return tp%mod;
}

inline int sum(int x, int y){   //将区间分为多条链，对于每条链直接查询
    int ret = 0;
    while(top[x] != top[y]){ //让x与y到达同一条链
        if(d[top[x]] < d[top[y]]) swap(x,y); //找到更深的点
        (ret += query(id[top[x]],id[x],rt)) %= mod;
        x = fa[top[x]];
    }
    if(id[x] > id[y]) swap(x,y);
    return (ret+query(id[x],id[y],rt))%mod;
}

inline void updates(int x, int y, int c){   //区间加z，将区间分为多条链
    while(top[x] != top[y]){
        if(d[top[x]] < d[top[y]]) swap(x,y);
        update(id[top[x]],id[x],rt,c);  //对于每条链直接修改
        x = fa[top[x]];
    }
    if(id[x] > id[y]) swap(x,y);
    update(id[x],id[y],rt,c);
}

signed main()
{
    scanf("%lld%lld%lld%lld",&n,&m,&root,&mod);
    rep(i,1,n) scanf("%lld",&val[i]);
    init();
    rep(i,1,n-1){
        int x,y; scanf("%lld%lld",&x,&y);
        add(x,y), add(y,x);
    }
    tot = 0, dfs1(root), dfs2(root, root);
```

```
    tot = 0, build(1,n,rt = ++tot);
    rep(i,1,m){
        int op,x,y,k; scanf("%lld",&op);
        if(op == 1){
            scanf("%lld%lld%lld",&x,&y,&k);
            updates(x,y,k);
        }
        else if(op == 2){
            scanf("%lld%lld",&x,&y);
            printf("%lld\n",sum(x,y));
        }
        else if(op == 3){
            scanf("%lld%lld",&x,&y);
            update(id[x],id[x]+size[x]-1,rt,y);
        }
        else{
            scanf("%lld",&x);
            printf("%lld\n",query(id[x],id[x]+size[x]-1,rt));
        }
    }
    return 0;
}
```

4.16.2 树上路径颜色段数量

题意:
一颗 n 个节点的树，两个操作:
将 $a->b$ 路径上的点都染成颜色 c
查询 $a->b$ 路径上的颜色段数量

思路:
很明显是一个树剖问题，树剖的基础实现就不多说了，我们来考虑一下线段树需要维护什么。

首先求的是路径上不同颜色段数量，因此肯定需要维护一个 cnt，表示路径上不同颜色段的数量。然后我们来看如何实现区间合并，主要观察的就是左区间的右端点和右区间的左端点是否一样，如果一样，合并的 cnt 需要减 1。但是如何去快速查询左右节点颜色呢，因此可以想到再维护区间左右节点颜色即可完成本题。

树剖成多个树链时需要注意链条交界处的颜色是否一致。

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl
    ;
typedef long long ll;
typedef double db;
#define int long long
const db EPS = 1e-9;
const int N = 1e5+100;
using namespace std;

struct Edge { int next,to;} e[2*N];
struct Node { int l,r,ls,rs,cnt,lazy,lc,rc;} t[2*N]; //cnt: 不同颜色个数, lc: 左端颜色, rc
    : 右端颜色
```

```cpp
int n,m,root,rt,mod,val[N],head[N],tot,fa[N],d[N],son[N],size[N],top[N],id[N],rk[N];
//top[x]: x节点所在链的顶端节点, id[x]: 节点dfs序, rk[x]: dfs序对应的节点
//val[x]: 每个点初始权值, fa[x]: 每个点父节点, d[x]: 节点深度, size[x]: 节点子树大小
//rt: 线段树根节点编号
void init(){
    memset(head,0,sizeof head);
    tot = 1, size[0] = 0;
}

void add(int x, int y){
    e[++tot].next = head[x], e[tot].to = y, head[x] = tot;
}

void dfs1(int x){   //求出每个点的子树大小、深度、重儿子
    size[x] = 1, d[x] = d[fa[x]]+1, son[x] = 0;
    for(int v,i = head[x]; i; i = e[i].next)
        if((v = e[i].to)!=fa[x]){
            fa[v] = x, dfs1(v), size[x] += size[v];
            if(size[son[x]] < size[v])
                son[x] = v;
        }
}

void dfs2(int x, int tp){   //求出每个节点的dfs序, dfs序对应的节点, 以及每个点所在链的顶端节点
    top[x] = tp, id[x] = ++tot, rk[tot] = x;
    if(son[x]) dfs2(son[x],tp);
    for(int v,i = head[x]; i; i = e[i].next)
        if((v = e[i].to)!=fa[x] && v!=son[x]) dfs2(v,v);
}

inline void pushup(int x){ //基础的线段树向上区间合并
    t[x].lc = t[t[x].ls].lc, t[x].rc = t[t[x].rs].rc, t[x].cnt = t[t[x].ls].cnt + t[t[x].rs].cnt;
    if(t[t[x].ls].rc == t[t[x].rs].lc) t[x].cnt--;
}

void build(int l, int r, int x){ //基础建树, 动态开点
    t[x].l = l, t[x].r = r, t[x].lazy = -1;
    if(l == r){
        t[x].lc = t[x].rc = val[rk[l]], t[x].cnt = 1; return;
    }
    int mid = (l+r)>>1;
    t[x].ls = ++tot, t[x].rs = ++tot;
    build(l,mid,t[x].ls), build(mid+1,r,t[x].rs), pushup(x);
}

inline void pushdown(int x){ //基础的线段树标记下放
    if(t[x].lazy!=-1 && t[x].l != t[x].r){
        int ls = t[x].ls, rs = t[x].rs, lz = t[x].lazy;
        t[ls].lazy = lz, t[rs].lazy = lz;
        t[ls].lc = t[ls].rc = lz, t[rs].lc = t[rs].rc = lz;
        t[ls].cnt = t[rs].cnt = 1;
        t[x].lazy = -1;
    }
}

void update(int l, int r, int x, int c){ //基础的线段树更新
    if(t[x].l >= l && t[x].r <= r){
        t[x].lazy = c, t[x].cnt = 1, t[x].lc = t[x].rc = c;
```

```
        return;
    }
    pushdown(x);
    int mid = (t[x].l+t[x].r)>>1;
    if(mid >= l) update(l,r,t[x].ls,c);
    if(mid < r) update(l,r,t[x].rs,c);
    pushup(x);
}

int query(int l, int r, int x){  //基础的线段树查询
    // LOG2("l",l,"r",r);
    // LOG2("t[x].l",t[x].l,"t[x].r",t[x].r);
    if(t[x].l >= l && t[x].r <= r) return t[x].cnt;
    pushdown(x);
    int mid = (t[x].l+t[x].r)>>1, tp = 0;
    if(mid >= l && mid >= r) tp += query(l,r,t[x].ls);
    else if(mid < l && mid < r) tp += query(l,r,t[x].rs);
    else if(mid >= l && mid < r){
        tp += query(l,r,t[x].ls);
        tp += query(l,r,t[x].rs);
        if(t[t[x].ls].rc == t[t[x].rs].lc) tp--;
    }
    return tp;
}

int query_color(int pos, int x){  //查询单点颜色
    if(t[x].l == pos) return t[x].lc;
    if(t[x].r == pos) return t[x].rc;
    if(pos >= t[x].l && pos <= t[x].r && t[x].cnt == 1) return t[x].lc;
    int mid = (t[x].l+t[x].r)>>1;
    if(pos <= mid) return query_color(pos,t[x].ls);
    else return query_color(pos,t[x].rs);
}

inline int sum(int x, int y){   //将区间分为多条链，对于每条链直接查询
    int ret = 0;
    while(top[x] != top[y]){ //让x与y到达同一条链
        if(d[top[x]] < d[top[y]]) swap(x,y); //找到更深的点
        ret += query(id[top[x]],id[x],rt);
        // LOG1("ret",ret);
        if(query_color(id[top[x]],rt) == query_color(id[fa[top[x]]],rt)) ret--;
        x = fa[top[x]];
    }
    if(id[x] > id[y]) swap(x,y);
    return (ret+query(id[x],id[y],rt));
}

inline void updates(int x, int y, int c){   //区间加z，将区间分为多条链
    while(top[x] != top[y]){
        if(d[top[x]] < d[top[y]]) swap(x,y);
        update(id[top[x]],id[x],rt,c);  //对于每条链直接修改
        x = fa[top[x]];
    }
    if(id[x] > id[y]) swap(x,y);
    update(id[x],id[y],rt,c);
}

signed main()
{
```

```
    scanf("%lld%lld",&n,&m);
    rep(i,1,n) scanf("%lld",&val[i]);
    init();
    rep(i,1,n-1){
        int x,y; scanf("%lld%lld",&x,&y);
        add(x,y), add(y,x);
    }
    scanf("%lld",&m);
    root = 1;
    tot = 0, dfs1(root), dfs2(root, root);
    tot = 0, build(1,n,rt = ++tot);
    rep(i,1,m){
        char op[10]; scanf("%s",op);
        int x,y,z;
        if(op[0] == 'Q'){
            scanf("%lld%lld",&x,&y);
            printf("%lld\n",sum(x,y));
        }
        else{
            scanf("%lld%lld%lld",&x,&y,&z);
            updates(x,y,z);
        }
    }
    return 0;
}
```

4.16.3 动态开点树剖

题意:
现在有 n 个城市，构成了一颗树。每个城市都有自己信仰的宗教，以及城市评级。现在一共有四种操作:
某个城市改信 c 教
某个城市的评级调整为 w
$x->y$ 路径上所有与 x 信仰相同的城市的评级之和
$x->y$ 路径上所有与 x 信仰相同的城市的评级最大值

$(N, Q \leq 10^5\ C \leq 10^5)$

思路:
需要维护的操作只是单点修改和区间最值与最大值，维护的操作都不难。主要困难的地方在于最值和最大值都只在信仰相同的城市之间统计，因此我们需要对每个信仰都建一颗线段树。

但是由于空间的限制，对每个宗教都建一颗完整的线段树是不可能的，因此我们需要动态开点的操作。采用动态开点的原因是本题初始最多只有 10^5 个点，操作最多也只有 10^5 次，因此有效的点最多只有 $2*10^5$ 个，所以我们只需要维护这些有效点即可。

我们再来仔细讲一下动态开点的原理。如下图所示，现在只有点 A 是有效点，因此我们只需给 $root->A$ 路径上的点分配空间，不需要给其他的点分配空间，因此就达到了简化空间的目的。所以动态开一个点的空间费用最多是 $logn$，我们只需要给没颗树记一个根节点，以及每个节点对应的左右儿子编号即可。因此我们也不需要之前建树的 $build$ 函数了，用 $update$ 函数动态插入每个点即可。

回到这道题来，解决的思路就很简单了。给每个宗教建一颗线段树，维护每颗线段树的根节点编号，然后对于每个线段树进行查询和修改即可，一个点从信仰 a 变为信仰 b，只需在 a 线段树中将这个点赋为 0，然后在 b 线段树将这个点再赋值即可。

```
#include <cstdio>
#include <iostream>
#include <cstring>
```

```
#include <algorithm>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl;
typedef long long ll;
typedef double db;
const db EPS = 1e-9;
using namespace std;
const int N = 2*1e5+1000;

int ls[20*N], rs[20*N], val[20*N], maxn[20*N], sz, rt[N];
int n,m,head[N],tot,son[N],size[N],d[N],id[N],rk[N],fa[N],top[N],ans1,ans2;
int re[N],lev[N];
struct Edge{
    int to,next;
}e[N];

void init() { tot = 1, memset(head,0,sizeof head); }
void add(int x, int y) { e[++tot].to = y, e[tot].next = head[x], head[x] = tot; }
void dfs1(int x){   //求出每个点的子树大小、深度、重儿子
    size[x] = 1, d[x] = d[fa[x]]+1, son[x] = 0;
    for(int v,i = head[x]; i; i = e[i].next)
        if((v = e[i].to)!=fa[x]){
            fa[v] = x, dfs1(v), size[x] += size[v];
            if(size[son[x]] < size[v])
                son[x] = v;
        }
}
void dfs2(int x, int tp){   //求出每个节点的dfs序, dfs序对应的节点, 以及每个点所在链的顶端节点
    top[x] = tp, id[x] = ++tot, rk[tot] = x;
    if(son[x]) dfs2(son[x],tp);
    for(int v,i = head[x]; i; i = e[i].next)
        if((v = e[i].to)!=fa[x] && v!=son[x]) dfs2(v,v);
}
void update(int& now, int l, int r, int x, int c){  //单点修改
    if(!now) now = ++sz;
    if(l == r){
        val[now] = c, maxn[now] = c;
        return;
    }
    int mid = (l+r)>>1;
    if(x <= mid) update(ls[now],l,mid,x,c);
    if(x > mid) update(rs[now],mid+1,r,x,c);
    val[now] = val[ls[now]]+val[rs[now]];
    maxn[now] = max(maxn[ls[now]],maxn[rs[now]]);
}
void query(int now, int l, int r, int xx, int yy){
    if(l >= xx && r <= yy){
        ans1 += val[now], ans2 = max(ans2,maxn[now]);
        return;
    }
    int mid = (l+r)>>1;
    if(xx <= mid) query(ls[now],l,mid,xx,yy);
    if(yy > mid) query(rs[now],mid+1,r,xx,yy);
}
inline void updates(int x, int y){  //区间加z, 将区间分为多条链
    int tp = re[x];
```

```
    while(top[x] != top[y]){
        if(d[top[x]] < d[top[y]]) swap(x,y);
        query(rt[tp],1,n,id[top[x]],id[x]); //对于每条链直接修改
        x = fa[top[x]];
    }
    if(id[x] > id[y]) swap(x,y);
    query(rt[tp],1,n,id[x],id[y]);
}
int main()
{
    scanf("%d%d",&n,&m);
    rep(i,1,n) scanf("%d%d",&lev[i],&re[i]);
    init();
    rep(i,1,n-1){
        int xx,yy; scanf("%d%d",&xx,&yy);
        add(xx,yy), add(yy,xx);
    }
    tot = sz = 0, dfs1(1), dfs2(1,1);
    rep(i,1,n) update(rt[re[i]],1,n,id[i],lev[i]);
    rep(i,1,m){
        char s[10]; int x,y;
        scanf("%s",s);
        if(s[1] == 'C'){
            scanf("%d%d",&x,&y);
            update(rt[re[x]],1,n,id[x],0);
            update(rt[y],1,n,id[x],lev[x]);
            re[x] = y;
        }
        else if(s[1] == 'W'){
            scanf("%d%d",&x,&y);
            update(rt[re[x]],1,n,id[x],y);
            lev[x] = y;
        }
        else{
            scanf("%d%d",&x,&y);
            ans1 = ans2 = 0;
            updates(x,y);
            if(s[1] == 'S') printf("%d\n",ans1);
            else printf("%d\n",ans2);
        }
    }
    return 0;
}
```

4.16.4 树剖换根

题意:
n 个点的树，每个顶点都有一个值，需要进行三种操作:
将树根修改为 x
将 $x->y$ 路径上所有点的值修改为 v
询问在当前树根状态下，以节点 x 为根的子树中的最小值
需要注意，树根修改了之后，节点 x 对应的子树也就变化了。

思路:
我们来分类讨论，枚举一下情况。我们需要求节点 x 在当前树根下的子树中的最小值。
如果当前树根就是 x，那很明显答案就是整棵树的节点最小值。
如果当前树根在 x 的子树中，比如 $A1$，那么我们可以发现答案就是去除 $A1$ 在 x 中的这一段子树部分，剩余部分中找最小值即可。

如果当前树根在 x 子树之外，比如 $A2$，那么答案就是在 x 原来对应的子树中找一个最小值即可。

所以这题就变成了对于 x 与当前树根求一个 LCA 的问题，然后只有在第二种情况中才需要进行特殊处理，即在 x 的子树找到对应 $A1$ 的那一个儿子即可。当然在树剖中，利用 top 数组就可以完成 LCA 能够完成的功能，见如下代码。

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl
    ;
typedef long long ll;
typedef double db;
#define int long long
const db EPS = 1e-9;
const int N = 1e5+100;
using namespace std;

struct Edge { int next,to;} e[2*N];
struct Node { int l,r,ls,rs,lazy,minn;} t[2*N];
int n,m,root,rt,mod,val[N],head[N],tot,fa[N],d[N],son[N],size[N],top[N],id[N],rk[N];
//top[x]: x节点所在链的顶端节点, id[x]: 节点dfs序, rk[x]: dfs序对应的节点
//val[x]: 每个点初始权值, fa[x]: 每个点父节点, d[x]: 节点深度, size[x]: 节点子树大小
//rt: 线段树根节点编号
void init() { memset(head,0,sizeof head); tot = 1, size[0] = 0; }
void add(int x, int y) { e[++tot].next = head[x], e[tot].to = y, head[x] = tot; }
void dfs1(int x){   //求出每个点的子树大小、深度、重儿子
    size[x] = 1, d[x] = d[fa[x]]+1, son[x] = 0;
    for(int v,i = head[x]; i; i = e[i].next)
        if((v = e[i].to)!=fa[x]){
            fa[v] = x, dfs1(v), size[x] += size[v];
            if(size[son[x]] < size[v])
                son[x] = v;
        }
}
void dfs2(int x, int tp){   //求出每个节点的dfs序, dfs序对应的节点, 以及每个点所在链的顶端节点
    top[x] = tp, id[x] = ++tot, rk[tot] = x;
    if(son[x]) dfs2(son[x],tp);
    for(int v,i = head[x]; i; i = e[i].next)
        if((v = e[i].to)!=fa[x] && v!=son[x]) dfs2(v,v);
}
inline void pushup(int x) { t[x].minn = min(t[t[x].ls].minn,t[t[x].rs].minn); }
void build(int l, int r, int x){ //基础建树, 动态开点
    t[x].l = l, t[x].r = r, t[x].lazy = 0;
    if(l == r){
        t[x].minn = val[rk[l]]; return;
    }
    int mid = (l+r)>>1;
    t[x].ls = ++tot, t[x].rs = ++tot;
    build(l,mid,t[x].ls), build(mid+1,r,t[x].rs), pushup(x);
}
inline int len(int x) { return t[x].r-t[x].l+1; }
inline void pushdown(int x){ //基础的线段树标记下放
    if(t[x].lazy && t[x].l != t[x].r){
        int ls = t[x].ls, rs = t[x].rs, lz = t[x].lazy;
```

```
        (t[ls].lazy=lz), (t[rs].lazy=lz);
        t[ls].minn = lz, t[rs].minn = lz;
        t[x].lazy = 0;
    }
}
void update(int l, int r, int x, int c){ //基础的线段树更新
    if(t[x].l >= l && t[x].r <= r){
        t[x].lazy = c, t[x].minn = c;
        return;
    }
    pushdown(x);
    int mid = (t[x].l+t[x].r)>>1;
    if(mid >= l) update(l,r,t[x].ls,c);
    if(mid < r) update(l,r,t[x].rs,c);
    pushup(x);
}
int query(int l, int r, int x){  //基础的线段树查询
    if(t[x].l >= l && t[x].r <= r) return t[x].minn;
    pushdown(x);
    int mid = (t[x].l+t[x].r)>>1, tp = 1e15;
    if(mid >= l) tp = min(tp,query(l,r,t[x].ls));
    if(mid < r) tp = min(tp,query(l,r,t[x].rs));
    return tp;
}
inline void updates(int x, int y, int c){   //区间加z, 将区间分为多条链
    while(top[x] != top[y]){
        if(d[top[x]] < d[top[y]]) swap(x,y);
        update(id[top[x]],id[x],rt,c);  //对于每条链直接修改
        x = fa[top[x]];
    }
    if(id[x] > id[y]) swap(x,y);
    update(id[x],id[y],rt,c);
}
int find(int x, int y){ // top数组求LCA
    int base = x;
    while(top[x] != top[y]){
        if(fa[top[y]] == base) return top[y]; //返回子树中存在当前树根的子儿子
        if(d[top[x]] < d[top[y]]) y = fa[top[y]];
        else x = fa[top[x]];
    }
    if(d[x] > d[y]) x = y;
    if(x == base) return son[x]; //返回子树中存在当前树根的子儿子
    else return 0; //当前树根在x子树之外
}
signed main()
{
    scanf("%lld%lld",&n,&m);
    init();
    rep(i,1,n-1){
        int x,y; scanf("%lld%lld",&x,&y);
        add(x,y), add(y,x);
    }
    rep(i,1,n) scanf("%lld",&val[i]);
    scanf("%lld",&root);
    tot = 0, dfs1(root), dfs2(root, root);
    tot = 0, build(1,n,rt = ++tot);
    rep(i,1,m){
        int op,x,y,z;
        scanf("%lld",&op);
```

```
        if(op == 1){
            scanf("%lld",&x);
            root = x;
        }
        else if(op == 2){
            scanf("%lld%lld%lld",&x,&y,&z);
            updates(x,y,z);
        }
        else{
            scanf("%lld",&x); int thp;
            if(root == x){
                printf("%lld\n",query(1,n,rt));
            }
            else if((thp = find(x,root))){
                printf("%lld\n",min(query(1,id[thp]-1,rt),query(id[thp]+size[thp],n,rt)
    ));
            }
            else printf("%lld\n",query(id[x],id[x]+size[x]-1,rt));
        }
    }
    return 0;
}
```

4.16.5 边剖 + 主席树

题意:
给定一棵 n 个节点的树，每条边都有一个权值，m 次查询，每次询问树上两点路径上边权小于 k 的边有多少条? $(1 \le n, m \le 10^5)$

思路:
比较裸的题目，可以离线操作，然后将询问值从小到大进行排序，然后每次单点修改，将比当前询问小的边加入树中。

此处需要注意是边权树剖，因此将每条边的权值压到深度更深的节点上，然后树剖路径查询时，最后两点处于同一条链时，将深度最浅的点改为该点的儿子即可。

比赛时没有考虑到将询问值排序，然后依次修改边权，因此采用了树上主席树的做法。先将所有的边都插入树中，然后直接查询区间中小于 k 的节点有多少个即可。

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl
    ;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2
    << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const db EPS = 1e-9;
const int N = 1e5+100;
using namespace std;

struct Edge { int next,to,w;} e[2*N];
struct Node { int l,r,ls,rs,sum;} t[20*N];
```

```cpp
int n,m,rt,mod,val[N],head[N],tot,fa[N],d[N],son[N],size[N],top[N],id[N],rk[N],root[N],
    rot;
//top[x]: x节点所在链的顶端节点, id[x]: 节点dfs序, rk[x]: dfs序对应的节点
//val[x]: 每个点初始权值, fa[x]: 每个点父节点, d[x]: 节点深度, size[x]: 节点子树大小
//rt: 线段树根节点编号
void init(){
    memset(head,0,sizeof head);
    tot = 1, size[0] = 0;
}

void add(int x, int y, int z){
    e[++tot].next = head[x], e[tot].to = y, head[x] = tot, e[tot].w = z;
}

void dfs1(int x){   //求出每个点的子树大小、深度、重儿子
    size[x] = 1, d[x] = d[fa[x]]+1, son[x] = 0;
    for(int v,i = head[x]; i; i = e[i].next)
        if((v = e[i].to)!=fa[x]){
            fa[v] = x, dfs1(v), size[x] += size[v];
            if(size[son[x]] < size[v])
                son[x] = v;
        }
}

void dfs2(int x, int tp){   //求出每个节点的dfs序, dfs序对应的节点, 以及每个点所在链的顶端节点
    top[x] = tp, id[x] = ++tot, rk[tot] = x;
    if(son[x]) dfs2(son[x],tp);
    for(int v,i = head[x]; i; i = e[i].next)
        if((v = e[i].to)!=fa[x] && v!=son[x]) dfs2(v,v);
}

inline void pushup(int x){ //基础的线段树向上区间合并
    t[x].sum = t[t[x].ls].sum+t[t[x].rs].sum;
}

int build(int l,int r) //主席树建树部分
{
    int p = ++tot; // 新建一个节点, 编号为p, 代表当前区间[l,r]
    t[p].l = l, t[p].r = r, t[p].sum = 0;
    if(l == r) return p;
    int mid = (l+r)>>1;
    t[p].ls = build(l,mid);
    t[p].rs = build(mid+1,r);
    return p;
}

inline int len(int x) { return t[x].r-t[x].l+1; }

int ask(int lp,int rp,int k){ //主席树查询[lp,rp]比k小的有多少个
    if(t[lp].l == t[lp].r) return (t[rp].sum-t[lp].sum);
    int mid = (t[lp].l+t[lp].r)>>1;
    int tp = 0;
    if(mid < k){
        tp += (t[t[rp].ls].sum-t[t[lp].ls].sum);
        tp += ask(t[lp].rs,t[rp].rs,k);
    }
    else tp += ask(t[lp].ls,t[rp].ls,k);
    return tp;
}
```

```

inline int solve(int x,int y,int k){
    int ret = 0;
    while(top[x] != top[y]){ //让x与y到达同一条链
        if(d[top[x]] < d[top[y]]) swap(x,y); //找到更深的点
        int xx = id[top[x]]-1, yy = id[x];
        int hp = ask(root[xx],root[yy],k);
        ret += hp;
        x = fa[top[x]];
    }
    if(x == y) return ret;
    if(id[x] > id[y]) swap(x,y);
    int xx = id[son[x]]-1, yy = id[y]; //边权树链剖分，最后两点同链之后，取最高点的儿子来计算
    //-1是因为主席树要获取-1时候的副本进行sum减运算
    int hp = ask(root[xx],root[yy],k);
    return (ret+hp);
}

int insert(int now,int pos,int k) //主席树插入一个新的值
{
    int p = ++tot;
    t[p] = t[now]; //建立副本
    if(t[p].l == t[p].r){
        t[p].sum += k; //在副本上修改
        return p;
    }
    int mid = (t[p].l+t[p].r)>>1;
    if(pos <= mid) t[p].ls = insert(t[p].ls,pos,k); //保留右儿子部分，把左儿子更新
    else t[p].rs = insert(t[p].rs,pos,k);
    t[p].sum = t[t[p].ls].sum + t[t[p].rs].sum;
    return p;
}

int ed[2*N][3];
int b[2*N],tb;

signed main()
{
    scanf("%d%d",&n,&m);
    init();
    rep(i,1,n-1){
        int x,y,z; scanf("%d %d %d",&x,&y,&z);
        add(x,y,z), add(y,x,z);
        ed[i][0] = x, ed[i][1] = y, ed[i][2] = z;
    }
    rot = 1;
    tot = 0, dfs1(rot), dfs2(rot, rot);
    rep(i,1,n-1){
        if(d[ed[i][0]] > d[ed[i][1]]) swap(ed[i][0],ed[i][1]);
        val[ed[i][1]] = ed[i][2];
        b[++tb] = ed[i][2];
    }
    sort(b+1,b+1+tb);
    tb = unique(b+1,b+1+tb)-b-1;
    tot = 0;
    root[0] = build(1,tb);
    rep(i,1,n){
        int x = lower_bound(b+1,b+1+tb,val[rk[i]])-b;
        root[i] = insert(root[i-1],x,1);
```

```
    }
    rep(i,1,m){
        int u,v,k; scanf("%d%d%d",&u,&v,&k);
        int pos = upper_bound(b+1,b+1+tb,k)-b;
        pos--;
        if(pos == 0) printf("0\n");
        else printf("%d\n",solve(u,v,pos));
    }
    return 0;
}
```

4.17 CDQ

4.17.1 陌上开花 (三维偏序)

描述:
有 n 朵花, 每朵花有三个属性: 花形 (s)、颜色 (c)、气味 (m), 用三个整数表示。
现在要对每朵花评级, 一朵花的级别是它拥有的美丽能超过的花的数量。
定义一朵花 A 比另一朵花 B 要美丽, 当且仅 Sa>=Sb,Ca>=Cb,Ma>=Mb。
显然, 两朵花可能有同样的属性。需要统计出评出每个等级的花的数量。

输入:
第一行为 N,K (1 <= N <= 100,000, 1 <= K <= 200,000), 分别表示花的数量和最大属性值。
以下 N 行，每行三个整数 si, ci, mi (1 <= si, ci, mi <= K)，表示第 i 朵花的属性

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl
    ;
typedef long long ll;
typedef double db;
const int N = 2*1e5+100;
const int M = 1e5+100;
const db EPS = 1e-9;
using namespace std;

int n,k,ans[N],tot,t[N];
struct Node{
    int a,b,c,s,ans;
    bool operator == (Node xx) const {
        return ((a == xx.a) && (b == xx.b) && (c == xx.c));
    }
    bool operator < (Node xx) const {
        if(b != xx.b) return b < xx.b;
        else return c < xx.c;
    }
}p[N];

bool cmp(Node x,Node y){
    if(x.a != y.a) return x.a < y.a;
    else if(x.b != y.b) return x.b < y.b;
    else return x.c < y.c;
}
```

```

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

inline void update(int x, int c){
    for(int i = x; i <= k; i += lowbit(i)) t[i] += c;
}

inline int ask(int x){
    int tp = 0;
    for(int i = x; i; i -= lowbit(i)) tp += t[i];
    return tp;
}

void CDQ(int l, int r)
{
    if(l == r) return;
    int mid = (l+r)>>1;
    CDQ(l,mid); CDQ(mid+1,r);
    sort(p+l,p+mid+1); sort(p+mid+1,p+r+1); //将第二维排序
    int i = l, j = mid+1;
    while(j <= r){
        while(i <= mid && p[i].b <= p[j].b){
            update(p[i].c,p[i].s);
            i++;
        }
        p[j].ans += ask(p[j].c);
        j++;
    }
    rep(kk,l,i-1) update(p[kk].c,-p[kk].s); //消除之前的影响，此处是 i，不是 mid
}

int main()
{
    scanf("%d%d",&n,&k);
    rep(i,1,n) scanf("%d%d%d",&p[i].a,&p[i].b,&p[i].c);
    //离散化
    sort(p+1,p+1+n,cmp);
    int cnt = 0;
    rep(i,1,n){
        cnt++;
        if(p[i]==p[i+1]) continue;
        p[++tot] = p[i], p[tot].s = cnt, cnt = 0, p[tot].ans = 0;
    }
    CDQ(1,tot);
    rep(i,1,tot)
        ans[p[i].ans+p[i].s-1] += p[i].s;
    rep(i,0,n-1) printf("%d\n",ans[i]);
    return 0;
}
```

4.17.2 动态逆序对

描述：
对于序列 A，它的逆序对数定义为满足 i<j，且 Ai>Aj 的数对 (i,j) 的个数。给 1 到 n 的一个排列，按照某种顺序依次删除 m 个元素，你的任务是在每次删除一个元素之前统计整个序列的逆序对数

输入：
输入第一行包含两个整数 n 和 m，即初始元素的个数和删除的元素个数。

以下 n 行每行包含一个 1 到 n 之间的正整数，即初始排列。
以下 m 行每行一个正整数，依次为每次删除的元素。
N<=100000 M<=50000

输出：
输出包含 m 行，依次为删除每个元素之前，逆序对的个数。

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const int N = 1e5+100;
const int M = 5*1e4+100;
const db EPS = 1e-9;
using namespace std;

int n,m,tot,a[N],vis[N],b[N],mp[N];
ll t[N],ans[N];
struct Node{
    int x,y,f; //插在x处，val = y，贡献对象为f
    bool operator < (Node xx) const {
        return x < xx.x;
    }
}p[N];

inline int lowbit(int x) { return x&(-x); }
inline void update(int x, int cc) { while(x <= n) t[x] += cc, x += lowbit(x); }
inline ll ask(int x) { ll tp = 0; while(x) tp += t[x], x -= lowbit(x); return tp; }

bool cmp(Node xx, Node yy){
    return xx.x > yy.x;
}

void solve(int l, int r){
    if(l == r) return;
    int mid = (l+r)>>1;
    solve(l,mid); solve(mid+1,r);
    sort(p+l,p+mid+1); sort(p+mid+1,p+r+1);
    int i = l, j = mid+1;
    while(j <= r){
        while(p[i].x <= p[j].x && i <= mid){
            update(p[i].y,1); i++;
        }
        int tp = ask(n)-ask(p[j].y);
        ans[p[j].f] += tp; j++;
    }
    rep(kk,l,i-1) update(p[kk].y,-1);

    sort(p+l,p+mid+1,cmp); sort(p+mid+1,p+r+1,cmp);
    i = l, j = mid+1;
```

```
    while(j <= r){
        while(p[i].x >= p[j].x && i <= mid){
            update(p[i].y,1); i++;
        }
        int tp = ask(p[j].y);
        ans[p[j].f] += tp; j++;
    }
    rep(kk,l,i-1) update(p[kk].y,-1);
}

int main()
{
    scanf("%d%d",&n,&m);
    rep(i,1,n){
        scanf("%d",&a[i]);
        mp[a[i]] = i;
    }
    rep(i,1,m){
        scanf("%d",&b[i]);
        vis[mp[b[i]]] = 1;
    }
    rep(i,1,n)
        if(!vis[i]) p[++tot] = {i,a[i],m};
    for(int i = m; i; i--){
        p[++tot] = {mp[b[i]],b[i],i};
    }
    solve(1,tot);
    for(int i = m-1; i; i--) ans[i] += ans[i+1];
    rep(i,1,m) printf("%lld\n",ans[i]);
    return 0;
}
```

4.17.3 矩阵前缀和

描述:
维护一个 W*W 的矩阵, 初始值均为 S. 每次操作可以增加某格子的权值, 或询问某子矩阵的总权值. 修改操作数 M<=160000, 询问数 Q<=10000,W<=2000000.

输入:
第一行两个整数,S,W; 其中 S 为矩阵初始值;W 为矩阵大小

接下来每行为一下三种输入之一 (不包含引号):

"1 x y a", 你需要把 (x,y)(第 x 行第 y 列) 的格子权值增加 a

"2 x1 y1 x2 y2", 你需要求出以左下角为 (x1,y1), 右上角为 (x2,y2) 的矩阵内所有格子的权值和, 并输出

"3", 表示输入结束

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl
    ;
```

```
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2
    << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const int N = 160000+4*1e4+100;
const int M = 1e4+100;
const db EPS = 1e-9;
using namespace std;

int T,n,ans[M],tot,num,t[2000010]; //tot-操作个数  num-查询个数
struct Node{
    int x,y,op,val,f; //x坐标 y坐标 op:1-修改 op:2-查询 val:答案贡献(1/-1) f:对应答案编号
    bool operator < (Node xx) const {
        if(x != xx.x) return x < xx.x;
        else return op < xx.op;
    }
}p[N];

inline int lowbit(int x) { return x&(-x); }
inline void update(int x, int val) { while(x <= n) t[x] += val, x += lowbit(x); }
inline int ask(int x){
    int tp = 0; while(x >= 1) tp += t[x], x -= lowbit(x);
    return tp;
}

void solve(int l, int r){
    if(l == r) return;
    int mid = (l+r)>>1;
    solve(l,mid); solve(mid+1,r);
    sort(p+l,p+mid+1); sort(p+mid+1,p+r+1);
    int i = l, j = mid+1;
    while(j <= r){
        if(p[j].op == 1){
            j++; continue;
        }
        while(p[i].x <= p[j].x && i <= mid){
            if(p[i].op == 1) update(p[i].y,p[i].val);
            i++;
        }
        ans[p[j].f] += p[j].val*ask(p[j].y);
        j++;
    }
    rep(kk,l,i-1){
        if(p[kk].op == 1) update(p[kk].y,-p[kk].val);
    }
}

int main()
{
    scanf("%d%d",&T,&n);
    int op,xx,yy,aa,zz;
    while(scanf("%d",&op)){
        if(op == 3) break;
        if(op == 1){
            scanf("%d%d%d",&xx,&yy,&aa);
            p[++tot] = {xx,yy,1,aa,0};
        }
        else{
            scanf("%d%d%d%d",&xx,&yy,&aa,&zz);
```

```
            ++num;
            p[++tot] = {aa,zz,2,1,num};
            p[++tot] = {xx-1,yy-1,2,1,num};
            p[++tot] = {xx-1,zz,2,-1,num};
            p[++tot] = {aa,yy-1,2,-1,num};
        }
    }
    solve(1,tot);
    rep(i,1,num) printf("%d\n",ans[i]);
    return 0;
}

/*
0 4
1 2 3 3
2 1 1 3 3
1 2 2 2
2 2 2 3 4
3
*/
```

4.18 莫队

4.18.1 普通莫队

莫队概述:

莫队本质上就是用分块去优化暴力的离线算法，将总复杂度降到 $O(n\sqrt{n})$ 的位置。说白了，就是分块 + 暴力。

我们先讲 ‘暴力的部分 ‘。比如一个长度为 n 的序列，m 次查询，每次查询询问区间 $[l,r]$ 之间的众数。对于这个问题，暴力求的话就是直接用桶记录每个数出现的次数，然后遍历区间 $[l,r]$，直接统计答案即可。这个暴力过程和莫队暴力过程没有任何区别，然后问题就变成了如何用分块来优化这个暴力呢?

在 ‘分块的部分 ‘，该算法将整个序列按照 $\sqrt{n}$ 大小进行分块，共分成 $\sqrt{n}$ 块，然后对于所有的询问，先按照左端点所在的块编号进行排序，如果块编号相同，再按照右端点排序。询问排序完之后，就直接暴力求解即可。代码的话看一下下面习题就可以掌握了。

最后就是 ‘时间复杂度 ‘的问题了。如何证明这个算法的时间复杂度呢? 我们对每一个块分开进行考虑，假设有 b_i 次操作在第 i 个块中，则在这个块中，右端点一定递增，因此右端点最多移动 n 次，而左端点每次最多移动 $\sqrt{n}$，一共最多移动 $b_i * \sqrt{n}$ 次，每次端点移动的时间复杂度为 $O(1)$，因此移动的总次数为 $\sum\limits_{i=1}^{\sqrt{n}}(b_i * \sqrt{n} + n) = m * \sqrt{n} + n * \sqrt{n}$，因此总复杂度为 $O(n\sqrt{n} + m\sqrt{n})$。

题意:
n 双颜色不同袜子，m 次询问，每次询问给出 $[L,R]$ 区间，询问在 $[L,R]$ 区间中随机抽出两双颜色相同的袜子的概率，输出最简分数形式 (A/B)。$(1 \le n, m \le 50000)$

思路:
普通莫队算法的复杂度是 $O(N\sqrt{N})$，实现关键点就在于能否在区间左右端点移动时，$O(1)$ 的更新答案。

我们观察这道题目，可以发现区间 $[L,R]$ 取出两双颜色相同袜子的概率 $= \frac{\frac{1}{2} * \sum_{i=L}^{R} num[i]}{C(R-L+1,2)}$，$num[i]$ 表示在区间 $[L,R]$ 中有多少双与 i 颜色相同的袜子，乘以 $\frac{1}{2}$ 的原因在于每一对颜色相同的袜子被计算了两遍。

分析到这里，就可以发现这是一道普通莫队的裸题，我们添加与删除时只需加上或减去当前与该点颜色相同的袜子数，这样同时可以避免重复计算。

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

```
#define rep(i,a,b) for(int i = a; i <= b; i++)
typedef long long ll;
const int N = 2*1e5+100;
using namespace std;

int a[N],pos[N],n,m,L,R;
ll ans[N][2],flag[N],Ans;
struct Node{
    int l,r,id;
    bool operator < (Node xx) const{
        if(pos[l] == pos[xx.l]) return r < xx.r;
        else return pos[l] < pos[xx.l];
    }
}Q[N];

ll gcd(ll a,ll b) {return b == 0 ? a:gcd(b,a%b);}

void add(int x){
    Ans += flag[a[x]];
    flag[a[x]]++;
}

void del(int x){
    flag[a[x]]--;
    Ans -= flag[a[x]];
}

int main()
{
    L = 1, R = 0;
    scanf("%d%d",&n,&m);
    int sz = sqrt(n);
    rep(i,1,n){
        scanf("%d",&a[i]);
        pos[i] = i/sz;
    }
    rep(i,1,m){
        scanf("%d%d",&Q[i].l,&Q[i].r);
        Q[i].id = i;
    }
    sort(Q+1,Q+1+m);
    rep(i,1,m){
        while(L < Q[i].l) del(L),L++;

        while(L > Q[i].l) L--, add(L);

        while(R < Q[i].r) R++, add(R);

        while(R > Q[i].r) del(R), R--;

        ll len = Q[i].r-Q[i].l+1;
        ll tp = len*(len-1ll)/(ll)2;
        ll g = gcd(Ans,tp);
        ans[Q[i].id][0] = Ans/g;
        ans[Q[i].id][1] = tp/g;
    }
    rep(i,1,m) printf("%lld/%lld\n",ans[i][0],ans[i][1]);
    return 0;
}
```

4.18.2 莫队求组合书前缀和

题意:

q 组询问，每次给出一个 n 和 m，求 $\sum\limits_{i=0}^{i=m} C_n^i$。$(1 \le n, m, q \le 2 * 10^5)$

思路:
令 $F(n,m) = \sum\limits_{i=1}^{m} C_n^i$，思考 $F(n,m)$ 与 $F(n,m+1)$ 和 $F(n+1,m)$ 之间的关系。

$F(n,m+1) = F(n+m) + C_n^{m+1}$，$F(n+1,m) = 2*F(n,m) - C_n^m$。预处理出阶乘和逆元之后，即可 $O(1)$ 进行端点移动。

总结:
这其实是一道广义莫队问题，所谓广义莫队问题就是题目中并没有明确指明查询区间 $[l,r]$ 的答案，而是将所查询的问题转化为 $F(n,m)$ 的形式，然后实现 $F(n,m)$ 与 $F(n+1,m)$ 以及 $F(n,m+1)$ 之间的 $O(1)$ 转移，只要求出之间转移的公式就可以直接 $O(n*\sqrt{n})$ 离线求出最终答案。

```
return 0;
```

4.18.3 带修改莫队

带修改莫队概述:
带修改的莫队仍然是利用分块对查询和修改排序，尽可能地减少运行时间。

假设我们按照 k 大小进行分块，则一共有 $\frac{n}{k}$ 个块，然后对于每个操作，一共有三个参数，分别是 l、r、id，表示区间左右端点和操作时间，我们先按照左端点的块号进行排序，再按照右端点的块号进行排序，最后按照操作时间进行排序。

莫队暴力时也需要维护三个值，L、R、T 表示当前控制的左右区间以及操作时间。‘对于每个查询，需要将 L、R、T 移动到指定位置再进行计算，因此可以将带修改莫队理解为三维莫队。

接下来估算复杂度，假设 m 次查询中，一共有 a 次查询，b 次修改。因此当确定左右端点块号时，即查询即按照时间排序时，T 最多移动 b 次，因此 T 的移动一共有 $\frac{n}{k} * \frac{n}{k} * b$ 次。而每次查询，区间左右端点最多移动 $2*k$ 次，因此 l、r 最多移动 $a*2*k$ 次，因此总时间复杂度为 $O(b*\frac{n^2}{k^2} + 2*a*k)$。我们可以求导求这个函数的最小值，可以发现最后的答案会在 $k = n^{\frac{2}{3}}$ 处取到最优解，因此整个算法的复杂度也就达到了 $O(n^{\frac{5}{3}})$ 处。

题意:
长度为 n 的初始序列，共有 m 次操作，操作 1 给出一个 l、r，令 c_i 为 i 在 $[l,r]$ 中出现的次数，询问 $Mex(c_0, c_1, ..., c_{10^9})$。操作 2 则将 a_p 改成 x。$(1 \le n, m \le 10^5)$

思路:
这个问题唯一的操作难点在于 mex 函数的求取，其实我们可以像求取 SG 函数的 mex 一样，直接暴力求取即可。然后其余部分就是常规的带修改莫队的操作了。

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <cmath>
#include <algorithm>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl
    ;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2
    << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
```

```
typedef double db;
const int N = 2*1e5+100;
const int M = 1e5+100;
const db EPS = 1e-9;
using namespace std;

int n,qq,a[N],b[N],tot,Qnum,Cnum,pos[N],ans[N],L,R,T,flag[N],vis[N];
struct Query{
    int l,r,id,t;
    bool operator < (Query xx) const {
        if(pos[l] != pos[xx.l]) return pos[l] < pos[xx.l];
        else if(pos[r] != pos[xx.r]) return pos[r] < pos[xx.r];
        else return t < xx.t;
    }
}q[M];
struct Change{
    int pos,val;
}C[M];

int find(int x){
    return lower_bound(b+1,b+1+tot,x)-b;
}

void add(int x){
    if(flag[a[x]]!=0) vis[flag[a[x]]]--;
    flag[a[x]]++; vis[flag[a[x]]]++;
}

void del(int x){
    vis[flag[a[x]]]--; flag[a[x]]--;
    if(flag[a[x]] != 0) vis[flag[a[x]]]++;
}

void Work(int x,int i){
    if(C[x].pos >= q[i].l && C[x].pos <= q[i].r){
        vis[flag[a[C[x].pos]]]--; flag[a[C[x].pos]]--;
        if(flag[a[C[x].pos]] != 0) vis[flag[a[C[x].pos]]]++;
        if(flag[C[x].val] != 0) vis[flag[C[x].val]]--;
        flag[C[x].val]++; vis[flag[C[x].val]]++;
    }
    swap(a[C[x].pos],C[x].val);
}

int solve(){
    rep(i,0,n)
        if(!vis[i]) return i;
}

int main()
{
    scanf("%d%d",&n,&qq);
    rep(i,1,n){
        scanf("%d",&a[i]);
        b[++tot] = a[i];
    }
    rep(i,1,qq){
        int op,l,r; scanf("%d%d%d",&op,&l,&r);
        if(op == 1) Qnum++, q[Qnum] = {l,r,Qnum,Cnum};
        else C[++Cnum] = {l,r}, b[++tot] = r;
```

```
    }
    sort(b+1,b+1+tot);
    tot = unique(b+1,b+1+tot)-b-1;
    int sz = pow(n,0.66666666666666);
    rep(i,1,n) pos[i] = i/sz;
    sort(q+1,q+1+Qnum);
    L = 1, R = 0, T = 0;
    vis[0] = 1;
    rep(i,1,n) a[i] = find(a[i]);
    rep(i,1,Cnum) C[i].val = find(C[i].val);
    rep(i,1,Qnum){
        while(L < q[i].l) del(L++);
        while(L > q[i].l) add(--L);
        while(R < q[i].r) add(++R);
        while(R > q[i].r) del(R--);
        while(T < q[i].t) Work(++T,i);
        while(T > q[i].t) Work(T--,i);
        ans[q[i].id] = solve();
    }
    rep(i,1,Qnum) printf("%d\n",ans[i]);
    return 0;
}
```

4.18.4 树上带修改莫队

树上带修莫队概述:
树上莫队问题仍然是通过分块进行解决，但是分块的序列发生了变化。这个序列需要满足，给出两点就能在序列上找出这两点之间的路径。

我们考虑常见的树上序列，dfs 序，但是很明显 dfs 序不满足这个条件，其中会有很多无效的节点。因此我们引出欧拉序来解决这个问题，欧拉序和 dfs 序的区别是，dfs 序只在遍历到这个节点时才会将这个节点加入序列，而欧拉序还会在回溯到这个节点时将节点加入序列。

因此在欧拉序中，每个点会有一个第一次到达的点和第二次到达的点，我们分别记为 $fir[i]$ 与 $las[i]$。对于树上两点 x、y ($fir[x] < fir[y]$)，u 为 x、y 两点的 lca，若 $x = u$，则在 $[fir[x], fir[y]]$ 这段区间中，只有 x 到 y 路径上的点只出现一次。若 $x \neq u$，则在 $[las[x], fir[y]]$ 这段区间中只有 x 到 y 路径上的点只出现一次，而且不包含 $lca(x, y)$ 这个点。因此我们在树上莫队问题中，需要记录每个点出现的次数，第一次出现则加贡献，第二次出现则减贡献，且若 $x \neq u$，还需加上 lca 的贡献。

解决完树上莫队的序列问题，就可以转化成普通莫队进行计算了。不带修改则块大小为 $\sqrt{2n}$，带修改则块大小为 $(2n)^{\frac{2}{3}}$，其中 $2n$ 为欧拉序长度。

题意:
n 个点的一棵树，每个点上都有一个糖果，糖果的种类不同，第 i 类糖果的贡献为 $V[i]$，第 j 次吃第 i 类糖果对答案的贡献为 $V[i] * W[j]$。现有 q 次操作，每次可以将第 x 个点上的糖果类型改为 y，也可以查询从 x 点到 y 点的答案。$(1 \leq n, q \leq 10^5)$

思路:
莫队问题只需要关注加入节点和删除节点对答案的影响，因此只需要统计每一类糖果在路径中出现的次数即可完成节点增删时对答案的影响。

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <cmath>
#include <algorithm>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
```

```
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const int N = 1e5+100;
const int M = 1e6+100;
const db EPS = 1e-9;
using namespace std;

int n,m,k,V[N],W[N],head[N],tot,C[N],qnum,cnum,f[N][25],t,d[N],Euler[2*N],ncnt,fir[N],las[N],pos[2*N],L,R,T,flag[N],vis[N];
//pos-分块位置、fir-欧拉序第一次、las-欧拉序第二次、Euler-欧拉序数组、ncnt-欧拉序数组长度
//vis-这个树上节点出现了几次，flag-这个糖果种类
ll ans[N],now;
struct Edge{
    int to,next;
}e[2*N];
struct Query{
    int l,r,id,lca,t; //l、r、id-查询顺序、lca-两点lca、t-之前有几次修改
    bool operator < (Query xx) const {
        if(pos[l] != pos[xx.l]) return pos[l] < pos[xx.l];
        else if(pos[r] != pos[xx.r]) return pos[r] < pos[xx.r];
        else return t < xx.t;
    }
}q[N];
struct Change{
    int pos, val;
}ch[N];

void add(int x,int y){
    e[++tot].to = y, e[tot].next = head[x], head[x] = tot;
}

void dfs(int u,int fa)
{
    Euler[++ncnt] = u; fir[u] = ncnt;
    d[u]=d[fa]+1; f[u][0]=fa;
    for(int i=1;(1<<i)<=d[u];i++)
        f[u][i]=f[f[u][i-1]][i-1];
    for(int i=head[u]; i; i=e[i].next){
        int v=e[i].to;
        if(v!=fa) dfs(v,u);
    }
    Euler[++ncnt] = u; las[u] = ncnt;
}

int LCA(int x,int y)
{
    if(d[x] > d[y]) swap(x,y);
    for(int i = t; i >= 0; i--)
        if(d[f[y][i]] >= d[x]) y = f[y][i];  //往上追溯，直至y和x位于同一深度
    if(x == y) return x;  //如果已经找到了，就返回x
    for(int i = t; i >= 0; i--)
        if(f[x][i] != f[y][i]) x = f[x][i], y = f[y][i];  //x和y同时往上走，一直到x和y恰好为lca的子节点
```

```
    return f[x][0];  //x和y共同的根节点就是lca
}

void Add(int pos){
    flag[C[pos]]++;
    now += (ll)W[flag[C[pos]]]*(ll)V[C[pos]];
}

void Del(int pos){
    now -= (ll)W[flag[C[pos]]]*(ll)V[C[pos]];
    flag[C[pos]]--;
}

void add_del(int pos){ //增加和减少取决于这个点被遍历了几次
    vis[pos] ? Del(pos) : Add(pos);
    vis[pos] ^= 1;
}

void work(int x){
    if(vis[ch[x].pos]){ //修改点为有效点
        add_del(ch[x].pos); //减掉
        swap(C[ch[x].pos], ch[x].val);
        add_del(ch[x].pos); //加上
    }
    else swap(C[ch[x].pos], ch[x].val);
}

int main()
{
    scanf("%d%d%d",&n,&m,&k);
    rep(i,1,m) scanf("%d",&V[i]);
    rep(i,1,n) scanf("%d",&W[i]);
    rep(i,1,n-1){
        int xx,yy; scanf("%d%d",&xx,&yy);
        add(xx,yy); add(yy,xx);
    }
    rep(i,1,n) scanf("%d",&C[i]);
    t = (int)(log(n)/log(2))+1;
    dfs(1,0);
    int sz = pow(ncnt,2.0/3.0);
    for(int i = 0; i <= ncnt; i++) pos[i] = i/sz;
    rep(i,1,k){
        int op,x,y; scanf("%d%d%d",&op,&x,&y);
        if(op){
            int lca = LCA(x,y);
            q[++qnum].t = cnum; q[qnum].id = qnum;
            if(fir[x] > fir[y]) swap(x,y);
            if(x == lca) q[qnum].l = fir[x], q[qnum].r = fir[y], q[qnum].lca = 0;
            else q[qnum].l = las[x], q[qnum].r = fir[y], q[qnum].lca = lca;
        }
        else ch[++cnum] = {x,y};
    }
    sort(q+1,q+1+qnum);
    L = 1, R = 0, T = 0;
    rep(i,1,qnum){
        while(L < q[i].l){
            add_del(Euler[L]); L++;
        }
        while(L > q[i].l){
```

```
            L--; add_del(Euler[L]);
        }
        while(R < q[i].r){
            R++; add_del(Euler[R]);
        }
        while(R > q[i].r){
            add_del(Euler[R]); R--;
        }
        while(T < q[i].t){
            ++T; work(T);
        }
        while(T > q[i].t){
            work(T); --T;
        }
        if(q[i].lca) add_del(q[i].lca); //lca不在欧拉序列区间中
        ans[q[i].id] = now;
        if(q[i].lca) add_del(q[i].lca); //恢复这个区间的状态
    }
    rep(i,1,qnum) printf("%lld\n",ans[i]);
    return 0;
}
```

4.18.5 回滚莫队（增加）

回滚莫队概述:
回滚莫队的关键点在于只能增加或删除节点，我们以求取 max 为例。求取 max 时，增加节点时可以顺便更新答案，但是删除节点时就非常不好维护，因此我们需要设计一个只需要增加节点的莫队算法。
1. 首先还是老套路，按照 $\sqrt{n}$ 进行分块，并确定每一块的左右边界，分别为 $xl[i]$ $xr[i]$。
2. 然后我们按照左端点所在块编号为第一关键字，右端点大小为第二关键字，对所有查询进行排序。
3. 接下来对于所有左右端点在同一块中的查询，我们直接暴力求取答案，复杂度为 $O(\sqrt{n})$。
4. 对于左端点所在块相同的查询，其右端点不断递增，因此右端点最多移动 $O(n)$，总共 $\sqrt{n}$ 个块，右端点复杂度为 $O(n\sqrt{n})$。
5. 接下来考虑左端点的移动，我们对于所有左端点所在块相同的查询，每一个查询结束之后都要把左端点移动到 $xr[i]+1$ 的位置，即左端点所在块的右端点 $+1$ 的位置，这样可以保证每次查询都是不断增加节点的，因此不会影响最终答案。每个查询，左端点最多移动距离为 $O(\sqrt{n})$，因此左端点移动的复杂度为 $O(m\sqrt{n})$。所以综合左右端点的移动，该算法的复杂度为 $O(m\sqrt{n}+n\sqrt{n})$，考虑到 m 的范围通常与 n 一致，因此最终复杂度为 $O(n\sqrt{n})$。

上述过程就是回滚莫队的求取过程，习题中分别给出了增加节点和减少节点的回滚莫队算法，其它具体实现细节可以查看代码。

题意:
长度为 n 的序列，每个数的大小为 x_i。一共 q 次查询，每次给出一个区间 l、r，询问区间 $[l,r]$ 中每个数贡献的最大值，一个数的贡献为 $x_i * cnt[x_i]$，即数大小 $*$ 该数出现次数。$(1 \le n, q, \le 10^5, 1 \le x_i \le 10^9)$

思路:
首先把序列离散化，然后用一个桶记录每一个数字出现的次数。

接下来就是回滚莫队的基本操作了，求出每块的左右端点，然后对查询排序。每次查询时判断左右端点是否在同一个快内，如果在就暴力求，如果不在就增加节点扩充区间。每个查询结束后，要将左端点再移动到该块的右边界 $+1$ 位置，具体的实现细节见代码。

```
#include <bits/stdc++.h>
#define rep(i,a,b) for(int i = a; i <= b; i++)
typedef long long ll;
const int N = 2e5+100;
using namespace std;

```

```
int n,m,sz,pos[N],a[N],b[N],tot,val[N],xl[N],xr[N],cnt[N],L,R,_cnt[N],lastblock;
//Maxn - 左右端点控制的最大值，temp - 临时最大值
//cnt - 左右端点移动时计数，_cnt - 左右端点同块时的计数
ll ans[N],Maxn,temp;
struct Node{
    int l,r,id;
    bool operator < (Node xx) const {
        if(pos[l] == pos[xx.l]) return r < xx.r;
        else return pos[l] < pos[xx.l];
    }
}q[N];

void init(){
    scanf("%d%d",&n,&m); sz = sqrt(n);
    rep(i,1,n) {scanf("%d",&a[i]); b[++tot] = a[i];}
    rep(i,1,m) {scanf("%d%d",&q[i].l,&q[i].r); q[i].id = i;}
    sort(b+1,b+1+tot); tot = unique(b+1,b+1+tot)-b-1;
    rep(i,1,n) val[i] = lower_bound(b+1,b+1+tot,a[i])-b;
    rep(i,1,n){
        pos[i] = i/sz;
        xl[pos[i]] = (xl[pos[i]] == 0 || xl[pos[i]] > i) ? i : xl[pos[i]];
        xr[pos[i]] = (xr[pos[i]] < i) ? i : xr[pos[i]];
    }
    sort(q+1,q+1+m);
}

inline ll add(int x){
    return (++cnt[val[x]])*(ll)b[val[x]];
}

inline void del(int x) {cnt[val[x]]--;}

void solve(){
    L = 1, R = 0, lastblock = -1;
    rep(i,1,m){
        if(pos[q[i].l] == pos[q[i].r]){
            ll temp = 0;
            rep(j,q[i].l,q[i].r) temp = max(temp,(++_cnt[val[j]])*(ll)b[val[j]]);
            rep(j,q[i].l,q[i].r) _cnt[val[j]]--;
            ans[q[i].id] = temp;
        }
        else{
            if(lastblock != pos[q[i].l]){
                while(L < xr[pos[q[i].l]]+1) del(L), L++;
                while(R > L-1) del(R), R--;
                Maxn = 0; lastblock = pos[q[i].l];
            }
            //Maxn为右半部分的最大值，不包含左端点所在块的情况
            while(R < q[i].r) R++, Maxn = max(Maxn,add(R));
            temp = Maxn;
            //temp从Maxn继承而来，表示整个区间的最大值
            while(L > q[i].l) L--, temp = max(temp,add(L));
            while(L < xr[pos[q[i].l]]+1) del(L), L++;
            ans[q[i].id] = temp;
        }
    }
}

int main()
```

```
{
    init();
    solve();
    rep(i,1,m) printf("%lld\n",ans[i]);
    return 0;
}
```

4.18.6 回滚莫队（减少）

题意:
长度为 n 的序列，每个数的大小为 a_i。一共 m 次查询，每次给出一个区间 l、r，询问区间 $[l,r]$ 中数的 mex，其中一个区间的 mex 指该区间内最小没有出现过的自然数。$(1 \le n,q, \le 2*10^5, 0 \le a_i \le 10^9)$

思路:
由于是求 mex，而数字总数为 $2e5$，因此不需要对数字进行离散化。然后我们来分析这个问题的关键点，即增删节点的特性。

不难发现，对于这个问题来说，删除节点可以 $O(1)$ 的更新答案，但是增加节点后答案的变化难以确定，因此考虑采用删除节点形式的回滚莫队来解决这个问题。

删除节点的回滚莫队，就是区间长度不断缩小的情况。因此我们需要对每个查询的左端点所在块编号进行升序，对每个查询的右端点进行降序，这样可以保证右端点是不断递减的。

然后对于左右端点在同一个块中的情况，我们依然是暴力求取答案。而对于不在同一块中的情况，我们需要每次查询结束后都将左端点移动到查询左端点所在的块的左边界上，这样才能保证区间长度在不断缩小。

除了上述这些回滚莫队的共性点之外，我们还需要关注一些特性点。对于这个问题，我们需要在最开始将左右边界分别设置为 1 和 n，这样的目的是保证区间长度是不断递减的。然后求取答案时，我们需要维护两部分答案，一部分是区间 $[l,r]$ 中完全包含左端点所在块的部分的答案，另一部分即为当前查询的结果。

保存第一部分答案的目的在于增加节点是不能 $O(1)$ 维护答案的，因此左端点递增之后答案就会变化而且不能恢复，所以如果不保存第一部分的答案是不能直接继承到下一个查询的，具体细节看代码就能够理解。

```
#include <bits/stdc++.h>
#define rep(i,a,b) for(int i = a; i <= b; i++)
const int N = 2e5+100;
using namespace std;

int n,m,a[N],sz,pos[N],xl[N],xr[N],cnt[N],ans[N],_cnt[N],lastblock,L,R;
struct Node{
    int l,r,id;
    bool operator < (Node xx) const {
        if(pos[l] == pos[xx.l]) return r > xx.r;
        else return pos[l] < pos[xx.l];
    }
}q[N];

void init(){
    scanf("%d%d",&n,&m);
    rep(i,1,n) scanf("%d",&a[i]);
    rep(i,1,m) scanf("%d%d",&q[i].l,&q[i].r), q[i].id = i;
    rep(i,1,n)
        if(a[i] > 2e5) a[i] = 2e5+1;
    int sz = sqrt(n);
    rep(i,1,n){
        pos[i] = i/sz; //点i所在块
        xl[pos[i]] = xl[pos[i]] == 0 ? i : xl[pos[i]]; //pos[i]块的左端点
        xr[pos[i]] = xr[pos[i]] < i ? i : xr[pos[i]]; //pos[i]块的右端点
```

```
    }
    sort(q+1,q+1+m);
}

inline void add(int x){
    cnt[a[x]]++;
}

inline void del(int x,int& hp){
    cnt[a[x]]--;
    if(cnt[a[x]] == 0 && a[x] < hp) hp = a[x];
}

void solve(){
    L = 1, R = n, lastblock = -1;
    rep(i,1,n) cnt[a[i]]++;
    int minn = 0;
    while(cnt[minn]) minn++;
    int base_min = minn;
    rep(i,1,m){
        if(pos[q[i].l] == pos[q[i].r]){
            rep(j,q[i].l,q[i].r) _cnt[a[j]]++;
            int now = 0;
            while(_cnt[now]) now++;
            rep(j,q[i].l,q[i].r) _cnt[a[j]]--;
            ans[q[i].id] = now;
        }
        else{
            if(lastblock != pos[q[i].l]){
                //每一次进入新的块时，右端点都是直接到n的，因此区间只有左端点在递增，可以不断O(1)维护答案
                while(R < n) R++, add(R);
                while(L < xl[pos[q[i].l]]) del(L,base_min), L++;
                minn = base_min; lastblock = pos[q[i].l];
            }
            //minn为包含左端点整个块的答案，用于继承到后续查询
            while(R > q[i].r) del(R,minn), R--;
            //temp为查询的答案
            int temp = minn;
            while(L < q[i].l) del(L,temp), L++;
            while(L > xl[pos[q[i].l]]) L--, add(L);
            ans[q[i].id] = temp;
        }
    }
}

int main()
{
    init();
    solve();
    rep(i,1,m) printf("%d\n",ans[i]);
    return 0;
}
```

4.19 虚树

题意：给定一棵树，敌人在 1 号节点，每次询问给出 k 个节点，每条边都有边权，问删掉树中边使得 1 号节点不能到达 k 个节点中任意一个节点需要的最小代价

输入：第一行一个整数 n，代表岛屿数量。
接下来 n-1 行，每行三个整数 u,v,w，代表 u 号岛屿和 v 号岛屿由一条代价为 c 的桥梁直接相连，保证 1<=u,v<=n 且 1<=c<=100000。
第 n+1 行，一个整数 m，代表敌方机器能使用的次数。
接下来 m 行，每行一个整数 ki，代表第 i 次后，有 ki 个岛屿资源丰富，接下来 k 个整数 h1,h2,⋯hk，表示资源丰富岛屿的编号。每次输出最小代价

思路：
首先考虑最暴力的 dp, 设 $f[x]$ 表示处理完以 x 为根的子树的最小花费

转移有两种情况
1. 断开自己与父亲的联系，代价为从根到该节点的最小值
2. 将子树内的节点全都处理掉的代价

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <queue>
#include <cmath>
#include <algorithm>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const int N = 250000+100;
const int M = 500000+100;
const db EPS = 1e-9;
using namespace std;

int n,tot1,tot2,head1[N],head2[N],m,dfn[N],d[N],f[N][20],t,idx,idq[N],top,stk[N],vis[N];
//idq —— 存取有效节点, top —— stk栈顶, vis —— 表示这个节点是否是有效节点
ll me[N];
struct Node{
    int to,next; ll w;
}e1[M],e2[M];

void add1(int x,int y,ll cc){
    e1[++tot1].to = y, e1[tot1].next = head1[x], e1[tot1].w = cc, head1[x] = tot1;
}
void add2(int x,int y,ll cc){
    e2[++tot2].to = y, e2[tot2].next = head2[x], e2[tot2].w = cc, head2[x] = tot2;
}
bool cmp(int a,int b){return dfn[a] < dfn[b];}

void bfs() //LCA部分
{
    queue<int> q;
    while(q.size()) q.pop();
    q.push(1); d[1] = 1;    //把1当做树根
    while(q.size())
    {
        int x = q.front(); q.pop();
```

```
        for(int i = head1[x]; i ;i = e1[i].next){
            int y = e1[i].to;
            if(d[y]) continue;
            d[y] = d[x]+1;
            f[y][0] = x;  //y走2^0步到达x
            for(int j = 1; j <= t;j++)
                f[y][j] = f[f[y][j-1]][j-1];
            q.push(y);
        }
    }
}

int LCA(int x,int y) //LCA部分
{
    if(d[x] > d[y]) swap(x,y);
    for(int i = t; i >= 0; i--)
        if(d[f[y][i]] >= d[x]) y = f[y][i];  //往上追溯，直至y和x位于同一深度
    if(x == y) return x;  //如果已经找到了，就返回x
    for(int i = t; i >= 0; i--)
        if(f[x][i] != f[y][i]) x = f[x][i], y = f[y][i];  //x和y同时往上走，一直到x和y恰好为lca的子节点
    return f[x][0];  //x和y共同的根节点就是lca
}

void dfs(int x,int fa){
    dfn[x] = ++idx;
    for(int i = head1[x]; i; i = e1[i].next){
        int y = e1[i].to;
        if(y == fa) continue;
        if(x == 1) me[y] = e1[i].w;
        else me[y] = min(me[x],e1[i].w);
        dfs(y,x);
    }
}

void insert(int u){ //构成虚树
    if(top == 1) {stk[++top] = u; return;}
    int lca = LCA(u,stk[top]); //与栈顶比较
    if(lca == stk[top]) {stk[++top] = u; return;}
    while(top > 1 && dfn[lca] <= dfn[stk[top-1]]){
        add2(stk[top-1],stk[top],0); --top;
    }
    if(lca != stk[top]){
        add2(lca,stk[top],0); stk[top] = lca;
    }
    stk[++top] = u;
}

ll DP(int x){ //使当前节点与子树中非有效节点不连通的最小代价
    ll cost = 0;
    for(int i = head2[x]; i; i = e2[i].next){
        int y = e2[i].to;
        cost += min(me[y],DP(y));
    }
    head2[x] = 0;
    if(vis[x]){
        vis[x] = 0;
        return me[x];
    }
```

```
    else return cost;
}

int main()
{
    scanf("%d",&n);
    t = (int)(log(n)/log(2))+1;
    rep(i,1,n-1){
        int xx,yy; ll cc;
        scanf("%d%d%lld",&xx,&yy,&cc);
        add1(xx,yy,cc); add1(yy,xx,cc);
    }
    bfs(); dfs(1,-1);
    scanf("%d",&m);
    rep(i,1,m){
        tot2 = 1;
        int sz; scanf("%d",&sz);
        rep(j,1,sz){
            scanf("%d",&idq[j]); vis[idq[j]] = 1;
        }
        sort(1+idq,1+idq+sz,cmp);
        top = 0; stk[++top] = 1; //加入树根
        rep(j,1,sz) insert(idq[j]); //构建虚树
        while(top > 1){
            add2(stk[top-1],stk[top],0); top--;
        }
        // LOG1("i",i);
        cout << DP(1) << endl;
    }
    return 0;
}
```

4.20 十字链表

题意：给出一个 $n*m$ 的矩阵，q 次操作，每次操作在矩阵中指定两个不重叠且不接触的小矩阵，将两个矩阵中的对应元素互换。在 q 次操作之后，输出最后的结果矩阵。$(2 \le n, m \le 10^3, 1 \le q \le 10^4)$

思路：此题是子矩阵交换，我们可以先考虑一维状态下的子序列交换，区间 $[a, b]$ 与 $[c, d]$ 交换，如下图所示。每个节点存储右指针，只需将节点 [a-1] 与 [c-1] 以及 [b] 与 [d] 交换右指针即可。此处需注意，如果两个区间重叠或相互接触，则会发生错误，可以自行手动模拟一下。

知道了一维情况下的操作，我们可以考虑二维情况下如何操作。首先每个节点需要维护向右与向下的指针，然后我们来考虑下图情况，两个子矩形相互交换元素。我们可以发现决定这个矩形具体所在位置只是 A1、A3 区域的向下指针与 A2、A4 区域的向右指针。

因此我们只需将 A1、A3 与 B1、B3 的向下指针进行交换，A2、A4 与 B2、B4 的向右指针进行交换即可完成题目要求。注意如果两个矩形有可能相交或接触，则此算法不可行。

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl
    ;
```

```
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2
    << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const int N = 2*1e6+100;
const int M = 1e5+100;
const db EPS = 1e-9;
using namespace std;

int n,m,q;
struct Node{
    int r,d,v;
}t[N];

int Pos(int x,int y){
    x++, y++;
    return (x-1)*(m+1)+y;
}

int main()
{
    scanf("%d%d%d",&n,&m,&q);
    rep(i,1,n)
        rep(j,1,m) scanf("%d",&t[Pos(i,j)].v);
    //一共n+1行，m+1列，从(0,0)开始编号
    rep(i,0,n)
        rep(j,0,m){
            t[Pos(i,j)].r = Pos(i,j+1);
            t[Pos(i,j)].d = Pos(i+1,j);
        }
    rep(i,1,q){
        int a,b,c,d,h,w;
        scanf("%d%d%d%d%d%d",&a,&b,&c,&d,&h,&w);
        int x = 1,y = 1,u,v;
        rep(j,1,a-1) x = t[x].d;
        rep(j,1,b-1) x = t[x].r;
        rep(j,1,c-1) y = t[y].d;
        rep(j,1,d-1) y = t[y].r;
        u = x, v = y; //先往下再往右，左下周长部分
        rep(j,1,h){
            u = t[u].d, v = t[v].d;
            swap(t[u].r,t[v].r);
        }
        rep(j,1,w){
            u = t[u].r, v = t[v].r;
            swap(t[u].d,t[v].d);
        }
        u = x, v = y; //先往右再往下，右上周长部分
        rep(j,1,w){
            u = t[u].r, v = t[v].r;
            swap(t[u].d,t[v].d);
        }
        rep(j,1,h){
            u = t[u].d, v = t[v].d;
            swap(t[u].r,t[v].r);
        }
    }
    int x = 1;
    x = t[x].d, x = t[x].r;
```

```
    rep(i,1,n){
        int y = x;
        rep(j,1,m){
            printf("%d",t[y].v);
            y = t[y].r;
            if(j == m) printf("\n");
            else printf(" ");
        }
        x = t[x].d;
    }
    return 0;
}
```

4.21 柯朵莉树

柯朵莉树应用范围：
一般适用于有区间赋值操作的题，且不能出现大量的修改操作，否则会时间爆炸

```
#include <bits/stdc++.h>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << " , " << z1 << ": " << z2 << endl;
#define IT set<Node>::iterator
typedef long long ll;
typedef double db;
const int N = 1e5+100;
const db EPS = 1e-9;
using namespace std;

struct Node{
    //节点含义是区间为[l,r]，区间中每个点大小均为v
    int l,r;
    mutable ll v; //否则无法直接修改set中的值
    Node(int L,int R = -1, ll V = 0):l(L),r(R),v(V) {}
    bool operator < (const Node& o) const {
        return l < o.l;
    }
};
set<Node> s;

ll pow_mod(ll a,ll b,ll mod){
    ll base = a%mod, ans = 1;
    while(b){
        if(b&1) ans = (ans*base)%mod;
        base = (base*base)%mod;
        b >>= 1;
    }
    return ans;
}

//分裂区间，返回区间左端点为pos的节点
IT split(int pos){
    IT it = s.lower_bound(Node(pos));
```

```
    if(it != s.end() && it->l == pos) return it;
    --it;
    int L = it->l, R = it->r;
    ll V = it->v;
    s.erase(it);
    s.insert(Node(L,pos-1,V));
    //pair<iterator,bool> insert (const value_type& val)
    return s.insert(Node(pos,R,V)).first;
}

//推平操作，合并set，令区间[l,r]中所有数等于val
void assign_val(int l,int r,ll val){
    IT itl = split(l), itr = split(r+1);
    s.erase(itl,itr);
    s.insert(Node(l,r,val));
}

//区间[l,r]中每个数加val
void add(int l,int r,ll val){
    IT itl = split(l), itr = split(r+1);
    for(; itl != itr; ++itl)
        itl->v += val;
}

//区间[l,r]中第k大
ll ranks(int l,int r,int k){
    vector<pair<ll,int> > vp; vp.clear();
    IT itl = split(l), itr = split(r+1);
    for(; itl != itr; ++itl)
        vp.push_back(pair<ll,int>(itl->v,itl->r - itl->l + 1));
    sort(vp.begin(),vp.end());
    for(vector<pair<ll,int> >::iterator it = vp.begin(); it != vp.end(); it++){
        k -= it->second;
        if(k <= 0) return it->first;
    }
}

//区间[l,r]x次方和
ll sum(int l,int r,int ex,ll mod){
    IT itl = split(l), itr = split(r+1);
    ll res = 0;
    for(; itl != itr; itl++)
        res = (res+(ll)(itl->r - itl->l + 1ll) * pow_mod(itl->v,(ll)ex,(ll)mod))%mod;
    return res;
}

int n,m;
ll seed,a[N],vmax;

ll rd(){
    ll ret = seed;
    seed = (seed*7ll+13ll)%((ll)1000000007);
    return ret;
}

int main()
{
    scanf("%d%d%lld%lld",&n,&m,&seed,&vmax);
    rep(i,1,n){
```

```
        a[i] = (rd()%vmax)+1ll;
        s.insert(Node(i,i,a[i]));
    }
    s.insert(Node(n+1,n+1,0)); //要在末尾添加一个节点
    rep(i,1,m){
        int op = int(rd()%4)+1;
        int l = int(rd()%n)+1;
        int r = int(rd()%n)+1;
        if(l > r) swap(l,r);
        int x,y;
        if(op == 3) x = int(rd()%(r-l+1))+1;
        else x = int(rd()%vmax)+1;
        if(op == 4)
            y = int(rd()%vmax)+1;
        //上方均为此题读入

        if(op == 1) add(l,r,(ll)x); //区间[l,r]加上x
        else if(op == 2) assign_val(l,r,(ll)x); //区间[l,r]所有数改成x
        else if(op == 3) printf("%lld\n",ranks(l,r,x)); //区间[l,r]中第k大数
        else printf("%lld\n",sum(l,r,x,y)); //区间[l,r]每个数字的x次方和%mod
    }
    return 0;
}
```

4.22 AC 自动机

4.22.1 AC 自动机模板

一、适用问题:

AC 自动机主要适用于多个模式串下的匹配问题，常嵌套 dp 进行考察或者利用其 $fail$ 树的性质进行出题。但是这些问题都有一个相同的特点，就是一定会题目给出或者人为建出多个字符串进行匹配，此点可以用于辨别 AC 自动机问题。

二、AC 自动机算法解析

基本步骤
1. 构造一颗 $Trie$ 树，作为 AC 自动机的搜索数据结构。
2. 构造 $fail$ 指针，使当前字符失配时跳转到具有最长公共前后缀的字符继续匹配。如同 kmp 算法一样，AC 自动机在匹配时如果当前字符匹配失败，那么利用 $fail$ 指针进行跳转。由此可知如果跳转，跳转后的串的前缀，必为跳转前的模式串的后缀并且跳转的新位置的深度（匹配字符个数）一定小于跳之前的节点。因此我们可以利用 bfs 在 $Trie$ 上面进行 $fail$ 指针的构造。

$fail$ 指针的构造
【定义】$fail$ 指针指向最长的可匹配后缀，$fail$ 指向点 p，表示从根节点到点 p 是当前匹配位置的最长后缀。
【构建方法】按深度 bfs 整颗字典树，得到当前节点的 $fail$ 指针需要查看其父亲的 $fail$ 指针是否有自己这个儿子，如果没有，需要继续跳 $fail$ 指针。

常数优化
最后我们在 AC 自动机上跑字符串的时候，我们在失配时，通过 $fail$ 求出匹配点，用该点更新失配节点对应儿子，优化常数。
举例】假如 $next[x][i] == 0$，表示节点 x 没有 i 这个节点，因此我们会不断跳 x 的 $fail$ 指针，直到找到一个节点 u，使得 $son[u][i]! = 0$，然后我们顺便设置 $son[x][i] = son[u][i]$，优化常数。

注意点
给出 n 个模式串以及 1 个匹配串，询问每个模式串在匹配串中出现的次数。在 AC 自动机上直接进行匹配时被匹配到的每个节点的 $fail$ 节点也都被匹配到了，不能忘记计算其 $fail$ 节点被匹配的次数。

```
struct Trie{
    int next[500010][26],fail[500010],end[500010]; //fail[i]是指从root到节点i这一段字母的最长后缀节点
    int root,L; //L相当于tot
    int newnode()
    {
        for(int i = 0;i < 26;i++)
            next[L][i] = -1; //将root节点的26个子节点都指向-1
        end[L++] = 0; //每一个子节点初始化都不是单词结尾
        return L-1;
    }
    void init()
    {
        L = 0;
        root = newnode(); //此处返回root = 0
    }
    void insert(char buf[])
    {
        int len = strlen(buf);
        int now = root;
        for(int i = 0;i < len;i++)
        {
            if(next[now][buf[i]-'a'] == -1)
                next[now][buf[i]-'a'] = newnode(); //不能在原有字典树中匹配的新字母则新建一个节点
            now = next[now][buf[i]-'a'];
        }
        end[now]++; //now这个节点是一个单词的结尾
    }
    void build()
    {
        queue<int>Q;
        fail[root] = root;
        for(int i = 0;i < 26;i++)
        {
            if(next[root][i] == -1)
                next[root][i] = root; //将root的未被访问的子节点指回root
            else
            {
                fail[next[root][i]] = root; //root子节点的失配指针指向root
                Q.push(next[root][i]); //队列中加入新节点
            }
        }
        while( !Q.empty() )
        {
            int now = Q.front();
            Q.pop();
            for(int i = 0;i < 26;i++)
                if(next[now][i] == -1)
                    next[now][i] = next[fail[now]][i]; //now的第i个节点未被访问，则将now的第i个节点指向now的fail节点的第i个节点
                else
                {
                    fail[next[now][i]]=next[fail[now]][i];
                    Q.push(next[now][i]);
                }
        }
    }
    int query(char buf[])
```

```
    {
        int len = strlen(buf);
        int now = root;
        int res = 0;
        for(int i = 0;i < len;i++)
        {
            now = next[now][buf[i]-'a'];
            int temp = now;
            while( temp != root )
            {
                res += end[temp]; //访问到了这里说明，从root到tmp都已经匹配成功
                end[temp] = 0; //加上这句话则同一模式串在匹配串中只会匹配一次，而不会多次匹配
                temp = fail[temp]; //循环遍历从root到tmp这串字符串的所有后缀
            }
        }
        return res;
    }
    void debug()
    {
        for(int i = 0;i < L;i++)
        {
            printf("id = %3d,fail = %3d,end = %3d,chi = [",i,fail[i],end[i]);
            for(int j = 0;j < 26;j++)
                printf("%2d",next[i][j]);
            printf("]\n");
        }
    }
}ac;
```

4.22.2 AC 自动机 +DP1

题意:
用 n 个基本字符组成一个长度为 m 的字符串，要求字符串中不能出现给定的 p 个非法串中任何一个，输出方案总数。$(1 \le n, m \le 50, 0 \le p \le 10)$

思路:
数据范围比较小，因此不难往 dp 上进行思考。又因为有多个非法串，考虑在 AC 自动机建的整个 $Trie$ 图上进行 dp。

我们定义状态为 $f[i][j]$，表示长度为 i，最后一个字符在 AC 自动机的第 j 个节点上，则枚举 j 的所有子节点，设 $now = next[j][k]$，即 now 为 j 的第 k 个子节点，则 $f[i][now] = f[i][now] + f[i-1][j]$，当且仅当 now 和 j 不为非法节点。

因此我们继续定义非法节点，一个点为非法节点，即该点所代表的字符串中出现了完整的非法串，很明显一个非法串的末尾节点是非法节点，并且若 $fail[now]$ 是非法节点，则 now 也为非法节点，因为 $fail[now]$ 节点所代表的字符串为 now 节点字符串的后缀。

除此之外，此题还有两个坑点。
1. 没有取模，因此需要大整数。
2. 字符的 $ASCII$ 码范围在 -128 128 之间，re 了一小时...

总结:
在 AC 自动机上进行 dp，就是以 AC 自动机上的节点作为状态进行转移，本质上与普通 dp 没有差别。

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

```
#define rep(i,a,b) for(int i = a; i <= b; i++)
typedef long long ll;
using namespace std;

char buf[60],base = 0;
int n,m,p,mp[301];

struct Trie{
    int next[2510][60],fail[2510],end[2510]; //fail[i]是指从root到节点i这一段字母的最长后缀节点
    int root,L; //L相当于tot
    int newnode()
    {
        for(int i = 0;i < 51;i++)
            next[L][i] = -1; //将root节点的26个子节点都指向-1
        end[L++] = 0; //每一个子节点初始化都不是单词结尾
        return L-1;
    }
    void init()
    {
        L = 0;
        root = newnode(); //此处返回root = 0
    }
    void insert(char buf[])
    {
        int len = strlen(buf);
        int now = root;
        for(int i = 0;i < len;i++)
        {
            int pos = mp[buf[i]-base+150];
            if(next[now][pos] == -1)
                next[now][pos] = newnode(); //不能在原有字典树中匹配的新字母则新建一个节点
            now = next[now][pos];
        }
        end[now] = 1;   //now这个节点是一个单词的结尾
    }
    void build()
    {
        queue<int>Q;
        fail[root] = root;
        for(int i = 0;i < 51;i++)
        {
            if(next[root][i] == -1)
                next[root][i] = root; //将root的未被访问的子节点指回root
            else
            {
                fail[next[root][i]] = root; //root子节点的失配指针指向root
                Q.push(next[root][i]); //队列中加入新节点
            }
        }
        while( !Q.empty() )
        {
            int now = Q.front();
            if(end[fail[now]]) end[now] = 1; //判断该节点是否非法
            Q.pop();
            for(int i = 0;i < 51;i++)
                if(next[now][i] == -1)
                    next[now][i] = next[fail[now]][i]; //now的第i个节点未被访问，则将now的第i个节点指向now的fail节点的第i个节点
```

```
                else
                {
                    fail[next[now][i]]=next[fail[now]][i];
                    Q.push(next[now][i]);
                }
        }
    }
}ac;

struct BigInteger{
    ll A[50];
    enum{MOD = 1000000};
    BigInteger(){memset(A, 0, sizeof(A)); A[0]=1;}
    void set(ll x){memset(A, 0, sizeof(A)); A[0]=1; A[1]=x;}
    void print(){
        printf("%lld", A[A[0]]);
        for (ll i=A[0]-1; i>0; i--){
            if (A[i]==0){printf("000000"); continue;}
            for (ll k=10; k*A[i]<MOD; k*=10ll) printf("0");
            printf("%lld", A[i]);
        }
        printf("\n");
    }
    ll& operator [] (int p) {return A[p];}
    const ll& operator [] (int p) const {return A[p];}
    BigInteger operator + (const BigInteger& B){
        BigInteger C;
        C[0]=max(A[0], B[0]);
        for (ll i=1; i<=C[0]; i++)
            C[i]+=A[i]+B[i], C[i+1]+=C[i]/MOD, C[i]%=MOD;
        if (C[C[0]+1] > 0) C[0]++;
        return C;
    }
    BigInteger operator * (const BigInteger& B){
        BigInteger C;
        C[0]=A[0]+B[0];
        for (ll i=1; i<=A[0]; i++)
            for (ll j=1; j<=B[0]; j++){
                C[i+j-1]+=A[i]*B[j], C[i+j]+=C[i+j-1]/MOD, C[i+j-1]%=MOD;
            }
        if (C[C[0]] == 0) C[0]--;
        return C;
    }
}f[2][2510];

int main()
{
    scanf("%d%d%d",&n,&m,&p);
    scanf("%s",buf);
    rep(i,0,n-1) mp[buf[i]-base+150] = i;
    ac.init();
    rep(i,1,p){
        scanf("%s",buf);
        ac.insert(buf);
    }
    ac.build();
    f[0][0].set(1);
    rep(i,1,m){
        rep(j,0,ac.L-1) f[i%2][j].set(0);
```

```
        rep(j,0,ac.L-1)
            rep(k,0,n-1){
                int now = ac.next[j][k];
                if(ac.end[now] == 0 && ac.end[j] == 0) f[i%2][now] = f[i%2][now] + f[(i-1)%2][j];
            }
    }
    BigInteger ans; ans.set(0);
    rep(i,0,ac.L-1) ans = ans+f[m%2][i];
    ans.print();
    return 0;
}
```

4.22.3 AC 自动机 + 最短路

题意:
给定 n 和 m，表示一共有 n 个点，每个点都有其所对应的坐标，m 条非法路径。先要从 1 号点走到 n 号点，但路径中不能出现 m 条非法路径中任意一条，求最短距离。$(1 \le n \le 50, 1 \le m \le 100)$

思路:
AC 自动机上跑最短路，思路比较明显。AC 自动机上的每一个节点都代表一个状态，且不能通过任何非法节点。

需要在 AC 自动机的根节点的所有儿子上都建立新节点，且如果一个节点的 $fail$ 节点为非法节点，则该节点也为非法节点。

建出 $Trie$ 图后，直接在图上跑 $dijkstra$ 最短路即可。

小坑点：坐标之差会爆 int...（找了半小时 bug）

```
#include <bits/stdc++.h>
#define rep(i,a,b) for(int i = a; i <= b; i++)
const int N = 100+10;
typedef double db;
const db inf = 1e15;
using namespace std;

void dbg() {cout << "\n";}
template<typename T, typename... A> void dbg(T a, A... x) {cout << a << ' '; dbg(x...);}
#define logs(x...) {cout << #x << " -> "; dbg(x);}

int n,m,base[100],vis[25100];
db X[N],Y[N],dis[25100];
struct Node{
    db ans; int a,b;
    bool operator < (Node xx) const {
        return ans > xx.ans;
    }
};
priority_queue<Node> q;

struct Trie{
    int next[25100][60],fail[25100],end[25100]; //fail[i]是指从root到节点i这一段字母的最长后缀节点
    int root,L; //L相当于tot
    int newnode()
    {
```

```
        for(int i = 0;i < 51;i++)
            next[L][i] = -1; //将root节点的26个子节点都指向-1
        end[L++] = 0; //每一个子节点初始化都不是单词结尾
        return L-1;
    }
    void init()
    {
        L = 0;
        root = newnode(); //此处返回root = 0
    }
    void insert(int len)
    {
        int now = root;
        for(int i = 0;i < len;i++)
        {
            int pos = base[i];
            if(next[now][pos] == -1)
                next[now][pos] = newnode(); //不能在原有字典树中匹配的新字母则新建一个节点
            now = next[now][pos];
        }
        end[now] = 1;   //now这个节点是一个单词的结尾
    }
    void build()
    {
        queue<int> Q;
        fail[root] = root;
        for(int i = 0;i < 51;i++)
        {
            if(next[root][i] == -1)
                next[root][i] = root; //将root的未被访问的子节点指回root
            else
            {
                fail[next[root][i]] = root; //root子节点的失配指针指向root
                Q.push(next[root][i]); //队列中加入新节点
            }
        }
        while( !Q.empty() )
        {
            int now = Q.front();
            if(end[fail[now]]) end[now] = 1; //判断该节点是否非法
            Q.pop();
            for(int i = 0;i < 51;i++)
                if(next[now][i] == -1)
                    next[now][i] = next[fail[now]][i]; //now的第i个节点未被访问，则将now的第i个节点指向now的fail节点的第i个节点
                else
                {
                    fail[next[now][i]]=next[fail[now]][i];
                    Q.push(next[now][i]);
                }
        }
    }
}ac;

db dist(int i,int j){
    //坐标之差会爆int...
    db tmp = (X[i]-X[j])*(X[i]-X[j])+(Y[i]-Y[j])*(Y[i]-Y[j]);
    tmp = sqrt(tmp);
    return tmp;
```

```
}

void dijkstra(){
    while(q.size()) q.pop();
    rep(i,0,50)
        if(ac.next[ac.root][i] == 0){
            int p = ac.newnode();
            ac.next[ac.root][i] = p;
            rep(j,0,50) ac.next[p][j] = 0;
        }
    rep(i,0,ac.L) dis[i] = inf, vis[i] = 0;
    q.push({0,ac.next[ac.root][1],1}); dis[ac.next[ac.root][1]] = 0;
    db ans = inf;
    while(q.size()){
        int now = q.top().a, id = q.top().b; q.pop();
        if(vis[now]) continue;
        vis[now] = 1;
        if(id == n) ans = min(ans,dis[now]);
        rep(i,id+1,n){
            db tp = dis[now]+dist(id,i);
            int y1 = now, y2 = ac.next[now][i];
            while(y2 == 0){
                y1 = ac.fail[y1];
                y2 = ac.next[y1][i];
            }
            ac.next[now][i] = y2;
            if(!vis[y2] && !ac.end[y2] && dis[y2] > tp){
                dis[y2] = tp;
                q.push({dis[y2],y2,i});
            }
        }
    }
    if(ans == inf) printf("Can not be reached!\n");
    else printf("%.2f\n",ans);
}

int main(){
    while(~scanf("%d%d",&n,&m)){
        if(n == 0 && m == 0) break;
        rep(i,1,n) scanf("%lf%lf",&X[i],&Y[i]);
        ac.init();
        rep(i,1,m){
            int k; scanf("%d",&k);
            rep(j,0,k-1) scanf("%d",&base[j]);
            ac.insert(k);
        }
        ac.build();
        dijkstra();
    }
}
```

4.22.4 AC 自动机 +DP2

题意:
给定 n 和 m，表示要组成一个长度为 n 的字符串，再给出 m 个字符串，每个串都有一个能量值，现让你组成一个串，使得你组成的串的总能量值最大。若能量值相同输出长度小的，长度相同则输出字典序小的。$(0 \le n \le 50, 0 \le m \le 100)$

思路:

涉及多个串的问题，一般都采用 AC 自动机进行解决，我们以 AC 自动机中每一个节点为状态，进行状态转移。定义 $f[i][j]$ 表示长度为 i 的串，当前处于 AC 自动机上的 j 节点的最大能量值，定义 $s[i][j]$ 为 $f[i][j]$ 状态下所组成的串，然后直接进行 dp 转移即可。

注意，此问题中每个节点的贡献值不单只是其自身串的贡献，还有其所有 $fail$ 节点贡献的累加值。

```
#include <bits/stdc++.h>
#define rep(i,a,b) for(int i = a; i <= b; i++)
const int inf = 1e8;
using namespace std;

struct Trie{
    int next[500010][26],fail[500010],end[500010]; //fail[i]是指从root到节点i这一段字母的最长后缀节点
    int root,L; //L相当于tot
    int newnode()
    {
        for(int i = 0;i < 26;i++)
            next[L][i] = -1; //将root节点的26个子节点都指向-1
        end[L++] = 0; //每一个子节点初始化都不是单词结尾
        return L-1;
    }
    void init()
    {
        L = 0;
        root = newnode(); //此处返回root = 0
    }
    void insert(char buf[],int tp)
    {
        int len = strlen(buf);
        int now = root;
        for(int i = 0; i < len; i++){
            if(next[now][buf[i]-'a'] == -1)
                next[now][buf[i]-'a'] = newnode(); //不能在原有字典树中匹配的新字母则新建一个节点
            now = next[now][buf[i]-'a'];
        }
        end[now] = tp;  //now这个节点是一个单词的结尾
    }
    void build()
    {
        queue<int>Q;
        fail[root] = root;
        for(int i = 0;i < 26;i++)
        {
            if(next[root][i] == -1)
                next[root][i] = root; //将root的未被访问的子节点指回root
            else
            {
                fail[next[root][i]] = root; //root子节点的失配指针指向root
                Q.push(next[root][i]); //队列中加入新节点
            }
        }
        while( !Q.empty() ) //按长度进行bfs
        {
            int now = Q.front();
            end[now] += end[fail[now]]; //继承前面节点的value
            Q.pop();
            for(int i = 0;i < 26;i++)
```

```
                if(next[now][i] == -1)
                    next[now][i] = next[fail[now]][i]; //now的第i个节点未被访问，则将now的第i个节点指向now的fail节点的第i个节点
                else
                {
                    fail[next[now][i]]=next[fail[now]][i];
                    Q.push(next[now][i]);
                }
        }
    }
    int query(char buf[])
    {
        int len = strlen(buf);
        int now = root;
        int res = 0;
        for(int i = 0;i < len;i++)
        {
            now = next[now][buf[i]-'a'];
            int temp = now;
            while( temp != root )
            {
                res += end[temp]; //访问到了这里说明，从root到tmp都已经匹配成功
                end[temp] = 0; //加上这句话则同一模式串在匹配串中只会匹配一次，而不会多次匹配
                temp = fail[temp]; //循环遍历从root到tmp这串字符串的所有后缀
            }
        }
        return res;
    }
}ac;

char buf[211][111];
int n,m,f[55][1200]; //pre记录从哪个节点来以及是啥字符
string s[55][1200];

void solve(){
    rep(i,0,n)
        rep(j,0,ac.L) f[i][j] = -inf, s[i][j] = "\0";
    f[0][0] = 0;
    rep(i,1,n)
        rep(j,0,ac.L-1){
            rep(k,0,25){
                int now = ac.next[j][k];
                int tp = f[i-1][j]+ac.end[now];
                string hp = s[i-1][j]+(char)(k+'a');
                if(f[i][now] < tp || (f[i][now] == tp && hp < s[i][now])){
                    f[i][now] = tp;
                    s[i][now] = hp;
                }
            }
        }
    int ans = 0;
    rep(i,1,n)
        rep(j,0,ac.L-1) ans = max(ans,f[i][j]);
    if(ans == 0) printf("\n");
    else{
        string str = "\0"; int len = 100;
        rep(i,1,n){
            rep(j,0,ac.L-1)
                if(f[i][j] == ans){
```

```
                    if(i < len || (i == len && s[i][j] < str))
                            len = i, str = s[i][j];
                }
        }
        cout << str << endl;
    }
}

int main()
{
    int _; scanf("%d",&_);
    while(_--)
    {
        scanf("%d%d",&n,&m);
        ac.init();
        rep(i,1,m) scanf("%s",buf[i]);
        rep(i,1,m){
            int tp; scanf("%d",&tp);
            ac.insert(buf[i],tp);
        }
        ac.build();
        solve();
    }
    return 0;
}
```

4.22.5 AC 自动机 + 状压 DP

题意:
给出 L 和 N，表示密码的长度和观察到的子串个数。再给出 N 个子串，询问长度为 L 的密码包含这 N 个子串的所有可能方案。如果方案总数小于 42，则输出所有方案。$(1 \le L \le 25, 1 \le N \le 10)$

思路:
N 范围比较小，因此考虑状压 dp，$f[i][S][j]$ 表示长度为 i，包含的子串状态为 S，当前在 AC 自动机的第 j 个节点上，所有可能的方案数，直接转移即可求取答案。

但是此题还需要输出方案，因此我们对于每个 $f[i][S][j]$ 开一个 $vector$ 存储所有可能情况，但是需要剪枝，不然会 mle，不能让任何一个 $vector$ 中的方案数大于 42，如此即可完成此题。

```
#include <bits/stdc++.h>
typedef long long ll;
using namespace std;

struct Trie{
    int next[200][26],fail[200],end[200]; //fail[i]是指从root到节点i这一段字母的最长后缀节点
    int root,L; //L相当于tot
    int newnode()
    {
        for(int i = 0;i < 26;i++)
            next[L][i] = -1; //将root节点的26个子节点都指向-1
        end[L++] = 0; //每一个子节点初始化都不是单词结尾
        return L-1;
    }
    void init()
    {
        L = 0;
        root = newnode(); //此处返回root = 0
```

```
    }
    void insert(char buf[],int flag)
    {
        int len = strlen(buf);
        int now = root;
        for(int i = 0;i < len;i++)
        {
            if(next[now][buf[i]-'a'] == -1)
                next[now][buf[i]-'a'] = newnode(); //不能在原有字典树中匹配的新字母则新建一个节点
            now = next[now][buf[i]-'a'];
        }
        end[now] = flag;    //now这个节点是一个单词的结尾
    }
    void build()
    {
        queue<int>Q;
        fail[root] = root;
        for(int i = 0;i < 26;i++)
        {
            if(next[root][i] == -1)
                next[root][i] = root; //将root的未被访问的子节点指回root
            else
            {
                fail[next[root][i]] = root; //root子节点的失配指针指向root
                Q.push(next[root][i]); //队列中加入新节点
            }
        }
        while( !Q.empty() )
        {
            int now = Q.front();
            end[now] += end[fail[now]];
            Q.pop();
            for(int i = 0;i < 26;i++)
                if(next[now][i] == -1)
                    next[now][i] = next[fail[now]][i]; //now的第i个节点未被访问，则将now的第i个节点指向now的fail节点的第i个节点
                else
                {
                    fail[next[now][i]]=next[fail[now]][i];
                    Q.push(next[now][i]);
                }
        }
    }
}ac;

char buf[110];
int n,m;
ll dp[27][1050][80];
vector<string> base[27][1050][80];

void solve(){
    memset(dp,0,sizeof dp);
    dp[0][0][0] = 1;
    for(int i = 1; i <= n; i++)
        for(int S = 0; S <= (1<<m)-1; S++)
            for(int j = 0; j <= ac.L-1; j++)
                for(int k = 0; k <= 25; k++){
                    int now = ac.next[j][k];
```

```
                    dp[i][S|ac.end[now]][now] += dp[i-1][S][j];
                }
    ll ans = 0;
    for(int j = 0; j <= ac.L-1; j++) ans += dp[n][(1<<m)-1][j];
    if(ans <= 42){
        for(int i = 0; i <= n; i++)
            for(int S = 0; S <= (1<<m)-1; S++)
                for(int j = 0; j <= ac.L-1; j++){
                    dp[i][S][j] = 0; base[i][S][j].clear();
                }
        dp[0][0][0] = 1;
        for(int i = 1; i <= n; i++)
            for(int S = 0; S <= (1<<m)-1; S++)
                for(int j = 0; j <= ac.L-1; j++){
                    if(dp[i-1][S][j] == 0) continue;
                    for(int k = 0; k <= 25; k++){
                        int now = ac.next[j][k];
                        dp[i][S|ac.end[now]][now] += dp[i-1][S][j];
                        if((int)base[i][S|ac.end[now]][now].size()+(int)base[i-1][S][j].size() > ans) continue;
                        if(dp[i-1][S][j] != 0 && (int)base[i-1][S][j].size() == 0){
                            string tp = "\0"; tp += (char)('a'+k);
                            base[i][S|ac.end[now]][now].push_back(tp);
                        }
                        else{
                            for(auto &v:base[i-1][S][j]){
                                string tp = v+(char)('a'+k);
                                base[i][S|ac.end[now]][now].push_back(tp);
                            }
                        }
                    }
                }
        vector<string> fin; fin.clear();
        for(int j = 0; j < ac.L; j++){
            for(auto &v:base[n][(1<<m)-1][j])
                fin.push_back(v);
        }
        printf("%lld\n",ans);
        sort(fin.begin(),fin.end());
        for(auto &v:fin)
            cout << v << endl;
    }
    else printf("%lld\n",ans);
}

int main()
{
    scanf("%d%d",&n,&m);
    ac.init();
    for(int i = 0;i < m;i++){
        scanf("%s",buf);
        ac.insert(buf,1<<i);
    }
    ac.build();
    solve();
    return 0;
}
```

4.22.6 AC 自动机 + 缩点

题意:
给定 m，表示构成字符串只能使用前 m 个字符，再给定一个包含 n 个禁止串的集合。然后询问是否能够构成一个无限长的串，满足以下三个条件。
1. 该串中只使用了前 m 个字符
2. 该串中不包含任何禁止串
3. 该串不能在任何位置开始出现循环节

如果可以构造则输出 Yes，否则输出 No。$(1 \le n \le 100, 1 \le m \le 26)$

思路:
先不考虑循环节的问题，如果单纯考虑能否构造一个只使用了前 m 个字符，不包含禁止串且长度无限的串，这个问题就是一个经典问题。

我们将禁止串插入 AC 自动机中，所有的 $next$ 出边组成一个有向图，我们将所有非法节点从图中去除，得到一个合法的有向图，我们可以在这个合法的图上任意走。因此仅当这个图中出现了环，我们才可以构建出一个无限长的串，否则不行。

在上面的基础上，我们考虑如何构建一个不出现循环节的长度无限的串，无限长则说明有环，那如果这是个简单环，则在不断绕行的过程中一定会出现循环节。而如果这个环中有好几个环，则可以在左边的环绕一圈，右边的再绕两圈，因此不会出现循环节。

因此此题只需要构建 AC 自动机之后，去除无效节点，然后对剩下的有向图跑强连通分量，对于每个强连通分量查看是否有多个环即可。一个简单环中最多只有 n 条边，如果大于 n 条边，则非简单环，因此只需判定一个环中的边数量即可。

```
#include <bits/stdc++.h>
const int N = 1e5+10;
typedef long long ll;
using namespace std;

char buf[N];
int n,m,pos[N],tot,head[N],dfn[N],top,sz[N],vis[N]; //top记录dfs序
vector<pair<int,int> > v[N];
ll c[N],ans[N];
struct Node{
    int to,next;
}e[N];
void add(int x,int y){
    e[++tot].to = y, e[tot].next = head[x], head[x] = tot;
}

struct Trie{
    int next[N][26],fail[N],end[N],fa[N]; //fail[i]是指从root到节点i这一段字母的最长后缀节点
    int root,L; //L相当于tot
    int newnode()
    {
        for(int i = 0;i < 26;i++)
            next[L][i] = -1; //将root节点的26个子节点都指向-1
        fa[L] = 0; end[L++] = 0; //每一个子节点初始化都不是单词结尾
        return L-1;
    }
    void init()
    {
        L = 0;
        root = newnode(); //此处返回root = 0
    }
    void insert(char buf[])
```

```
    {
        int len = strlen(buf), cnt = 0;
        int now = root;
        for(int i = 0;i < len;i++)
        {
            if(buf[i] == 'B') now = fa[now];
            else if(buf[i] == 'P') pos[++cnt] = now;
            else{
                if(next[now][buf[i]-'a'] == -1){
                    next[now][buf[i]-'a'] = newnode(); //不能在原有字典树中匹配的新字母则新建一个节点
                    fa[next[now][buf[i]-'a']] = now;
                }
                now = next[now][buf[i]-'a'];
            }
        }
    }
    void build()
    {
        queue<int>Q;
        fail[root] = root;
        for(int i = 0;i < 26;i++)
        {
            if(next[root][i] == -1)
                next[root][i] = root; //将root的未被访问的子节点指回root
            else
            {
                fail[next[root][i]] = root; //root子节点的失配指针指向root
                Q.push(next[root][i]); //队列中加入新节点
            }
        }
        while( !Q.empty() )
        {
            int now = Q.front();
            add(fail[now],now);
            Q.pop();
            for(int i = 0;i < 26;i++)
                if(next[now][i] == -1)
                    next[now][i] = next[fail[now]][i]; //now的第i个节点未被访问，则将now的第i个节点指向now的fail节点的第i个节点
                else
                {
                    fail[next[now][i]]=next[fail[now]][i];
                    Q.push(next[now][i]);
                }
        }
    }
}ac;

inline int lowbit(int x) {return x&(~x+1);}
inline void update(int x,ll v) {for(;x<=ac.L;x+=lowbit(x)) c[x] += v;}
inline ll ask(int x){
    ll tp = 0;
    while(x) tp += c[x], x -= lowbit(x);
    return tp;
}

void dfs(int x){
    dfn[x] = ++top; sz[x] = 1;
```

```
    for(int i = head[x]; i; i = e[i].next){
        int y = e[i].to; dfs(y);
        sz[x] += sz[y];
    }
}

void solve(){
    int len = strlen(buf), now = ac.root;
    int cnt = 0;
    for(int i = 0; i < len; i++){
        if(buf[i] == 'B'){
            update(dfn[now],-1);
            now = ac.fa[now];
            vis[now] = 0;
            continue;
        }
        else if(buf[i] == 'P'){
            cnt++;
            for(auto &hp:v[cnt]){
                int x = hp.first, y = cnt, id = hp.second;
                ll mp = ask(dfn[pos[x]]+sz[pos[x]]-1)-ask(dfn[pos[x]]-1);
                ans[id] = mp;
            }
            continue;
        }
        else now = ac.next[now][buf[i]-'a'];
        if(!vis[now]){
            vis[now] = 1;
            update(dfn[now],1);
        }
        else{
            vis[now] = 0;
            update(dfn[now],-1);
        }
    }
    for(int i = 1; i <= m; i++) printf("%lld\n",ans[i]);
}

int main()
{
    scanf("%s",buf);
    scanf("%d",&m);
    for(int i = 1; i <= m; i++){
        int l,r; scanf("%d%d",&l,&r);
        v[r].push_back(make_pair(l,i));
    }
    ac.init();
    ac.insert(buf);
    tot = 1;
    ac.build();
    for(int i = 0; i < ac.L; i++)
        if(!dfn[i]) dfs(i);
    solve();
    return 0;
}
```

4.22.7 阿狸打字机

题意:
给出一个长串。其中'B' 表示最后一个字母被删除，'P' 表示将当前字符打印出来。一共 m 次询问，每次询问给出一个 (x, y)，表示查询第 x 次打印的字符串在第 y 次打印的字符串中出现了多少次。$(1 \le n, m \le 10^5)$

思路:
首先先将长串加入到 AC 自动机中，然后记录每个节点的父节点，遇到'B' 就往上走，遇到 P 就标记第 x 次打印对应的是 AC 自动机上的哪个节点。

构造完 AC 自动机后，问题就变成了如何查询其中一个节点在另一个节点中出现的次数。我们首先考虑暴力匹配，其实就是取出字符串 y，然后在 AC 自动机中跑，所有经过的节点都标记一下，然后查询有多少个被标记的节点中包含 x。即在 y 跑完之后，倒着将标记压到 $fail$ 节点上，然后查询 x 节点的标记次数即可。

在这个暴力过程中，我们可以发现一个节点在另一个节点中出现的次数，取决于其所有 $fail$ 子孙被标记的次数，因此我们将查询离线，按照右端点排序。然后根据 AC 自动机的 $fail$ 指针建立 $fail$ 树，求出 dfs 序，用树状数组维护每个节点子树中被标记的次数。

总结:
此题最大的收获是彻底加深了对于 $fail$ 指针的理解，一个节点在另一个字符串中出现的次数，取决了该节点所有 $fail$ 子树中被标记的次数之和。

```
#include <bits/stdc++.h>
const int N = 1e5+10;
typedef long long ll;
using namespace std;

char buf[N];
int n,m,pos[N],tot,head[N],dfn[N],top,sz[N],vis[N]; //top记录dfs序
vector<pair<int,int> > v[N];
ll c[N],ans[N];
struct Node{
    int to,next;
}e[N];
void add(int x,int y){
    e[++tot].to = y, e[tot].next = head[x], head[x] = tot;
}

struct Trie{
    int next[N][26],fail[N],end[N],fa[N]; //fail[i]是指从root到节点i这一段字母的最长后缀节点
    int root,L; //L相当于tot
    int newnode()
    {
        for(int i = 0;i < 26;i++)
            next[L][i] = -1; //将root节点的26个子节点都指向-1
        fa[L] = 0; end[L++] = 0; //每一个子节点初始化都不是单词结尾
        return L-1;
    }
    void init()
    {
        L = 0;
        root = newnode(); //此处返回root = 0
    }
    void insert(char buf[])
    {
        int len = strlen(buf), cnt = 0;
        int now = root;
        for(int i = 0;i < len;i++)
        {
            if(buf[i] == 'B') now = fa[now];
            else if(buf[i] == 'P') pos[++cnt] = now;
```

```
            else{
                if(next[now][buf[i]-'a'] == -1){
                    next[now][buf[i]-'a'] = newnode(); //不能在原有字典树中匹配的新字母则新建一个节点
                    fa[next[now][buf[i]-'a']] = now;
                }
                now = next[now][buf[i]-'a'];
            }
        }
    }
    void build()
    {
        queue<int>Q;
        fail[root] = root;
        for(int i = 0;i < 26;i++)
        {
            if(next[root][i] == -1)
                next[root][i] = root; //将root的未被访问的子节点指回root
            else
            {
                fail[next[root][i]] = root; //root子节点的失配指针指向root
                Q.push(next[root][i]); //队列中加入新节点
            }
        }
        while( !Q.empty() )
        {
            int now = Q.front();
            add(fail[now],now);
            Q.pop();
            for(int i = 0;i < 26;i++)
                if(next[now][i] == -1)
                    next[now][i] = next[fail[now]][i]; //now的第i个节点未被访问，则将now的第i个节点指向now的fail节点的第i个节点
                else
                {
                    fail[next[now][i]]=next[fail[now]][i];
                    Q.push(next[now][i]);
                }
        }
    }
}ac;

inline int lowbit(int x) {return x&(~x+1);}
inline void update(int x,ll v) {for(;x<=ac.L;x+=lowbit(x)) c[x] += v;}
inline ll ask(int x){
    ll tp = 0;
    while(x) tp += c[x], x -= lowbit(x);
    return tp;
}

void dfs(int x){
    dfn[x] = ++top; sz[x] = 1;
    for(int i = head[x]; i; i = e[i].next){
        int y = e[i].to; dfs(y);
        sz[x] += sz[y];
    }
}

void solve(){
```

```
    int len = strlen(buf), now = ac.root;
    int cnt = 0;
    for(int i = 0; i < len; i++){
        if(buf[i] == 'B'){
            update(dfn[now],-1);
            now = ac.fa[now];
            vis[now] = 0;
            continue;
        }
        else if(buf[i] == 'P'){
            cnt++;
            for(auto &hp:v[cnt]){
                int x = hp.first, y = cnt, id = hp.second;
                ll mp = ask(dfn[pos[x]]+sz[pos[x]]-1)-ask(dfn[pos[x]]-1);
                ans[id] = mp;
            }
            continue;
        }
        else now = ac.next[now][buf[i]-'a'];
        if(!vis[now]){
            vis[now] = 1;
            update(dfn[now],1);
        }
        else{
            vis[now] = 0;
            update(dfn[now],-1);
        }
    }
    for(int i = 1; i <= m; i++) printf("%lld\n",ans[i]);
}

int main()
{
    scanf("%s",buf);
    scanf("%d",&m);
    for(int i = 1; i <= m; i++){
        int l,r; scanf("%d%d",&l,&r);
        v[r].push_back(make_pair(l,i));
    }
    ac.init();
    ac.insert(buf);
    tot = 1;
    ac.build();
    for(int i = 0; i < ac.L; i++)
        if(!dfn[i]) dfs(i);
    solve();
    return 0;
}
```

4.23 回文串问题

4.23.1 Manacher 模板

一、适用问题:

$Manacher$ 算法主要解决的是给出一个字符串，$O(n)$ 复杂度下求出以字符串中任意一个节点为中心所能扩展的最大距离。

二、$Manacher$ 算法解析

(1) 扩充字符串
为了统一奇偶字符串，算法首先在每两个字符（包括头尾）之间加没出现的字符（如 *），这样所有字符串长度就都是奇数了，简化了问题。
例如: $abcde \longrightarrow *a*b*c*d*e*$

(2) 具体 dp 过程
记录 p_i 表示 i 能向两边推（包括 i）的最大距离，如果能求出 p 数组，那每一个回文串就都确定了。

我们假设 $p[1\ i]$ 已经求好了，现在要求 $p[i]$。假设之前能达到的最右边为 R，对应的中点为 pos，j 是 i 的对称点。
1. 第一种情况，$i + p[j] < R$，即 $p[i] = p[j]$。
2. 第二种情况，$i + p[j] \geq R$，先设 $p[i] = R - i$ ，然后再继续增加 $p[i]$，并令 $pos = i$，更新 R。

由于 R 一定是递增的，因此时间复杂度为 $O(n)$，可以发现一个串本质不同的回文串最多有 n 个，因此只有 R 增加的时候才会产生本质不同的回文串。

(3) 算法特点
1. $Manacher$ 充分利用了回文的性质，从而达到线性时间。
2. $Manacher$ 右端点递增过程中产生了所有的本质不同回文串，即一个串中本质不同回文串最多只有 n 个。
3. $Manacher$ 算法的核心在于求出每一个节点往两边推的最远距离，所有涉及该算法的问题也都是在这个功能上做文章。

```
void Manacher(char s1[]){
    int len, tot, R = 0, pos = 0;
    //对字符串加'#'号
    len = strlen(s1+1);
    s2[0] = '$';
    s2[ln=1]='#';
    for(int i = 1; i <= len; i++)
      s2[++ln] = s1[i], s2[++ln] = '#';
    //求p[i]数组
    for(int i = 1; i <= ln; i++){
      if(R >= i) p[i] = min(p[pos*2-i],R-i+1);
        if(p[i] + i > R){
        for(; s2[R+1] == s2[i*2-R-1] && R+1 <= ln && i*2-R-1 <= ln; R++); //小心多组数据
        pos = i;
        p[i] = R-i+1;
      }
    }
}
```

4.23.2 回文自动机模板

一、适用问题:

回文自动机其实可以当作回文串问题的大杀器，主要可以解决以下问题:
1. 本质不同回文串个数
2. 每个本质不同回文串出现的次数
3. 以某一个节点为结尾的回文串个数
4. ...

而且都是在 O(n) 的复杂度内解决了这些问题。但是要注意，回文自动机并不能完全替代马拉车算法，因为马拉车针对的是以一个节点为中心所能扩展的最远的距离，而回文自动机更多的是处理以一个节点为结尾的回文串数量。

二、回文自动机算法解析

(1) 基础思想
与 AC 自动机一样，回文自动机的构造也使用了 $Trie$ 树的结构。

不同点在于回文自动机有两个根，一个是偶数Ⓕ度回文串的根，一个是奇数Ⓕ度回文串的根，简称为奇根和偶根。自动机中每个节点表示一个回文串，且都有一个 $fail$ 指针，指向当前节点所表示的回文串的最长回文后缀（不包括其本身）。

(2) 具体构建过程
首先将偶根的 $fail$ 指针指向奇根，奇根的 $fail$ 指针指向偶根，然后令奇根的长度为 -1，偶根长度为 1。

在构建过程中，始终要记录一个 $last$ 指针，表示上一次插入之后所位于的回文自动机节点。然后判断插入当前节点之后是否能够形成一个新的回文串，如果不能则跳 $fail$ 指针，整体逻辑与 AC 自动机的构建没有太大差别。

$fail$ 指针表示当前节点所代表的回文串的最长后缀回文串，此处 $fail$ 指针的定义与 AC 自动机有一定区别，但本质目的相同，都是为了尽可能地保存之前的匹配结果。

而 $fail$ 指针的构建是根据当前节点的父节点的 $fail$ 节点构建而来，与 AC 自动机的 $fail$ 构建没有太大差别。

大致上就是这样一个构建过程，具体细节可以查看下述模板，也可以查看 oi_wiki 的图解构建。

(3) 统计每个本质不同回文串的出现次数
与 AC 自动机一样，每个节点所代表字符串出现的次数要加上其所有 $fail$ 子孙出现的次数，因此需要倒序遍历所有节点将节点出现次数累加到其 $fail$ 节点上。

```cpp
#include <bits/stdc++.h>
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define per(i,a,b) for(int i = a; i >= b; i--)
const int N = 1e5+10;
using namespace std;

struct PAM{
    #define KIND 26
    int n,last,tot;
    int len[N],trie[N][KIND],fail[N],cnt[N],S[N],num[N];
    //len[i]: 节点i所代表的回文串长度, fail[i]: 当前回文串的最长回文后缀（不包括自身）
    //cnt[i]: 节点i所代表的回文串的个数, S[i]: 第i次添加的字符, num[i]: 以第i个字符为结尾的回文串个数
    //last: 上一个字符构成最长回文串的位置，方便下一个字符的插入
    //tot: 总结点个数 = 本质不同的回文串的个数+2, n: 插入字符的个数
    int newnode(int l){
        rep(i,0,KIND-1) trie[tot][i] = 0;
        cnt[tot] = 0, len[tot] = l, num[tot] = 0;
        return tot++;
    }
    inline void init(){
        tot = n = last = 0, newnode(0), newnode(-1);
        S[0] = -1, fail[0] = 1;
    }
    int get_fail(int x){ //获取fail指针
        while(S[n-len[x]-1] != S[n]) x = fail[x];
        return x;
    }
    inline int insert(int c){ //插入字符
        c -= 'a';
        S[++n] = c;
        int cur = get_fail(last);
        //在节点cur前的字符与当前字符相同，即构成一个回文串
        if(!trie[cur][c]){ //这个回文串没有出现过
```

```
            int now = newnode(len[cur]+2);
            fail[now] = trie[get_fail(fail[cur])][c];
            trie[cur][c] = now;
            num[now] = num[fail[now]]+1; //更新以当前字符为结尾的回文串的个数
        }
        last = trie[cur][c];
        cnt[last]++; //更新当前回文串的个数
        return num[last]; //返回以当前字符结尾的回文串的个数
    }
    void count(){ //统计每个本质不同回文串的个数
        per(i,tot-1,0) cnt[fail[i]] += cnt[i];
    }
}pam;
```

4.23.3 模板题

题意:
n 个字符，找出其中所有的奇长度回文串，按长度降序排列，求出前 k 个回文串长度的乘积。$(1 \le n \le 10^6, 1 \le k \le 10^{12})$

思路:
模板题。直接对于所有本质不同回文串求出其个数，然后用 *pair* 存到 *vector* 中，再利用快速幂计算答案即可。

```
#include <bits/stdc++.h>
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define per(i,a,b) for(int i = a; i >= b; i--)
typedef long long ll;
const int N = 1e6+1000;
const ll mod = 19930726;
using namespace std;

struct PAM{
    #define KIND 26
    int n,last,tot;
    int len[N],trie[N][KIND],fail[N],S[N],num[N];
    ll cnt[N];
    //len[i]: 节点i所代表的回文串长度, fail[i]: 当前回文串的最长回文后缀 (不包括自身)
    //cnt[i]: 节点i所代表的回文串的个数, S[i]: 第i次添加的字符, num[i]: 以第i个字符为结尾的回文串个数
    //last: 上一个字符构成最长回文串的位置, 方便下一个字符的插入
    //tot: 总结点个数 = 本质不同的回文串的个数+2, n: 插入字符的个数
    int newnode(int l){
        rep(i,0,KIND-1) trie[tot][i] = 0;
        cnt[tot] = 0, len[tot] = l, num[tot] = 0;
        return tot++;
    }
    inline void init(){
        tot = n = last = 0, newnode(0), newnode(-1);
        S[0] = -1, fail[0] = 1;
    }
    int get_fail(int x){ //获取fail指针
        while(S[n-len[x]-1] != S[n]) x = fail[x];
        return x;
    }
    inline int insert(int c){ //插入字符
        c -= 'a';
        S[++n] = c;
        int cur = get_fail(last);
```

```
        //在节点cur前的字符与当前字符相同，即构成一个回文串
        if(!trie[cur][c]){ //这个回文串没有出现过
            int now = newnode(len[cur]+2);
            fail[now] = trie[get_fail(fail[cur])][c];
            trie[cur][c] = now;
            num[now] = num[fail[now]]+1; //更新以当前字符为结尾的回文串的个数
        }
        last = trie[cur][c];
        cnt[last]++; //更新当前回文串的个数
        return num[last]; //返回以当前字符结尾的回文串的个数
    }
    void count(){ //统计每个本质不同回文串的个数
        per(i,tot-1,0) cnt[fail[i]] += cnt[i];
    }
}pam;

int n;
ll k;
char buf[N];
vector<pair<int,ll> > v;

ll pow_mod(ll a,ll b){
    ll ans = 1, base = a;
    while(b){
        if(b&1) ans = (ans*base)%mod;
        base = (base*base)%mod;
        b /= 2ll;
    }
    return ans;
}

int main(){
    scanf("%d%lld",&n,&k);
    scanf("%s",buf);
    pam.init();
    rep(i,0,n-1) pam.insert(buf[i]);
    pam.count();
    ll num = 0;
    rep(i,0,pam.tot-1)
        if(pam.len[i] > 0 && pam.len[i]%2 == 1){
            v.push_back(make_pair(pam.len[i],pam.cnt[i]));
            num += pam.cnt[i];
        }
    sort(v.begin(),v.end());
    if(num < k) printf("-1\n");
    else{
        ll ans = 1;
        int pos = v.size()-1;
        while(k > 0){
            if(k >= v[pos].second){
                ans = (ans*pow_mod(v[pos].first,v[pos].second))%mod;
                k -= v[pos].second;
            }
            else{
                ans = (ans*pow_mod(v[pos].first,k))%mod;
                k = 0;
            }
            pos--;
        }
```

```
        printf("%lld\n",ans);
    }
    return 0;
}
```

4.23.4 Manacher 例题

题意:
给定一个长度为 n 的字符串，询问该字符串中长度最长的双倍回文串的长度。双倍回文串指 AA^TAA^T，如 $abbaabba$ 即是双倍回文串，而 $ababa$ 则不是。$(1 \le n \le 5 * 10^5)$

思路:
由于只需要求最长回文串，并不要求输出个数，因此我们直接在 $Manacher$ 的 dp 过程中求出答案即可。

首先明确一点，$Manacher$ 右端点递增过程中，涉及到了所有的本质不同回文串，因此我们在每次右端点递增时，令当前节点为双倍回文串的中心节点，然后查询左半部分是否也为一个回文串，如果是则更新答案。

变型:
当然此题也可以进行延伸，可以将查询内容改成有多少个双倍回文串。如果询问有多少个双倍回文串的话，我的想法是用回文自动机 + $Manacher$ 算法进行解决。

首先 $Manacher$ 求出每一个节点为中心所能扩展的最远距离。然后在根据当前字符串建立回文自动机，并且在构建时对于每个节点记录构成该节点的末尾坐标 pos，之后再遍历回文自动机上的所有节点，利用 $Manacher$ 计算的 p 数组来验证该节点所代表的回文串是否是双倍回文串即可。

```
#include <bits/stdc++.h>
const int N = 5e5+100;
using namespace std;

int n,tot,p[2*N],R,pos,ans;
char s[N],s2[2*N];

void manacher(){
    s2[0] = '$';
    s2[tot = 1] = '#';
    for(int i = 1; i <= n; i++)
        s2[++tot] = s[i], s2[++tot] = '#';
    for(int i = 1; i <= tot; i++){
        if(R >= i) p[i] = min(p[pos*2-i],R-i+1);
        if(p[i] + i > R){
            for(; s2[R+1] == s2[i*2-R-1]; R++){
                int y = (3*i-R-1)/2;
                if(s2[i] == '#' && s2[R+1] == '#' && (R+1-i)%4 == 0 && p[y] >= i-y+1)
    ans = max(ans,2*(i-y));
            }
            pos = i;
            p[i] = R-i+1;
        }
    }
    printf("%d\n",ans);
}

int main(){
    scanf("%d",&n); scanf("%s",s+1);
    manacher();
}
```

4.23.5 Manacher+ 回文自动机

题意:
给定一个长度为 n 的字符串，先可以选择该字符串中的一个节点，将其替换成 & 符号，问替换之后该字符串中本质不同回文串个数。$(1 \le n \le 5 * 10^5)$

思路:
一个比较明显的考虑方式就是枚举替换的位置，然后计算该位置改变后增加的回文串数量与减少的回文串数量。

首先我们考虑增加的回文串数量，显然就是以该节点为中心所形成的回文串总数，因此我们直接用马拉车的 p 数组进行求解即可。

然后再来考虑减少的回文串数量。我们对于回文自动机中所有本质不同回文串的末尾出现位置记录一个 mx 和 mi，表示最远出现的位置和最早出现的位置，记该回文串长度为 len。则如果 $mx - len + 1 \le mi$ ，则修改位置出现在 $[mx - len + 1, mi]$ 中时，本质不同回文串数量就会减 1，因此我们维护一个差分数组，$sum[mx - len + 1]$ 加 1，$sum[mi]$ 减 1，然后对于每个自动机上的节点都这样处理一遍即可。

最后再枚举替换点统计答案。

```cpp
#include <bits/stdc++.h>
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define per(i,a,b) for(int i = a; i >= b; i--)
const int N = 1e5+10;
using namespace std;

char buf[N],s2[2*N];
int sum[N],p[2*N];

void dbg() {cout << "\n";}
template<typename T, typename... A> void dbg(T a, A... x) {cout << a << ' '; dbg(x...);}
#define logs(x...) {cout << #x << " -> "; dbg(x);}

struct PAM{
    #define KIND 26
    int n,last,tot;
    int len[N],trie[N][KIND],fail[N],cnt[N],S[N],num[N],mx[N],mi[N];
    //len[i]: 节点i所代表的回文串长度, fail[i]: 当前回文串的最长回文后缀 (不包括自身)
    //cnt[i]: 节点i所代表的回文串的个数, S[i]: 第i次添加的字符, num[i]: 以第i个字符为结尾的回文串个数
    //last: 上一个字符构成最长回文串的位置, 方便下一个字符的插入
    //tot: 总结点个数 = 本质不同的回文串的个数+2, n: 插入字符的个数
    int newnode(int l){
        rep(i,0,KIND-1) trie[tot][i] = 0;
        cnt[tot] = 0, len[tot] = l, num[tot] = 0, mx[tot] = 0, mi[tot] = 2*1e5;
        return tot++;
    }
    inline void init(){
        tot = n = last = 0, newnode(0), newnode(-1);
        S[0] = -1, fail[0] = 1;
    }
    int get_fail(int x){ //获取fail指针
        while(S[n-len[x]-1] != S[n]) x = fail[x];
        return x;
    }
    inline int insert(int c,int pos){ //插入字符
        c -= 'a';
        S[++n] = c;
```

```
        int cur = get_fail(last);
        //在节点cur前的字符与当前字符相同，即构成一个回文串
        if(!trie[cur][c]){ //这个回文串没有出现过
            int now = newnode(len[cur]+2);
            fail[now] = trie[get_fail(fail[cur])][c];
            trie[cur][c] = now;
            num[now] = num[fail[now]]+1; //更新以当前字符为结尾的回文串的个数
        }
        last = trie[cur][c];
        cnt[last]++; //更新当前回文串的个数
        mx[last] = max(mx[last],pos);
        mi[last] = min(mi[last],pos); //更新每个回文串的最早/晚出现位置
        return num[last]; //返回以当前字符结尾的回文串的个数
    }
    void count(){ //统计每个本质不同回文串的个数
        per(i,tot-1,2){
            cnt[fail[i]] += cnt[i];
            mx[fail[i]] = max(mx[fail[i]],mx[i]);
            mi[fail[i]] = min(mi[fail[i]],mi[i]);
            if(mi[i] >= mx[i]-len[i]+1){
                sum[mx[i]-len[i]+1] += 1;
                sum[mi[i]+1] += -1;
            }
        }
    }
}pam;

void manacher(){
    int len = strlen(buf+1), tot = 1, R = 0, pos = 0;
    s2[0] = '$'; s2[tot = 1] = '#';
    for(int i = 1; i <= len; i++)
        s2[++tot] = buf[i], s2[++tot] = '#';
    for(int i = 1; i <= tot; i++){
        if(R >= i) p[i] = min(p[pos*2-i],R-i+1);
        if(p[i] + i > R){
            for(; s2[R+1] == s2[i*2-R-1] && R+1 <= tot && i*2-R-1 <= tot; R++);
            pos = i;
            p[i] = R-i+1;
        }
    }
}

int main(){
    int _; scanf("%d",&_);
    while(_--){
        int ans = 0, ct = 0;
        scanf("%s",buf+1);
        pam.init();
        int len = strlen(buf+1);
        rep(i,0,len) sum[i] = 0;
        rep(i,1,len) pam.insert(buf[i],i);
        pam.count();
        manacher();
        rep(i,1,len){
            sum[i] += sum[i-1];
            int to = pam.tot-2-sum[i]+((p[i*2]-2)/2+1);
            if(ans < pam.tot-2-sum[i]+((p[i*2]-2)/2+1)){
                ans = pam.tot-2-sum[i]+((p[i*2]-2)/2+1);
                ct = 1;
```

```
            }
            else if(ans == pam.tot-2-sum[i]+((p[i*2]-2)/2+1)) ct++;
        }
        printf("%d %d\n",ans,ct);
    }
    return 0;
}
```

4.23.6 两颗回文自动机

题意:
给定两个字符串，询问满足条件的四元组个数，(x, y, u, v) 满足条件当且仅当 $s[x...y] = h[u...v]$，且 $s[x...y]$ 为回文串。$(1 \le n \le 10^5)$

思路:
建两颗回文自动机，然后从奇根和偶根分别遍历，同时遍历两棵树，如果到达相同位置则计算对答案的贡献，也算是比较裸的一道题目。

```
#include <bits/stdc++.h>
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define per(i,a,b) for(int i = a; i >= b; i--)
const int N = 4*1e5+10;
typedef long long ll;
using namespace std;

char buf1[N],buf2[N];
ll ans = 0;

struct PAM{
    #define KIND 26
    int n,last,tot;
    int len[N],trie[N][KIND],fail[N],S[N],num[N];
    ll cnt[N];
    //len[i]: 节点i所代表的回文串长度, fail[i]: 当前回文串的最长回文后缀 (不包括自身)
    //cnt[i]: 节点i所代表的回文串的个数, S[i]: 第i次添加的字符, num[i]: 以第i个字符为结尾的回文串个数
    //last: 上一个字符构成最长回文串的位置, 方便下一个字符的插入
    //tot: 总结点个数 = 本质不同的回文串的个数+2, n: 插入字符的个数
    int newnode(int l){
        rep(i,0,KIND-1) trie[tot][i] = 0;
        cnt[tot] = 0, len[tot] = l, num[tot] = 0;
        return tot++;
    }
    inline void init(){
        tot = n = last = 0, newnode(0), newnode(-1);
        S[0] = -1, fail[0] = 1;
    }
    int get_fail(int x){ //获取fail指针
        while(S[n-len[x]-1] != S[n]) x = fail[x];
        return x;
    }
    inline int insert(int c){ //插入字符
        c -= 'a';
        S[++n] = c;
        int cur = get_fail(last);
        //在节点cur前的字符与当前字符相同, 即构成一个回文串
        if(!trie[cur][c]){ //这个回文串没有出现过
```

```
            int now = newnode(len[cur]+2);
            fail[now] = trie[get_fail(fail[cur])][c];
            trie[cur][c] = now;
            num[now] = num[fail[now]]+1; //更新以当前字符为结尾的回文串的个数
        }
        last = trie[cur][c];
        cnt[last]++; //更新当前回文串的个数
        return num[last]; //返回以当前字符结尾的回文串的个数
    }
    void count(){ //统计每个本质不同回文串的个数
        per(i,tot-1,2) cnt[fail[i]] += cnt[i];
    }
}pam[2];

void dfs(int t1,int t2){
    rep(i,0,KIND-1){
        int y1 = pam[0].trie[t1][i], y2 = pam[1].trie[t2][i];
        if(y1 && y2){
            dfs(y1,y2);
            ans = ans+pam[0].cnt[y1]*pam[1].cnt[y2];
        }
    }
}

int main(){
    int _; scanf("%d",&_);
    rep(Ca,1,_){
        ans = 0;
        pam[0].init(); pam[1].init();
        scanf("%s",buf1); scanf("%s",buf2);
        int len1 = strlen(buf1), len2 = strlen(buf2);
        rep(i,0,len1-1) pam[0].insert(buf1[i]);
        rep(i,0,len2-1) pam[1].insert(buf2[i]);
        pam[0].count(); pam[1].count();
        dfs(0,0); dfs(1,1);
        printf("Case #%d: %lld\n",Ca,ans);
    }
    return 0;
}
```

4.23.7 马拉车 + 字典树

题意:
给出 n 个字符串，求解在其中 $n*n$ 个拼接中，回文串的数量。($\sum len(i) \le 2*10^6$)

思路:
考虑马拉车 + 字典树解决该问题。

两个串拼在一起是回文串，即 s 与 h 拼接之后为回文串，主要有两种情况。
1. $|s| \le |h|$，则将 s 与 h 的反串进行匹配，h 中剩下的未匹配部分为回文串。
2. $|h| \le |s|$，则将 s 与 h 的反串进行匹配，s 中剩下的未匹配部分为回文串。

分类完之后即可用马拉车算法求出回文串范围，然后用字典树进行反串匹配。

```
#include <iostream>
#include <cstdio>
#include <algorithm>
```

```cpp
#include <cstring>
#include <vector>
#define mem(a,b) memset(a,b,sizeof a);
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define per(i,a,b) for(int i = a; i >= b; i--)
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
typedef long long ll;
typedef double db;
const int N = 2e6+1000;
const db EPS = 1e-9;
using namespace std;

int n,trie[N][26],tot,p[2*N],flag[N],L[N],sum1[N],sum2[N];
char s[N],s1[N],s2[2*N];
ll ans = 0;

void init(char base[],int len,int op = 0){
    //manacher
    int ln = 1, R = 0, pos = 0;
    rep(i,1,len) s1[i] = base[i-1];
    s2[0] = '$'; s2[ln = 1] = '#';
    for(int i = 1; i <= len; i++)
        s2[++ln] = s1[i], s2[++ln] = '#';
    for(int i = 1; i <= ln; i++){
        if(R >= i) p[i] = min(p[pos*2-i],R-i+1);
        if(p[i]+i > R){
            for(; s2[R+1] == s2[i*2-R-1] && R+1 <= ln && i*2-R-1 <= ln; R++);
            pos = i;
            p[i] = R-i+1;
        }
    }
    //预处理
    if(op){
        for(int i = 1; i <= len; i++){
            int tp = i+(len-i)/2;
            flag[i] = 0;
            tp *= 2;
            if((len-i) % 2 == 1) tp += 2;
            else tp++;
            if(p[tp] >= ln-tp+1) flag[i] = 1;
        }
        return;
    }
    for(int i = 1; i <= len; i++){
        int tp = i-(i-1)/2;
        flag[i] = 0;
        tp *= 2;
        if((i-1)%2 == 1) tp -= 2;
        else tp--;
        if(i != 1 && p[tp] >= tp) flag[i] = 1;
    }
    //插入字典树
    int now = 0;
    for(int i = len; i >= 1; i--){
        if(!trie[now][s1[i]-'a']){
            trie[now][s1[i]-'a'] = ++tot, sum1[tot] = sum2[tot] = 0;
            rep(j,0,25) trie[tot][j] = 0;
        }
        now = trie[now][s1[i]-'a'];
```

```
        if(flag[i]) sum2[now]++; //后半部分回文
        if(i == 1) sum1[now]++; //末尾
    }
}

void solve(char base[],int len){
    init(base,len,1);
    rep(i,1,len) s1[i] = base[i-1];
    int now = 0;
    for(int i = 1; i <= len; i++){
        if(!trie[now][s1[i]-'a']) return;
        now = trie[now][s1[i]-'a'];
        if(i == len) ans += sum2[now]+sum1[now];
        else if(flag[i]) ans += sum1[now];
    }
}

int main()
{
    while(~scanf("%d",&n)){
        tot = 0; ans = 0;
        rep(i,0,25) trie[0][i] = 0;
        int l = 0;
        rep(i,1,n){
            int tp; scanf("%d%s",&tp,s+l);
            init(s+l,tp);
            L[i] = l; L[i+1] = l+tp;
            l += tp;
        }
        ans = 0;
        rep(i,1,n) solve(s+L[i],L[i+1]-L[i]);
        printf("%lld\n",ans);
    }
    return 0;
}
```

4.24 后缀数组

4.24.1 后缀数组模板

一、适用问题:

后缀数组的题目非常灵活多变，主要涉及字符串所有后缀的字典序比较以及最长公共前缀。

本文主要介绍后缀数组的一些经典应用，虽然是经典应用，但是其思想应该属于后缀数组类问题的本质思想。

二、算法介绍:

求解后缀数组的算法主要有倍增法、$DC3$ 算法，具体的算法实现此处就略过了，想要具体了解的话可以自行 $google$ 搜索。

下文给出的后缀数组模板是 $DC3$ 算法，时间复杂度为 $O(n)$。
(1) 后缀数组中各大数组介绍
1. $suffix[i]$: 以 i 为起始位置的后缀
2. $sa[i]$: 排名第 i 的后缀的起始位置
3. $rak[i]$: 表示 $suffix[i]$ 的排名
4. $height[i]$: $suffix(sa[i-1])$ 和 $suffix(sa[i])$ 的最长公共前缀
5. $h[i] = height[rak[i]]$，$h[i] \geq h[i-1]-1$

有了上述数组，我们便可以求出字符串任意两个后缀的最长公共前缀。
$suffix[i]$ 和 $suffix[j]$ 之间的最长公共前缀 $= min(height[rak[i]+1], ..., height[rak[j]])$

后缀数组所有的题目都是在这几个数组上做文章，具体的题目类型分类见下文例题。

```
#include <bits/stdc++.h>
const int N = 1e6+10;
using namespace std;

int height[N],sa[N],rak[N],len;
/*
    suffix[i]: 以i为起始位置的后缀
    sa[i]: 排名第i的后缀的起始位置
    rak[i]: 表示suffix[i]的排名
    height[i]: suffix(sa[i-1])和suffix(sa[i])的最长公共前缀
        · h[i] = height[rak[i]], h[i] >= h[i-1]-1
        · suffix[i]和suffix[j]之间的最长公共前缀 = min(height[rak[i]+1]...height[rak[j]])
*/

bool cmp(int *y,int a,int b,int k)
{
    int a1 = y[a],b1 = y[b];
    int a2 = a + k >= len ? -1 : y[a + k];
    int b2 = b + k >= len ? -1 : y[b + k];
    return a1 == b1 && a2 == b2;
}

int t1[N],t2[N],cc[N];

void get_sa(char s[])
{
    int *x = t1,*y = t2,m = 200;
    for(int i = 0;i < m;i ++) cc[i] = 0;
    for(int i = 0;i < len;i ++) ++ cc[x[i] = s[i]];
    for(int i = 1;i < m;i ++) cc[i] += cc[i - 1];
    for(int i = len - 1;~i;i --) sa[-- cc[x[i]]] = i;
    for(int k = 1;k < len;k <<= 1)
    {
        int p = 0;
        for(int i = len - k;i < len;i ++)  y[p ++] = i;
        for(int i = 0;i < len;i ++) if(sa[i] >= k) y[p ++] = sa[i] - k;
        for(int i = 0;i < m;i ++) cc[i] = 0;
        for(int i = 0;i < len;i ++) ++ cc[x[y[i]]];
        for(int i = 1;i < m;i ++) cc[i] += cc[i - 1];
        for(int i = len - 1;~i;i --) sa[-- cc[x[y[i]]]] = y[i];
        swap(x,y); m = 1; x[sa[0]] = 0;

        for(int i = 1;i < len;i ++)
            x[sa[i]] = cmp(y,sa[i - 1],sa[i],k) ? m - 1 : m ++;
        if(m >= len) break;
    }
}

void get_height(char s[])
{
    for(int i = 0;i < len;i ++) rak[sa[i]] = i;
    int h = 0;
    height[0] = 0;
    for(int i = 0;i < len;i ++)
```

```
    {
        if(!rak[i]) continue;
        int j = sa[rak[i] - 1];
        if(h) h --;
        while(s[i + h] == s[j + h]) h ++;
        height[rak[i]] = h;
    }
}

char s[N];

int main(){
    scanf("%s",s);
    len = strlen(s);
    get_sa(s);
    get_height(s);
    return 0;
}
```

4.24.2 单字符串常见问题

一、最长公共前缀

题意: 给定一个字符串，询问某两个后缀的最长公共前缀。

思路: 模板题。对字符串求出后缀数组后，用线段树或 st 表进行 rmq 即可。

二、可重叠最长重复子串

题意: 给定一个字符串，求最长重复子串，这两个子串可以重叠。

思路: 模板题。即求字符串中任意两个后缀最长公共前缀的最大值，即输出 $height$ 数组中的最大值即可。

三、不可重叠最长重复子串

题意: 给定一个字符串，求最长不可重复子串，这两个子串不可以重叠。

思路: 这个题与上面那个题最大的区别在于，此题找到的两个子串不可以重叠。因此我们在此处引入后缀数组中一个重要的思想，二分 + $height$ 分组。

我们二分一个答案 w，然后根据这个答案对 $height$ 数组进行分组，即每一组中的 $height$ 数组值大于 w。然后记录一个 $maxx$ 和 $minn$ 分别表示该组中所有后缀起始位置的最大值和最小值，然后判断两个位置之间的差是否大于等于 w，如果大于等于则 w 可行。

四、可重叠的 k 次最长重复子串
题意: 给定一个字符串，求至少出现 k 次的最长重复子串，这 k 个子串可以重叠。

思路: 利用上面那个题的 == 二分 + $height$ 分组 == 思想可以轻松解决此题。即分完组之后判断该组内的字符串数是否大于 w 即可。

五、不相同的子串的个数

题意: 给定一个字符串，求不相同的子串个数。

思路: 求出该字符串的后缀数组，然后遍历 $height$ 数组，排名为 i 的后缀的起始位置为 $sa[i]$，如果可以相同的话，对答案的贡献为 $n-sa[i]+1$。但是现在要求不相同的子串个数，因此要减去最长的相同前缀，即贡献为 $n-sa[i]+1-height[i]$。

六、连续重复子串 [整个串]

题意: 给定一个字符串 L，已知这个字符串是由某个字符串 S 重复 R 次得到的，求 R 的最大值。

思路: 设 LEN 为字符串 L 的长度。枚举字符串 S 的长度 len，保证 len 能够被 LEN 整除。然后再判断 $suffix[0]$ 与 $suffix[len]$ 的最长公共前缀长度是否为 $LEN - len$，如果是则表示该长度符合答案。

这里求后缀最长公共前缀时不需要 rmq，由于 $suffix[0]$ 是固定的，因此只需要求出 $height[i]$ 数组到 $height[rak[0]]$ 的最小值即可，总时间复杂度为 $O(n)$。

七、重复次数最多的连续重复子串 [子串]
题意: 给定一个字符串 L，在该字符串的所有子串中找到一个重复度最多的子串，重复度相同则输出字典序最小。重复度即上题定义的重复次数，例如"$ababab$" 重复次数为 3，"$abcd$" 重复次数为 1。

思路: 我们枚举 len，去求解长度为 len 下最大的重复度 R，即有无一个子串由 R 个长度为 len 的子串连续拼接而成。

因此我们将字符串 L 分段，$0\ len\ 2*len\ 3*len\ ...\ k*len$，对于 $suffix[i*len]$ 与 $suffix[(i+1)*len]$ 求出最长公共前缀 x，再另 $pos = i*len - (len - x\%len)$，求 $suffix[pos]$ 与 $suffix[pos+len]$ 的最长公共前缀，然后更新答案。

这样操作的原因其实是枚举所有的以长度 len 进行重复的子串，这也是对 len 取模数的原因。可能这样说还是有些难以理解，推荐自己静下来想一想，或者看看代码。

最后分析下时间复杂度，$\frac{n}{1} + \frac{n}{2} + \frac{n}{3} + ... + \frac{n}{n} = O(nlogn)$。

```
#include <iostream>
#include <cstdio>
#include <algorithm>
#include <cstring>
#include <cmath>
#define mem(a,b) memset(a,b,sizeof a);
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define per(i,a,b) for(int i = a; i >= b; i--)
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
typedef long long ll;
typedef double db;
const int N = 1e5+10;
const db EPS = 1e-9;
using namespace std;

void dbg() {cout << "\n";}
template<typename T, typename... A> void dbg(T a, A... x) {cout << a << ' '; dbg(x...)
    ;}
#define logs(x...) {cout << #x << " -> "; dbg(x);}

int n,height[N],sa[N],rak[N],len; //所有数组都从0开始计数
/*
    suffix[i]: 以i为起始位置的后缀
    sa[i]: 排名第i的后缀的起始位置
    rak[i]: 表示suffix[i]的排名
    height[i]: suffix(sa[i-1])和suffix(sa[i])的最长公共前缀
        · h[i] = height[rak[i]], h[i] >= h[i-1]-1
        · suffix[i]和suffix[j]之前的最长公共前缀 = min(height[rak[i]+1]...height[rak[j]])
*/

bool cmp(int *y,int a,int b,int k)
{
    int a1 = y[a],b1 = y[b];
    int a2 = a + k >= len ? -1 : y[a + k];
    int b2 = b + k >= len ? -1 : y[b + k];
    return a1 == b1 && a2 == b2;
```

```
}

int t1[N],t2[N],cc[N];

void get_sa(char s[])
{
    int *x = t1,*y = t2,m = 200;
    len = strlen(s);
    for(int i = 0;i < m;i ++) cc[i] = 0;
    for(int i = 0;i < len;i ++) ++ cc[x[i] = s[i]];
    for(int i = 1;i < m;i ++) cc[i] += cc[i - 1];
    for(int i = len - 1;~i;i --) sa[-- cc[x[i]]] = i;
    for(int k = 1;k < len;k <<= 1)
    {
        int p = 0;
        for(int i = len - k;i < len;i ++)  y[p ++] = i;
        for(int i = 0;i < len;i ++) if(sa[i] >= k) y[p ++] = sa[i] - k;
        for(int i = 0;i < m;i ++) cc[i] = 0;
        for(int i = 0;i < len;i ++) ++ cc[x[y[i]]];
        for(int i = 1;i < m;i ++) cc[i] += cc[i - 1];
        for(int i = len - 1;~i;i --) sa[-- cc[x[y[i]]]] = y[i];
        swap(x,y); m = 1; x[sa[0]] = 0;

        for(int i = 1;i < len;i ++)
            x[sa[i]] = cmp(y,sa[i - 1],sa[i],k) ? m - 1 : m ++;
        if(m >= len) break;
    }
}

void get_height(char s[])
{
    len = strlen(s);
    for(int i = 0;i < len;i ++) rak[sa[i]] = i;
    int h = 0;
    height[0] = 0;
    for(int i = 0;i < len;i ++)
    {
        if(!rak[i]) continue;
        int j = sa[rak[i] - 1];
        if(h) h --;
        while(s[i + h] == s[j + h]) h ++;
        height[rak[i]] = h;
    }
}

char s[N];

int st[N][20];

void init(){
    for(int i = 0; i < len; i++) st[i][0] = height[i];
    for(int j = 1; (1<<j) <= len; j++){
        for(int i = 0; i + (1<<j) - 1 < len; i++)
            st[i][j] = min(st[i][j-1],st[i+(1<<(j-1))][j-1]);
    }
}

int query(int l,int r){
    int k = (int)(log((double)(r - l + 1)) / log(2.0));
```

```
    return min(st[l][k],st[r-(1<<k)+1][k]);
}

int main(){
    int _ = 0;
    while(~scanf("%s",s)){
        if(s[0] == '#') break;
        len = strlen(s);
        get_sa(s);
        get_height(s);
        init();
        int ans = 0, LEN = 0;
        rep(L,1,len){
            for(int j = L; j < len; j += L){
                int p1 = rak[j];
                int p2 = rak[j-L];
                if(p1 > p2) swap(p1,p2);
                int thp = query(p1+1,p2);
                if((thp/L)+1 > ans)
                    ans = (thp/L)+1, LEN = L;

                thp = L-thp%L;
                if(j-L-thp < 0) continue;
                p1 = rak[j-thp], p2 = rak[j-L-thp];
                if(p1 > p2) swap(p1,p2);
                thp = query(p1+1,p2);
                if((thp/L)+1 > ans)
                    ans = (thp/L)+1, LEN = L;
            }
        }
        int pos = -1;
        rep(i,0,len-1-LEN){
            int p1 = rak[i], p2 = rak[i+LEN];
            if(p1 > p2) swap(p1,p2);
            int thp = query(p1+1,p2);
            if((thp/LEN)+1 == ans){
                if(pos == -1) pos = i;
                else if(rak[i] < rak[pos]) pos = i;
            }
        }
        s[pos+LEN*ans] = '\0';
        printf("Case %d: %s\n",++_,s+pos);
    }
    return 0;
}
```

4.24.3 最长公共子串 (双串)

题意:
给定两个字符串 A 和 B，求最长公共子串。

思路:
这里介绍一下后缀数组的另一个套路，即涉及两个或多个串时，往往会将两个串用一个没出现的符号（比如'$'），将两个串连接在一起。

比如此题即可将两个串拼接在一起，求出后缀数组之后，枚举 $height$ 数组，判断 $height$ 数组中相邻的两个后缀是否来自不同的串，如果是则更新答案。

```cpp
#include <iostream>
#include <cstdio>
#include <algorithm>
#include <cstring>
#include <cmath>
#define mem(a,b) memset(a,b,sizeof a);
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define per(i,a,b) for(int i = a; i >= b; i--)
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
typedef long long ll;
typedef double db;
const int N = 2*1e5+10;
const db EPS = 1e-9;
using namespace std;

void dbg() {cout << "\n";}
template<typename T, typename... A> void dbg(T a, A... x) {cout << a << ' '; dbg(x...);}
#define logs(x...) {cout << #x << " -> "; dbg(x);}

int n,height[N],sa[N],rak[N],len; //所有数组都从0开始计数
/*
    suffix[i]: 以i为起始位置的后缀
    sa[i]: 排名第i的后缀的起始位置
    rak[i]: 表示suffix[i]的排名
    height[i]: suffix(sa[i-1])和suffix(sa[i])的最长公共前缀
        · h[i] = height[rak[i]], h[i] >= h[i-1]-1
        · suffix[i]和suffix[j]之前的最长公共前缀 = min(height[rak[i]+1]...height[rak[j]])
*/

bool cmp(int *y,int a,int b,int k)
{
    int a1 = y[a],b1 = y[b];
    int a2 = a + k >= len ? -1 : y[a + k];
    int b2 = b + k >= len ? -1 : y[b + k];
    return a1 == b1 && a2 == b2;
}

int t1[N],t2[N],cc[N];

void get_sa(char s[])
{
    int *x = t1,*y = t2,m = 200;
    len = strlen(s);
    for(int i = 0;i < m;i ++) cc[i] = 0;
    for(int i = 0;i < len;i ++) ++ cc[x[i] = s[i]];
    for(int i = 1;i < m;i ++) cc[i] += cc[i - 1];
    for(int i = len - 1;~i;i --) sa[-- cc[x[i]]] = i;
    for(int k = 1;k < len;k <<= 1)
    {
        int p = 0;
        for(int i = len - k;i < len;i ++)  y[p ++] = i;
        for(int i = 0;i < len;i ++) if(sa[i] >= k) y[p ++] = sa[i] - k;
        for(int i = 0;i < m;i ++) cc[i] = 0;
        for(int i = 0;i < len;i ++) ++ cc[x[y[i]]];
        for(int i = 1;i < m;i ++) cc[i] += cc[i - 1];
        for(int i = len - 1;~i;i --) sa[-- cc[x[y[i]]]] = y[i];
        swap(x,y); m = 1; x[sa[0]] = 0;

```

```
        for(int i = 1;i < len;i ++)
            x[sa[i]] = cmp(y,sa[i - 1],sa[i],k) ? m - 1 : m ++;
        if(m >= len) break;
    }
}

void get_height(char s[])
{
    len = strlen(s);
    for(int i = 0;i < len;i ++) rak[sa[i]] = i;
    int h = 0;
    height[0] = 0;
    for(int i = 0;i < len;i ++)
    {
        if(!rak[i]) continue;
        int j = sa[rak[i] - 1];
        if(h) h --;
        while(s[i + h] == s[j + h]) h ++;
        height[rak[i]] = h;
    }
}

char s1[N],s2[N],s[N];

int main(){
    scanf("%s",s1);
    scanf("%s",s2);
    int len1 = strlen(s1), len2 = strlen(s2);
    rep(i,0,len1-1) s[i] = s1[i];
    s[len1] = '$';
    rep(i,0,len2-1) s[i+len1+1] = s2[i];
    len = strlen(s);
    get_sa(s);
    get_height(s);
    //[0,len1-1] [len1+1,len1+len2]
    int ans = 0;
    rep(i,1,len1+len2){
        int p1 = sa[i], p2 = sa[i-1];
        if(p1 <= len1-1 && p2 >= len1+1) ans = max(ans,height[i]);
        else if(p2 <= len1-1 && p1 >= len1+1) ans = max(ans,height[i]);
    }
    printf("%d\n",ans);
    return 0;
}
int n;
ll k;
char buf[N];
vector<pair<int,ll> > v;

ll pow_mod(ll a,ll b){
    ll ans = 1, base = a;
    while(b){
        if(b&1) ans = (ans*base)%mod;
        base = (base*base)%mod;
        b /= 2ll;
    }
    return ans;
}

```

```
int main(){
    scanf("%d%lld",&n,&k);
    scanf("%s",buf);
    pam.init();
    rep(i,0,n-1) pam.insert(buf[i]);
    pam.count();
    ll num = 0;
    rep(i,0,pam.tot-1)
        if(pam.len[i] > 0 && pam.len[i]%2 == 1){
            v.push_back(make_pair(pam.len[i],pam.cnt[i]));
            num += pam.cnt[i];
        }
    sort(v.begin(),v.end());
    if(num < k) printf("-1\n");
    else{
        ll ans = 1;
        int pos = v.size()-1;
        while(k > 0){
            if(k >= v[pos].second){
                ans = (ans*pow_mod(v[pos].first,v[pos].second))%mod;
                k -= v[pos].second;
            }
            else{
                ans = (ans*pow_mod(v[pos].first,k))%mod;
                k = 0;
            }
            pos--;
        }
        printf("%lld\n",ans);
    }
    return 0;
}
```

4.24.4 长度不小于 k 的公共子串个数

题意:
给定两个字符串 A 和 B，求长度不小于 k 的公共子串的个数，即询问有多少个三元组 (i, j, len) 表示 $A[i]\ A[i+len-1]$ 与 $B[j]\ B[j+len-1]$ 相同，且 $len \geq k$。

思路:
首先将两个串拼接起来，求出后缀数组之后，按 k 对 $height$ 数组进行分组。对于两个最长公共前缀为 h 的后缀，其对答案的贡献为 $h-k+1$，因此我们分组之后，先考虑 A 对 B 的贡献，维护一个单调栈，存储信息为 $pair(int, int)$ 表示 $height$ 的值与个数，每当有更小的值进入时就不断向前合并，主要是 $second$ 的信息合并，并且记录一个全局的 $total = \sum first * second$，用于计算贡献。

之后再计算一遍 B 对 A 的贡献即可结束此题。

```
#include <iostream>
#include <cstdio>
#include <algorithm>
#include <cstring>
#include <cmath>
#define mem(a,b) memset(a,b,sizeof a);
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define per(i,a,b) for(int i = a; i >= b; i--)
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
typedef long long ll;
```

```
typedef double db;
const int N = 2*1e5+10;
const db EPS = 1e-9;
using namespace std;

void dbg() {cout << "\n";}
template<typename T, typename... A> void dbg(T a, A... x) {cout << a << ' '; dbg(x...);}
#define logs(x...) {cout << #x << " -> "; dbg(x);}

int n,k,height[N],sa[N],rak[N],len; //所有数组都从0开始计数
/*
    suffix[i]: 以i为起始位置的后缀
    sa[i]: 排名第i的后缀的起始位置
    rak[i]: 表示suffix[i]的排名
    height[i]: suffix(sa[i-1])和suffix(sa[i])的最长公共前缀
        · h[i] = height[rak[i]], h[i] >= h[i-1]-1
        · suffix[i]和suffix[j]之前的最长公共前缀 = min(height[rak[i]+1]...height[rak[j]])
*/

bool cmp(int *y,int a,int b,int k)
{
    int a1 = y[a],b1 = y[b];
    int a2 = a + k >= len ? -1 : y[a + k];
    int b2 = b + k >= len ? -1 : y[b + k];
    return a1 == b1 && a2 == b2;
}

int t1[N],t2[N],cc[N];

void get_sa(char s[])
{
    int *x = t1,*y = t2,m = 200;
    len = strlen(s);
    for(int i = 0;i < m;i ++) cc[i] = 0;
    for(int i = 0;i < len;i ++) ++ cc[x[i] = s[i]];
    for(int i = 1;i < m;i ++) cc[i] += cc[i - 1];
    for(int i = len - 1;~i;i --) sa[-- cc[x[i]]] = i;
    for(int k = 1;k < len;k <<= 1)
    {
        int p = 0;
        for(int i = len - k;i < len;i ++)  y[p ++] = i;
        for(int i = 0;i < len;i ++) if(sa[i] >= k) y[p ++] = sa[i] - k;
        for(int i = 0;i < m;i ++) cc[i] = 0;
        for(int i = 0;i < len;i ++) ++ cc[x[y[i]]];
        for(int i = 1;i < m;i ++) cc[i] += cc[i - 1];
        for(int i = len - 1;~i;i --) sa[-- cc[x[y[i]]]] = y[i];
        swap(x,y); m = 1; x[sa[0]] = 0;

        for(int i = 1;i < len;i ++)
            x[sa[i]] = cmp(y,sa[i - 1],sa[i],k) ? m - 1 : m ++;
        if(m >= len) break;
    }
}

void get_height(char s[])
{
    len = strlen(s);
    for(int i = 0;i < len;i ++) rak[sa[i]] = i;
```

```
    int h = 0;
    height[0] = 0;
    for(int i = 0;i < len;i ++)
    {
        if(!rak[i]) continue;
        int j = sa[rak[i] - 1];
        if(h) h --;
        while(s[i + h] == s[j + h]) h ++;
        height[rak[i]] = h;
    }
}

char s1[N],s2[N],s[N];

int tot;
pair<int,int> st[N];

int main(){
    while(~scanf("%d",&k)){
        if(k == 0) break;
        scanf("%s",s1);
        scanf("%s",s2);
        int len1 = strlen(s1), len2 = strlen(s2);
        rep(i,0,len1-1) s[i] = s1[i];
        s[len1] = '$';
        rep(i,0,len2-1) s[i+len1+1] = s2[i];
        len = strlen(s);
        get_sa(s);
        get_height(s);

        //A:[0,len1-1], B:[len1+1,len1+len2]
        ll ans = 0, total = 0;
        tot = -1;
        // rep(i,0,len1+len2) logs(i,sa[i],height[i]);
        rep(i,1,len1+len2){
            if(height[i] < k) {tot = -1; total = 0; continue;}
            else{
                // logs(i,sa[i]);
                int cnt = 0;
                if(sa[i-1] <= len1-1) cnt = 1;
                while(tot >= 0 && st[tot].first >= height[i]){
                    total -= (ll)(st[tot].first-k+1)*(ll)st[tot].second;
                    cnt += st[tot].second;
                    tot--;
                }
                if(cnt != 0){
                    st[++tot] = make_pair(height[i],cnt);
                    total += (ll)(height[i]-k+1)*(ll)cnt;
                }
                if(sa[i] >= len1+1) ans += total;
            }
        }
        tot = -1; total = 0;
        rep(i,1,len1+len2){
            if(height[i] < k) {tot = -1; total = 0; continue;}
            else{
                int cnt = 0;
                if(sa[i-1] >= len1+1) cnt = 1;
                while(tot >= 0 && st[tot].first >= height[i]){
```

```
                    total -= (ll)(st[tot].first-k+1)*(ll)st[tot].second;
                    cnt += st[tot].second;
                    tot--;
                }
                if(cnt != 0){
                    st[++tot] = make_pair(height[i],cnt);
                    total += (ll)(height[i]-k+1)*(ll)cnt;
                }
                // logs(i,sa[i],height[i],total);
                if(sa[i] <= len1-1) ans += total;
            }
        }
        printf("%lld\n",ans);
        rep(i,0,len1+len2) s[i] = '\0';
    }
    return 0;
}
```

4.24.5 n 字符串问题

一、至少在 k 个字符串中出现的最长子串

题意: 给定 n 个字符串，求出现在不少于 k 个字符串中的最长子串。

思路: 首先将 n 个字符串拼接在一起，再二分答案进行 $height$ 分组，然后查看每一组中出现了多少个不同的字符串，如果大于等于 k 则符合题意。

二、每个字符串至少出现两次且不重叠的最长子串

题意: 给定 n 个字符串，求在每个字符串中至少出现两次且不重叠的最长子串。

思路: 与上题没有太大的差别，还是先将 n 个字符串拼接在一起，再二分答案进行 $height$ 分组，记录每一组中出现位置的最大值和最小值，然后查看有多少组的最大值和最小值之差符合条件，如果大于等于 k 则符合题意。

```
return 0;
```

4.25 整体二分

4.25.1 无修改第 K 大

一、适用问题:
整体二分，即对所有的查询进行一个整体的二分答案，需要数据结构题满足以下性质。
1. 询问的答案具有可二分性
2. 修改对判定答案的贡献相对独立，修改之间互不影响效果
3. 修改如果对判定答案有贡献，则贡献为一确定的与判定标准无关的值
4. 贡献满足交换律、结合律，具有可加性
5. 题目允许离线操作

(来自《浅谈数据结构题的几个非经典解法》)
上面的性质看上去复杂，其实只要满足询问答案具有可二分性，且题目允许离线操作，就可以考虑一下是否可以利用整体二分算法进行求解。

二、整体二分算法解析
上面的文字可能有些过于理论，我们现在用浅显一点的方式来理解这个算法。

假设你现在有 q 次查询，查询区间第 k 大的值。首先考虑如果只有 1 个查询，是否可以直接二分解决。

显然是可以的，我们只需要定位到具体区间，数一下小于等于当前二分值的数个数是否大于等于 k 即可。于是问题就变成了如何从单次二分演变到整体二分。

我们首先维护一个操作序列，即每个点的赋值和查询，共 $n+q$ 个操作。然后实现一个 $solve(l,r,L,R)$ 函数，表示当前的操作序列在 $[L,R]$ 范围内，而该操作序列中所有的查询操作的答案都在 $[l,r]$ 中。

于是我们二分一个值 $mid=(l+r)/2$，然后将 $[L,R]$ 中所有的赋值操作中数值小于等于 mid 的数加入到对应位置，比如 $a[x]\le mid$，则 $sum[x]=sum[x]+1$，对于所有的查询操作，判断其查询区间 $[x,y]$ 的值是否大于等于 k，如果是则将其递归到 $solve(l,mid)$ 中，否则递归到 $solve(mid+1,r)$ 中，具体内容看一下下面的例题就可以理解。

最后分析一下时间复杂度，最多分了 $logn$ 层，每一层的时间复杂度为 $O(nlogn)$，因此总时间复杂度为 $O(nlog^2n)$。

最后附上《浅谈数据结构题的几个非经典解法》中对该算法的理论概述。

询问的答案可二分且修改对判定标准的贡献相对独立，且贡献的值与判定标准无关。因此如果我们已经计算过某一些修改对询问的贡献，那么这个贡献永远不会改变，我们没有必要当判定标准改变时再次计算这部分修改的贡献，只要记录下当前的总贡献，再进一步二分时，直接加上新的贡献即可。

三、例题解析
题意:
无修改的区间第 k 大数问题。$(1\le n\le 10^5, 1\le m\le 5000)$

思路:
主席树模板题，但此处我们要用整体二分的方法来解决此题。

首先我们将所有赋值操作和查询操作都放到一个数组中，形成了此题的操作序列。然后就是代码中的核心关键点 $solve(l,r,L,R)$ 函数，该函数表示区间 $[L,R]$ 中的操作序列中的查询操作的答案一定在 $[l,r]$ 范围内。

因此问题就变成了如何将 $[L,R]$ 中的序列进行分组，再递归到 $solve(l,mid)$ 和 $solve(mid+1,r)$ 中。我们只需遍历 $[L,R]$ 中的所有操作，如果是赋值操作，则判断数值 x 是否大于 mid，如果小于等于 mid，则将该操作丢到 q_1 数组中，并在树状数组的 x 位置加 1；否则将操作丢到 q_2 数组中。

如果是查询操作，则在树状数组中查询区间 $[q[i].x,q[i].y]$ 的值 tmp，如果 $k\le tmp$，则将该操作丢到 q_1 中；否则将 $k=k-tmp$，然后丢到 q_2 中。

可能说起来比较复杂，但是代码比较清晰，推荐直接对代码进行理解。

```
#include <iostream>
#include <algorithm>
#include <cstdio>
#define mem(a,b) memset(a,b,sizeof a);
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define per(i,a,b) for(int i = a; i >= b; i--)
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
typedef long long ll;
typedef double db;
const int N = 1e5+100;
const int inf = 1e9+10;
const db EPS = 1e-9;
using namespace std;

void dbg() {cout << "\n";}
template<typename T, typename... A> void dbg(T a, A... x) {cout << a << ' '; dbg(x...)
    ;}
#define logs(x...) {cout << #x << " -> "; dbg(x);}

int n,m,ans[N],c[N];
struct Node{int x,y,k,id;}q[2*N],q1[2*N],q2[2*N];
```

```
inline int lowbit(int x) {return x&(~x+1);}
inline void update(int x,int v) {for(; x<=n; x+=lowbit(x)) c[x] += v;}
inline int ask(int x){
    int res = 0;
    while(x) res += c[x], x -= lowbit(x);
    return res;
}

void solve(int l,int r,int L,int R){
    if(l > r || L > R) return;
    if(l == r){
        rep(i,L,R) if(q[i].id) ans[q[i].id] = l;
        return;
    }
    int cnt1 = 0, cnt2 = 0, mid = (l+r)>>1;
    rep(i,L,R){
        if(q[i].id){ //查询
            int tmp = ask(q[i].y)-ask(q[i].x-1);
            if(q[i].k <= tmp) q1[++cnt1] = q[i];
            else q[i].k -= tmp, q2[++cnt2] = q[i];
        }
        else{ //赋值
            if(q[i].x <= mid) update(q[i].y,1), q1[++cnt1] = q[i];
            else q2[++cnt2] = q[i];
        }
    }
    rep(i,1,cnt1) if(!q1[i].id) update(q1[i].y,-1);
    rep(i,1,cnt1) q[L+i-1] = q1[i];
    rep(i,1,cnt2) q[L+cnt1+i-1] = q2[i];
    solve(l,mid,L,L+cnt1-1); solve(mid+1,r,L+cnt1,R);
}

int main()
{
    scanf("%d%d",&n,&m);
    rep(i,1,n) {scanf("%d",&q[i].x); q[i].id = 0; q[i].y = i;}
    rep(i,1,m) {scanf("%d%d%d",&q[i+n].x,&q[i+n].y,&q[i+n].k); q[i+n].id = i;}
    solve(-inf,inf,1,n+m);
    rep(i,1,m) printf("%d\n",ans[i]);
    return 0;
}
```

4.25.2 带修改第 K 大

题意:
带修改的区间第 k 大数问题。$(1 \le n \le 5 * 10^4, 1 \le m \le 10^4)$

思路:
带修改第 k 大问题，如果要用主席树来解决的话，则需要再加上一层树状数组来维护修改信息，即用树套树解决该问题。

但是如果用整体二分来处理这个问题的话，难度则会瞬间骤降。其实此题与上题唯一的区别就是这题多了一个修改操作，而修改操作无非就是删除原来的数，加上新的数。

因此对于 $a[x] = y$ 的修改操作，我们将其拆成两部分，第一部分为删除 $a[x]$，然后令 $a[x] = y$，第二部分是加上 $a[x]$，具体细节可以参考下面的代码实现。

```
#include <iostream>
```

```
#include <algorithm>
#include <cstdio>
#include <cstring>
#define mem(a,b) memset(a,b,sizeof a);
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define per(i,a,b) for(int i = a; i >= b; i--)
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
typedef long long ll;
typedef double db;
const int N = 1e5+100;
const int inf = 1e9+10;
const db EPS = 1e-9;
using namespace std;

void dbg() {cout << "\n";}
template<typename T, typename... A> void dbg(T a, A... x) {cout << a << ' '; dbg(x...);}
#define logs(x...) {cout << #x << " -> "; dbg(x);}

int n,m,ans[N],c[N],a[N];
struct Node{int x,y,k,id;}q[2*N],q1[2*N],q2[2*N];
inline int lowbit(int x) {return x&(~x+1);}
inline void update(int x,int v) {for(; x<=n; x+=lowbit(x)) c[x] += v;}
inline int ask(int x){
    int res = 0;
    while(x) res += c[x], x -= lowbit(x);
    return res;
}

void solve(int l,int r,int L,int R){
    if(l > r || L > R) return;
    if(l == r){
        rep(i,L,R) if(q[i].k) ans[q[i].id] = l;
        return;
    }
    int cnt1 = 0, cnt2 = 0, mid = (l+r)>>1;
    rep(i,L,R){
        if(q[i].k){ //查询
            int tmp = ask(q[i].y)-ask(q[i].x-1);
            if(q[i].k <= tmp) q1[++cnt1] = q[i];
            else q[i].k -= tmp, q2[++cnt2] = q[i];
        }
        else{ //赋值
            if(q[i].x <= mid) update(q[i].id,q[i].y), q1[++cnt1] = q[i];
            else q2[++cnt2] = q[i];
        }
    }
    rep(i,1,cnt1) if(!q1[i].k) update(q1[i].id,-q1[i].y);
    rep(i,1,cnt1) q[L+i-1] = q1[i];
    rep(i,1,cnt2) q[L+cnt1+i-1] = q2[i];
    solve(l,mid,L,L+cnt1-1); solve(mid+1,r,L+cnt1,R);
}

int main()
{
    int _; scanf("%d",&_);
    while(_--){
        scanf("%d%d",&n,&m);
        int cnt = 0, tot = 0;
```

```
        memset(c,0,sizeof c);
        rep(i,1,n) {
            scanf("%d",&a[i]);
            q[++cnt] = {a[i],1,0,i};
        }
        rep(i,1,m){
            char op[5]; scanf("%s",op);
            int x,y,k;
            if(op[0] == 'Q'){
                scanf("%d%d%d",&x,&y,&k);
                q[++cnt] = {x,y,k,++tot};
            }
            else{
                scanf("%d%d",&x,&y);
                q[++cnt] = {a[x],-1,0,x};
                a[x] = y;
                q[++cnt] = {a[x],1,0,x};
            }
        }
        solve(-inf,inf,1,cnt);
        rep(i,1,tot) printf("%d\n",ans[i]);
    }
    return 0;
}
```

4.25.3 线段树 + 整体二分

题意:
n 个位置，m 个操作。操作有两种，1 a b c 表示在第 a 个位置到第 b 个位置，每个位置加入一个数 c；2 a b c 表示询问从第 a 个位置到第 b 个位置，第 c 大的数是多少。$(1 \le n, m \le 5 * 10^4)$

思路:
其实和上面第二个问题没有太大的差别，只不过上一个问题是单点修改，而这题变成了区间修改。因此我们用线段树维护一下整体二分即可解决。

```
#include <iostream>
#include <algorithm>
#include <cstdio>
#include <cstring>
#define mem(a,b) memset(a,b,sizeof a);
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define per(i,a,b) for(int i = a; i >= b; i--)
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
typedef long long ll;
typedef double db;
const int N = 1e5+100;
const int inf = 1e9+10;
const db EPS = 1e-9;
using namespace std;

void dbg() {cout << "\n";}
template<typename T, typename... A> void dbg(T a, A... x) {cout << a << ' '; dbg(x...)
    ;}
#define logs(x...) {cout << #x << " -> "; dbg(x);}

int n,m;
ll ans[N],sum[2*N],lazy[2*N];
```

```
struct Node{ll x,y,k,id;}q[2*N],q1[2*N],q2[2*N];

inline int get_id(int l,int r) {return (l+r)|(l!=r);}
inline void pushDown(int l,int r){
    int mid = (l+r)>>1, now = get_id(l,r), ls = get_id(l,mid), rs = get_id(mid+1,r);
    sum[ls] += lazy[now]*(ll)(mid-l+1); sum[rs] += lazy[now]*(ll)(r-mid);
    lazy[ls] += lazy[now]; lazy[rs] += lazy[now];
    lazy[now] = 0;
}
inline void update(int l,int r,int L,int R,int v){
    int now = get_id(l,r);
    if(L <= l && r <= R){
        sum[now] += (ll)v*(ll)(r-l+1);
        lazy[now] += v;
        return;
    }
    if(lazy[now]) pushDown(l,r);
    int mid = (l+r)>>1;
    if(L <= mid) update(l,mid,L,R,v);
    if(R > mid) update(mid+1,r,L,R,v);
    sum[now] = sum[get_id(l,mid)]+sum[get_id(mid+1,r)];
}
inline ll query(int l,int r,int L,int R){
    int now = get_id(l,r);
    if(L <= l && r <= R) return sum[now];
    if(lazy[now]) pushDown(l,r);
    int mid = (l+r)>>1;
    ll thp = 0;
    if(L <= mid) thp += query(l,mid,L,R);
    if(R > mid) thp += query(mid+1,r,L,R);
    return thp;
}

void solve(int l,int r,int L,int R){
    if(l > r || L > R) return;
    if(l == r){
        rep(i,L,R) if(q[i].id) ans[q[i].id] = l;
        return;
    }
    int cnt1 = 0, cnt2 = 0, mid = (l+r)>>1;
    rep(i,L,R){
        if(q[i].id){ //查询
            ll tmp = query(1,n,q[i].x,q[i].y);
            if(q[i].k <= tmp) q2[++cnt2] = q[i];
            else q[i].k -= tmp, q1[++cnt1] = q[i];
        }
        else{ //赋值
            //由于右区间的起始点为mid+1, 因此此处为 >= mid+1
            if(q[i].k >= mid+1) update(1,n,q[i].x,q[i].y,1), q2[++cnt2] = q[i];
            else q1[++cnt1] = q[i];
        }
    }
    rep(i,1,cnt2) if(!q2[i].id) update(1,n,q2[i].x,q2[i].y,-1);
    rep(i,1,cnt1) q[L+i-1] = q1[i];
    rep(i,1,cnt2) q[L+cnt1+i-1] = q2[i];
    solve(l,mid,L,L+cnt1-1); solve(mid+1,r,L+cnt1,R);
}

int main()
```

```
{
    scanf("%d%d",&n,&m);
    int tot = 0;
    rep(i,1,m){
        ll op,x,y,k;
        scanf("%lld%lld%lld%lld",&op,&x,&y,&k);
        if(op == 1) q[i] = {x,y,k,0};
        else q[i] = {x,y,k,++tot};
    }
    solve(-inf,inf,1,m);
    rep(i,1,tot) printf("%lld\n",ans[i]);
    return 0;
}
```

4.25.4 并查集 + 整体二分

题意:
一个 n 个点，m 条边的图，第 i 条边连接 a_i 和 b_i，保证图是连通的。

现在有 q 次询问，每次询问给出一个三元组 x y z，表示询问从 x y 两个点出发，一共扩展 z 个不同的点（包括起始点），求所经过的边中最大编号的最小值。$(3 \le n \le 10^5, 1 \le q \le 10^5)$

思路:
此题较之上面三题，没有那么套路，因此我们先从只有一个询问开始找思路。

首先考虑能不能把图变成树，因为图上问题往往都很复杂，而变成树上问题后我们的可操作空间会大很多。继续思考不难发现，如果我们按边的编号为权值构建一棵最小生成树，每次询问的答案也一定会落在最小生成树上的边上。

转到树上问题之后，我们考虑能不能二分答案然后 $check$，如果只有一个询问的话，显然是可以的。只需要维护一个可加边可删边的按秩合并的并查集即可。

既然单个查询可以二分，那一定可以用整体二分的方法对所有查询进行二分。我们在二分值为 mid 时，将所有编号小于等于 mid 的边连接起来，然后再递归到 $[mid+1, r]$ 区间。右区间递归结束后，再撤销二分值为 mid 时连接的边，然后递归 $[l, mid]$。

如果连接操作是 $fa[x] = y, sz[y] = sz[y] + sz[x]$，那么撤销操作的时候，如果只有 $fa[x] = x, sz[y] = sz[y] - sz[x]$ 是不够的，需要从 y 开始不断向上访问，对于访问到的每一个节点都减去 $sz[x]$，如此才能保证撤销操作的正确性。

```
#include <bits/stdc++.h>
#define mem(a,b) memset(a,b,sizeof a);
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define per(i,a,b) for(int i = a; i >= b; i--)
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
typedef long long ll;
typedef double db;
const db EPS = 1e-9;
const int N = 1e6+100;
using namespace std;

void dbg() {cout << "\n";}
template<typename T, typename... A> void dbg(T a, A... x) {cout << a << ' '; dbg(x...)
    ;}
#define logs(x...) {cout << #x << " -> "; dbg(x);}

int n,m,Q,fa[N],sz[N],ans[N];
struct Node {int x,y,k,id,h1,h2;} q[N],q1[N],q2[N];

```

```
int find(int x) {return x == fa[x] ? x : find(fa[x]);}
int calc(int x,int y){
    int fx = find(x), fy = find(y);
    return fx == fy ? sz[fx] : (sz[fx] + sz[fy]);
}
pair<int,int> merge(int x,int y){
    int fx = find(x), fy = find(y);
    if(fx == fy) return make_pair(-1,-1);
    if(sz[fx] < sz[fy]){
        fa[fx] = fy, sz[fy] += sz[fx];
        return make_pair(fx,fy);
    }
    else{
        fa[fy] = fx, sz[fx] += sz[fy];
        return make_pair(fy,fx);
    }
}
void Delete(int x,int y) {
    fa[x] = x;
    while(y){
        sz[y] -= sz[x];
        if(y == fa[y]) break;
        y = fa[y];
    }
}

void solve(int l,int r,int L,int R){
    if(l > r || L > R) return;
    if(l == r){
        rep(i,L,R) if(q[i].k) ans[q[i].id] = l;
        return;
    }
    int mid = (l+r)>>1, cnt1 = 0, cnt2 = 0;
    // logs(mid,L,R);
    rep(i,L,R){
        if(q[i].k){ //查询
            int tmp = calc(q[i].x,q[i].y);
            if(q[i].k <= tmp) q1[++cnt1] = q[i];
            else q2[++cnt2] = q[i];
        }
        else{
            if(q[i].id <= mid){
                pair<int,int> tmp = merge(q[i].x,q[i].y);
                q1[++cnt1] = q[i];
                q1[cnt1].h1 = tmp.first; q1[cnt1].h2 = tmp.second;
            }
            else q2[++cnt2] = q[i];
        }
    }
    rep(i,1,cnt1) q[L+i-1] = q1[i];
    rep(i,1,cnt2) q[L+cnt1+i-1] = q2[i];
    solve(mid+1,r,L+cnt1,R);
    rep(i,1,cnt1) if(!q[L+i-1].k) Delete(q[L+i-1].h1,q[L+i-1].h2);
    solve(l,mid,L,L+cnt1-1);
}

int main()
{
    int cnt = 0;
```

```
    scanf("%d%d",&n,&m);
    rep(i,1,n) fa[i] = i, sz[i] = 1;
    rep(i,1,m){
        int x,y; scanf("%d%d",&x,&y);
        if(merge(x,y).first == -1) continue;
        q[++cnt] = {x,y,0,i,0,0};
        // logs(i);
    }
    scanf("%d",&Q);
    rep(i,1,Q){
        int x,y,z; scanf("%d%d%d",&x,&y,&z);
        q[++cnt] = {x,y,z,i,0,0};
    }
    rep(i,1,n) fa[i] = i, sz[i] = 1;
    solve(0,m+1,1,cnt);
    rep(i,1,Q) printf("%d\n",ans[i]);
    return 0;
}
```

4.26 分块

题意：长度为 n 的序列，每个点都有一个颜色，一共有 C 个颜色，支持两种操作，第一种给出 l r x，将区间 $[l,r]$ 全部染成 x，第二种给出一个 x，询问 x 颜色一共在序列中出现了多少次。

所有操作结束后，还要查询每种颜色出现的次数，给出最多出现的次数。($1 \le n, C, m \le 10^5$)

思路: 非常明显的区间推平操作，应该使用珂朵莉树进行解决，即用 set 维护每一块相同区域。

老司机树是可以的，但是并没有采用，还是使用了分块的方法，虽然 T 的飞起... 一开始的想法是给每个块记一个 map，标记每个块中各个数字出现的次数，如果整个块被赋值了，就 $map.clear()$。但是 T 的不行，第 63 个点怎么也过不去...

后来采用了完全的暴力分块，即每个块打个 $lazy$，如果 $lazy$ 不为 0，则整个块中所有点大小均为 $lazy$。如果 $lazy$ 为 0，则暴力修改、查询。那么如何分析这个做法的时间复杂度呢?

一共两种操作，修改操作每次最多涉及 $\sqrt{n}$ 个块，如果块被完全覆盖，$O(1)$ 打上标记，如果块只被部分覆盖，下放标记 $O(\sqrt{n})$，修改点值 $O(\sqrt{n})$，由于每次最多只有 2 个块被部分覆盖，最多 $\sqrt{n}$ 个块被完全覆盖，因此修改操作总时间复杂度为 $O(\sqrt{n})$。

而对于查询操作，如果该块被打上了标记，则查询复杂度为 $O(1)$，如果没有打上标记，则直接暴力查询，时间复杂度为 $O(\sqrt{n})$。此题是修改与查询次数一致，而修改操作每次最多使得 2 个块标记消失。因此可以发现，我们可以故意设置数据使得此种做法 TLE，即利用修改操作使得所有块的 $lazy$ 标记消失，然后每次查询颜色时就会遍历所有块，可以将时间复杂度卡成 $O(n^2)$。(不过此题数据较水，此种方法也可以过，而且很快，390 ms 即可通过...)

因此正确的做法是当块没有 $lazy$ 标记时，为了避免直接暴力询问，我们对于每个块记录一个 $unordered_map$，查询复杂度为 $O(1)$。修改复杂度会变成 $O(\sqrt{n} * log(n))$，查询复杂度为 $O(\sqrt{n})$，但是避免了被卡的可能。(790 ms 可以通过)

总结: 对于一些不太常见的数据结构问题，一定要尝试考虑分块，因为分块方法用途很多，而且算法核心是暴力。

而对于分块算法，一定要多多尝试，你可能觉得这样分块会被你故意构造的一组数据卡成 $O(n^2)$，但是出题人未必就考虑到了你的这种做法，因此一定要多尝试，不要只去尝试那些 $O(n*logn*\sqrt{n})$ 的算法。当然如果能直接想到 $O(\sqrt{n})$ 的算法则是最好的!

```
#include <bits/stdc++.h>
#define mem(a,b) memset(a,b,sizeof a);
#define rep(i,a,b) for(int i = a; i <= b; i++)
```

```
#define per(i,a,b) for(int i = a; i >= b; i--)
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
typedef long long ll;
typedef double db;
const int N = 1e5+10;
const db EPS = 1e-9;
using namespace std;

void dbg() {cout << "\n";}
template<typename T, typename... A> void dbg(T a, A... x) {cout << a << ' '; dbg(x...);}
#define logs(x...) {cout << #x << " -> "; dbg(x);}

int n,m,a[N],L[N],R[N],pos[N],cnt,sz,lazy[N],num[N];
unordered_map<int,int> mp[1010];

void change(int l,int r,int x){
    rep(i,pos[l],pos[r]){
        if(L[i] >= l && R[i] <= r){
            // mp[i].clear();
            // mp[i][x] = R[i]-L[i]+1;
            lazy[i] = x;
        }
        else{
            if(lazy[i]){
                rep(j,L[i],R[i]) a[j] = lazy[i];
                lazy[i] = 0;
            }
            rep(j,max(l,L[i]),min(R[i],r)){
                // mp[i][a[j]]--;
                // mp[i][x]++;
                a[j] = x;
            }
        }
    }
}

int ask(int x){
    int ans = 0;
    rep(i,1,cnt){
        if(lazy[i]){
            if(lazy[i] == x) ans += R[i]-L[i]+1;
        }
        else{
            rep(j,L[i],R[i])
                if(a[j] == x) ans++;
        }
    }
    return ans;
}

int main()
{
    int h; scanf("%d%d%d",&n,&h,&m);
    sz = sqrt(n);
    cnt = n/sz;
    if(n%sz != 0) cnt++;
    rep(i,1,cnt){
        L[i] = sz*(i-1)+1;
```

```
        R[i] = min(sz*i,n);
        mp[i][1] = R[i]-L[i]+1;
    }
    rep(i,1,n) pos[i] = ((i-1)/sz)+1;
    rep(i,1,n) a[i] = 1;

    while(m--){
        ll p,x,a,b;
        scanf("%lld%lld%lld%lld",&p,&x,&a,&b);
        ll s = ask(p);
        ll m1 = (a+s*s)%n;
        ll m2 = (a+(s+b)*(s+b))%n;
        change(min(m1,m2)+1,max(m1,m2)+1,x);
    }
    rep(i,1,cnt)
        if(lazy[i]){
            rep(j,L[i],R[i]) a[j] = lazy[i];
            lazy[i] = 0;
        }
    rep(i,1,n)
        num[a[i]]++;
    int maxx=0;
    rep(i,1,h)
        maxx = max(maxx,num[i]);
    printf("%d\n",maxx);
    return 0;
}
```

5 图论

5.1 最短路

5.1.1 Dijkstra

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <string>
#include <queue>
#include <algorithm>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const int N = 5*1e6+100;
const int M = 5*1e6+100;
const ll inf = 5*1e16;
const db EPS = 1e-9;
using namespace std;

struct Edge{
    int to,next;
    ll w;
}e[M];
int head[N],tot,n,vis[N];
ll c[N],dis[N];

void add(int x,int y,ll w){
    e[++tot].to = y, e[tot].next = head[x], e[tot].w = w, head[x] = tot;
}

priority_queue< pair<ll,int> > q;

ll dijkstra(int s)
{
    while(q.size()) q.pop();
    memset(vis,0,sizeof vis);
    rep(i,0,2*n+2) dis[i] = inf;
    dis[s] = 0;
    q.push(make_pair(0ll,s));
    while(q.size())
    {
        int x = q.top().second;
        q.pop();
        if(vis[x]) continue;
        vis[x] = 1; //只有从队列中弹出时才是更新好了这个点，此时才可以更新vis，不能在加入的时候就更新vis
        for(int i = head[x]; i; i = e[i].next)
        {
            int y = e[i].to;
            ll z = e[i].w;
            if(dis[y] > (ll)(dis[x] + z)){
```

```
                dis[y] = dis[x]+z;
                q.push(make_pair(-dis[y],y));
            }
        }
    }
    if(dis[n*2+2] == inf) return -1;
    return dis[2*n+2];
}

int main()
{
    __; cin >> n; tot = 1;
    rep(i,1,n) cin >> c[i];
    string b1,b2;
    int r1,r2;
    rep(i,1,n){
        string s1,s2;
        cin >> s1; s2 = s1;
        reverse(s2.begin(),s2.end());
        if(i == 1){
            b1 = s1, b2 = s2;
            r1 = i, r2 = i+n;
            add(2*n+1,r1,0);
            add(2*n+1,r2,c[1]);
        }
        else{
            if(b1 <= s1) add(r1,i,0ll);
            if(b1 <= s2) add(r1,i+n,c[i]);
            if(b2 <= s1) add(r2,i,0ll);
            if(b2 <= s2) add(r2,i+n,c[i]);
            b1 = s1, b2 = s2;
            r1 = i, r2 = i+n;
        }
    }
    add(n,2*n+2,0);
    add(2*n,2*n+2,0);
    ll x1 = dijkstra(2*n+1);
    cout << x1 << endl;
    return 0;
}
```

5.1.2 Dijkstra 记录路径

```
#include <cstdio>
#include <iostream>
#include <stack>
#include <cstring>
using namespace std;
const int N = 100+10;
const int inf = 0x3fffffff;

int e[N][N],dis[N],path[N],vis[N];
int n,m;

void dijkstra(int x)
{
    //初始化dis数组
    for(int i = 1;i <= n;i++)
```

```
        dis[i] = e[x][i];

    memset(vis,0,sizeof(vis));
    vis[x] = 1;
    for(int i = 1;i <= n-1;i++)
    {
        int minn = inf;
        int u;
        for(int j = 1;j <= n;j++)
        {
            if(!vis[j] && dis[j] < minn)
            {
                minn = dis[j];
                u = j;
            }
        }
        vis[u] = 1;
        for(int j = 1;j <= n;j++)
        {
            if(e[u][j] < inf)
            {
                if(e[u][j]+dis[u] < dis[j])
                {
                    dis[j] = e[u][j]+dis[u];
                    path[j] = u;    //记录该点是被哪一个点松弛的
                }
            }
        }
    }
}

void print(int x,int y)   //x为起点，y为终点
{
    stack<int> q;
    int tmp = y;
    while(path[y] != -1)
    {
        q.push(y);
        y = path[y];
    }
    q.push(y);

    //打印路径
    printf("%d=>%d, length:%d, path: %d ",x,tmp,dis[tmp],x);
    while(!q.empty())       //先进后出,获得正序
    {
        printf("%d ",q.top());//打印堆的头节点
        q.pop();            //将堆的头节点弹出
    }
    printf("\n");
}

int main()
{
    cin>>n>>m;
    memset(path,-1,sizeof(path));
    //把图初始化
    for(int i = 1;i <= n;i++)
    {
```

```
        for(int j = 1;j <= n;j++)
        {
            if(i == j) e[i][j] = 0;
            else e[i][j] = inf;
        }
    }
    //读入边
    for(int i = 1;i <= m;i++)
    {
        int t1,t2,t3;
        cin>>t1>>t2>>t3;
        e[t1][t2] = t3;
    }
    dijkstra(1);
    cout<<"n:"<<n<<endl;
    print(1,n);
    return 0;
}
```

5.1.3 分层图

题意:
给一张有向图，给一个 k，可以将图中 k 条边的边权变为 0，求从 1->n 的最短路径

解法:
由于 k<=10，由此考虑分层图做法
给每个点建 10 个分身，dis[i][j] 表示从城市 1 到达城市 i，一共改了 j 条边的最短路径
其余操作和正常最短路求法一致，更新的时候有两种更新方法。假设队列弹出的节点是 dis[i][j]，y 为 i 可达到的点
则用 dis[i][j] 去更新 dis[y][j] 和 dis[y][j+1]，相当于扩点操作，其余操作和 dij 模板一致
如此操作即可求出答案

```
#include <cstdio>
#include <iostream>
#include <algorithm>
#include <cstring>
#include <queue>
#define rep(i,a,b) for(int i = a;i <= b;i++)
using namespace std;
const int N = 1e5+100;
const int M = 5*1e5+10;
typedef long long ll;
const ll inf = 1e15;

struct Edge{
    int to,next,w;
}e[M];
int n,m,k,head[N],tot;
ll dis[N][15];
int vis[N][15];

void init()
{
    tot = 1;
    rep(i,0,n) head[i] = 0;
}

void add(int x,int y,int z)
```

```
{
    e[++tot].to = y, e[tot].next = head[x], head[x] = tot, e[tot].w = z;
}

struct Point{
    ll ans;
    int u,cnt;
}thp;

bool operator < (Point a,Point b)
{
    return a.ans > b.ans;
}

priority_queue<Point> q;

void dijstra(int s)
{
    while(q.size()) q.pop();
    thp.ans = 0, thp.u = s, thp.cnt = 0;
    q.push(thp);
    rep(i,1,n)
        rep(j,0,k) vis[i][j] = 0;
    rep(i,1,n)
        rep(j,0,k) dis[i][j] = inf;
    dis[s][0] = 0;
    while(q.size())
    {
        thp = q.top();
        q.pop();
        int u = thp.u;
        int x = thp.cnt;
        if(vis[u][x]) continue;
        vis[u][x] = 1;
        for(int i = head[u]; i ; i = e[i].next)
        {
            int y = e[i].to;
            if(!vis[y][x]){
                if(dis[y][x] > dis[u][x]+e[i].w){
                    dis[y][x] = dis[u][x] + e[i].w;
                    thp.u = y, thp.ans = dis[y][x], thp.cnt = x;
                    q.push(thp);
                }
            }
            if(!vis[y][x+1] && (x < k)){
                if(dis[y][x+1] > dis[u][x]){
                    dis[y][x+1] = dis[u][x];
                    thp.u = y, thp.ans = dis[y][x+1], thp.cnt = x+1;
                    q.push(thp);
                }
            }
        }
    }
}

int main()
{
    int T;
    scanf("%d",&T);
```

```
    while(T--)
    {
        init();
        scanf("%d%d%d",&n,&m,&k);
        rep(i,1,m)
        {
            int x,y,z;
            scanf("%d%d%d",&x,&y,&z);
            add(x,y,z);
        }
        dijstra(1);
        printf("%lld\n",dis[n][k]);
    }
    return 0;
}
```

5.1.4 spfa 判断负环

出现负环的条件有两个:
1. 最短路边数 $\geq$ n
2. 入队次数 $\geq$ n
spfa 类似于 bfs 的过程
第 i 轮入队意味着确定了最短路边数为 i 的最短路长度
如果一个点入队了 n 次，则意味着最短路边数至少为 n，即出现了环

```
#include <bits/stdc++.h>
#define mem(a,b) memset(a,b,sizeof a);
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define per(i,a,b) for(int i = a; i >= b; i--)
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
typedef long long ll;
typedef double db;
const int inf = 1e8;
const int N = 1e3+10;
const db EPS = 1e-9;
using namespace std;

void dbg() {cout << "\n";}
template<typename T, typename... A> void dbg(T a, A... x) {cout << a << ' '; dbg(x...)
    ;}
#define logs(x...) {cout << #x << " -> "; dbg(x);}

int n,m,Q,a[N],vis[N],dis[N],cnt[N];
vector<pair<int,int> > G[N];
queue<int> q;

void init(){
    rep(i,1,n) G[i].clear();
    rep(i,1,n) vis[i] = cnt[i] = 0, dis[i] = inf;
}

// 找到被负环影响的所有点，并标记
void dfs(int x){
    cnt[x] = inf;
    for(auto item:G[x])
        if(cnt[item.first] != inf) dfs(item.first);
}
```

```

void spfa(int s){
    while(q.size()) q.pop();
    dis[s] = 0; vis[s] = 1;
    q.push(s);
    while(q.size()){
        int x = q.front(); q.pop();
        vis[x] = 0;
        if(cnt[x] >= n) continue;
        for(auto item:G[x]){
            int y = item.first, w = item.second;
            if(cnt[y] == inf) continue;
            if(dis[y] > dis[x] + w){
                dis[y] = dis[x] + w;
                // cnt[x] 记录最短路边数
                cnt[y] = cnt[x] + 1;
                if(cnt[y] >= n){
                    if(cnt[y] == n) dfs(y);
                    continue;
                }
                if(!vis[y]) q.push(y), vis[y] = 1;
            }
        }
    }
}

int main()
{
    int _; scanf("%d",&_);
    rep(Ca,1,_){
        scanf("%d",&n);
        init();
        rep(i,1,n) scanf("%d",&a[i]);
        scanf("%d",&m);
        rep(i,1,m){
            int x,y; scanf("%d%d",&x,&y);
            int v = (a[y]-a[x])*(a[y]-a[x])*(a[y]-a[x]);
            G[x].push_back(make_pair(y,(a[y]-a[x])*(a[y]-a[x])*(a[y]-a[x])));
        }
        spfa(1);
        scanf("%d",&Q);
        printf("Case %d:\n", Ca);
        rep(i,1,Q){
            int p; scanf("%d",&p);
            if(dis[p] == inf || cnt[p] == inf || dis[p] < 3) printf("?\n");
            else printf("%d\n", dis[p]);
        }
    }
    return 0;
}
```

5.1.5 最短路计数问题

题意: 一个高度为 h 的电梯，初始位置在第一层。电梯有四个按钮。
1. 向上移动 a 格
2. 向上移动 b 格
3. 向上移动 c 格
4. 返回第一层

问 1 h 层中有多少层是可达的。$(1 \le h \le 10^{18}, 1 \le a, b, c \le 100000)$

思路: 观察数据范围可以发现，a b c 数据范围比较小，可以从此处着手。
不难发现，如果 2 可以达到，则 $2+a$、$2+2*a$、$2+3*a$... 均可到达。即我们以 a 作为基底，如果 x $(x<a)$ 可以到达，则 y $(y\%a=x)$ 也可到达。
因此问题转化为对于 $x(x<a)$ 来说，最小的可达的 y 是多少? $(y\%a=x)$ 既然问题变成了求最小，那么不难想到使用最短路来解决这个问题。
我们枚举 x $(0 \le x < a)$，令 x 与 $(x+b)\%a$ 和 $(x+c)\%a$ 连边，边权分别为 b、c。然后起点为 1，即可求出到达所有 x 时的最小值。
求取答案时就可以枚举 x，然后求 $dis[x]$ h 中有多少个 $mod\ a = x$ 的值即可。

```
#include <bits/stdc++.h>
#define rep(i,a,b) for(int i = a; i <= b; i++)
typedef long long ll;
typedef double db;
const int N = 1e5+100;
const int M = 1e6+100;
const ll inf = 1e15;
const db EPS = 1e-9;
using namespace std;

ll h,a,b,c,dis[N];
int head[N],tot,vis[N];
struct Node{
    int to,next;
    ll w;
}e[M];

void init() {tot = 1;}

void add(int x, int y, ll w){
    e[++tot].to = y, e[tot].next = head[x], head[x] = tot, e[tot].w = w;
}

void dijkstra(int s){
    priority_queue<pair<ll,int> > q;
    while(q.size()) q.pop();
    rep(i,0,a-1) dis[i] = inf, vis[i] = 0;
    dis[s%a] = 1;
    q.push(make_pair(-dis[s%a],s%a));
    while(q.size()){
        int x = q.top().second; q.pop();
        if(vis[x]) continue;
        vis[x] = 1;
        for(int i = head[x]; i; i = e[i].next){
            int y = e[i].to;
            if(dis[y] > dis[x] + e[i].w){
                dis[y] = dis[x] + e[i].w;
                q.push(make_pair(-dis[y],y));
            }
        }
    }
}

int main()
{
    scanf("%lld%lld%lld%lld",&h,&a,&b,&c);
    init();
    rep(i,0,a-1){
```

```
        add(i,(i+b)%a,b);
        add(i,(i+c)%a,c);
    }
    dijkstra(1);
    ll ans = 0;
    rep(i,0,a-1){
        if(dis[i] == inf || dis[i] > h) continue;
        ans += 1ll + (h-dis[i])/a;
    }
    printf("%lld\n",ans);
    return 0;
}
```

5.2 最小生成树

```
//最小生成树 ——Kruscal
#include <cstdio>
#include <iostream>
#include <algorithm>
#include <cstring>
using namespace std;

const int maxn = 1000+10;
const int maxm = 4000+10;

struct edge{
    int u,v,w;
}e[maxn];
int n,m,sum;   //sum为总共的路程
int p[maxn]; //记录每个点的祖先

bool cmp(edge e1,edge e2)
{
    return e1.w < e2.w;
}

int find(int x)
{
    return p[x]==x?x:p[x]=find(p[x]);
}

bool merge(int x,int y)
{
    int t1 = find(x);
    int t2 = find(y);
    if(t1!=t2)
    {
        p[t1] = t2;
        return true;
    }
    return false;
}

int main()
{
    scanf("%d%d",&n,&m);
    for(int i = 1;i <= m;i++)
    {
```

```
        scanf("%d%d%d",&e[i].u,&e[i].v,&e[i].w);
    }
    sort(e+1,e+m+1,cmp);
    for(int i = 1;i <= n;i++) p[i] = i;

    //Kruscal算法核心部分
    int count = 0;  //最小生成树已经有几条边了
    sum = 0;
    int maxx = -1;
    int minn = 10010;
    for(int i = 1;i <= m;i++)
    {
        //判断一条边的两个顶点是否已经连通，即判断是否已在同一个集合中
        if(merge(e[i].u,e[i].v))
        {
            count++;
            sum = sum+e[i].w;
            maxx = max(maxx,e[i].w);
            minn = min(minn,e[i].w);
        }
        if(count == n-1) break;
    }
    cout<<sum<<endl;
    cout<<maxx-minn<<endl;

    return 0;
}
```

5.3 最近公共祖先

5.3.1 LCA (dfs 版本)

```
#include <cstdio>
#include <iostream>
#include <algorithm>
#include <cstring>
#include <queue>
#include <cmath>
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl
    ;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2
    << " , " << z1 << ": " << z2 << endl;
#define rep(i,a,b) for(int i = a;i <= b;i++)
using namespace std;
const int N = 5*1e5+100;

int f[N][25],d[N],dis[N];  //f[x][k]:表示点x向根节点走2^k步到达的点
//d[x]:表示点x在图中的深度  dis[x]:表示根节点到点x的距离
struct Edge{
    int to,next;
}e[2*N];    //树的边为n-1，此处是因为无向边，所以正反各一条
int head[N],tot,t,n,m,s; //n个点，m次询问

void init()
{
    tot = 1;
    memset(head,0,sizeof head);
    memset(d,0,sizeof d);
```

```
}

void add(int x,int y)
{
    e[++tot].to = y; e[tot].next = head[x]; head[x] = tot;
}

void dfs(int u,int fa)
{
    d[u]=d[fa]+1; f[u][0]=fa;
    for(int i=1;(1<<i)<=d[u];i++)
        f[u][i]=f[f[u][i-1]][i-1];
    for(int i=head[u]; i; i=e[i].next){
        int v=e[i].to;
        if(v!=fa) dfs(v,u);
    }
}

int lca(int x,int y)
{
    if(d[x] > d[y]) swap(x,y);
    for(int i = t; i >= 0; i--)
        if(d[f[y][i]] >= d[x]) y = f[y][i];  //往上追溯，直至y和x位于同一深度
    if(x == y) return x;  //如果已经找到了，就返回x
    for(int i = t; i >= 0; i--)
        if(f[x][i] != f[y][i]) x = f[x][i], y = f[y][i];  //x和y同时往上走，一直到x和y恰好为lca的子节点
    return f[x][0];  //x和y共同的根节点就是lca
}

int main()
{
    scanf("%d%d%d",&n,&m,&s);
    t = (int)(log(n)/log(2))+1;
    init();
    rep(i,1,n-1){
        int x,y;
        scanf("%d%d",&x,&y);
        add(x,y), add(y,x);
    }
    dfs(s,0);
    for(int i = 1;i <= m;i++)
    {
        int x,y;
        scanf("%d%d",&x,&y);
        printf("%d\n",lca(x,y));
    }
    return 0;
}
```

5.3.2 LCA (bfs 版本)

```
#include <cstdio>
#include <iostream>
#include <algorithm>
#include <cstring>
#include <queue>
#include <cmath>
```

```cpp
#define rep(i,a,b) for(int i = a;i <= b;i++)
using namespace std;
const int N = 50010;

int f[N][20],d[N],dis[N];  //f[x][k]:表示点x向根节点走2^k步到达的点
//d[x]:表示点x在图中的深度  dis[x]:表示根节点到点x的距离
struct Edge{
    int to,next,w;
}e[2*N];        //树的边为n-1，此处是因为无向边，所以正反各一条
int head[N],tot,t,n,m; //n个点，m次询问

void init()
{
    tot = 1;
    memset(head,0,sizeof head);
    memset(d,0,sizeof d);
}

void add(int x,int y,int z)
{
    e[++tot].to = y; e[tot].next = head[x]; e[tot].w = z; head[x] = tot;
}

void bfs()
{
    queue<int> q;
    while(q.size()) q.pop();
    q.push(1); d[1] = 1; dis[1] = 0;    //把1当做树根
    while(q.size())
    {
        int x = q.front(); q.pop();
        for(int i = head[x]; i ;i = e[i].next){
            int y = e[i].to;
            if(d[y]) continue;
            d[y] = d[x]+1;
            dis[y] = dis[x]+e[i].w; //dist[y]:从1到y的距离
            f[y][0] = x;  //y走2^0步到达x
            for(int j = 1; j <= t;j++)
                f[y][j] = f[f[y][j-1]][j-1];
            q.push(y);
        }
    }
}

/*
    这里有一个非常容易错的点，因为每一个点的父节点初始化为0，所以如果你的点从0-n编号的话，一定要注意bfs的时候将0作为根节点
    否则如果用1作为根节点的话，你会发现1的父节点是0，那么程序就开始出错了！

    所以更好的方法是以后给点编号的时候，统一用1-n编号，不要用0编号，否则极易出错

    从1-n编号的话，可以从任意起点拉起一棵树，但是要注意题目中n的最小值
*/

int lca(int x,int y)
{
    if(d[x] > d[y]) swap(x,y);
    for(int i = t; i >= 0; i--)
        if(d[f[y][i]] >= d[x]) y = f[y][i];  //往上追溯，直至y和x位于同一深度
```

```
    if(x == y) return x;  //如果已经找到了，就返回x
    for(int i = t; i >= 0; i--)
        if(f[x][i] != f[y][i]) x = f[x][i], y = f[y][i];  //x和y同时往上走，一直到x和y恰好为lca的子节点
    return f[x][0];  //x和y共同的根节点就是lca
}

int main()
{
    int T;
    scanf("%d",&T);
    while(T--)
    {
        scanf("%d%d",&n,&m);
        t = (int)(log(n)/log(2))+1;
        init();
        rep(i,1,n-1){
            int x,y,z;
            scanf("%d%d%d",&x,&y,&z);
            add(x,y,z), add(y,x,z);
        }
        bfs();
        for(int i = 1;i <= m;i++)
        {
            int x,y;
            scanf("%d%d",&x,&y);
            printf("%d\n",dis[x]+dis[y]-2*dis[lca(x,y)]);
        }
    }
    return 0;
}
```

5.4 拓扑排序

题意：
一个 n*n 的网格图，每行每列各涂一次，每行每列均有 26 种涂法，输出涂色方案。

思路：
可以发现本题有个性质，最后涂的行或列，一定只有一个元素。
因此开一个结构体记录每行每列，各个颜色的涂色情况，以及该行或列一共有几种不同的颜色。
然后类似于跑拓扑排序，将只有一种颜色的行列加入队列，再依次撤销。
将新的颜色数变为 1 的行列加入队列，即可完成本题。

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <queue>
#include <algorithm>
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x) cout << "x: " << x << endl;
#define LOG2(x,y) cout << "x: " << x << ", y: " << y << endl;
typedef long long ll;
typedef double db;
const db EPS = 1e-9;
using namespace std;
const int N = 3000+100;

```

```
int n;
char mp[N][N];
struct Node{
    int total;
    int num[30];
}t[2*N];
int ans[2*N][4],tot;
int vis[2*N];

void solve()
{
    tot = 0;
    queue<int> q;
    while(q.size()) q.pop();
    rep(i,1,n*2) {
        if(t[i].total == 1) q.push(i);
        else if(t[i].total == 0) ans[++tot][1] = i, ans[tot][2] = 0, vis[i] = 1;
    }
    while(q.size()){
        int x = q.front();
        q.pop();
        if(vis[x]) continue;
        t[x].total = 0;
        ans[++tot][1] = x;
        vis[x] = 1;
        rep(i,0,27)
            if(t[x].num[i] >= 1) ans[tot][2] = i;
        int pp = ans[tot][2];
        if(x <= n){
            rep(i,n+1,n*2){
                if(mp[x][i-n]-'a' != pp) continue;
                t[i].num[pp]--;
                if(t[i].num[pp] == 0){
                    t[i].total--;
                    if(t[i].total == 1) q.push(i);
                }
            }
        }
        else{
            rep(i,1,n){
                if(mp[i][x-n]-'a' != pp) continue;
                t[i].num[pp]--;
                if(t[i].num[pp] == 0){
                    t[i].total--;
                    if(t[i].total == 1) q.push(i);
                }
            }
        }
    }
    for(int i = 2*n; i >= 1; i--)
    {
        int x = ans[i][1], y = ans[i][2];
        if(x <= n) printf("h %d ",x);
        else printf("v %d ",x-n);
        printf("%c\n",'a'+y);
    }
}

int main()
```

```
{
    scanf("%d",&n);
    rep(i,1,n) scanf("%s",mp[i]+1);
    rep(i,1,n)
        rep(j,1,n){
            if(mp[i][j] == '?') continue;
            int x = mp[i][j]-'a';
            t[i].num[x]++;
            if(t[i].num[x] == 1) t[i].total++;
            t[j+n].num[x]++;
            if(t[j+n].num[x] == 1) t[j+n].total++;
        }
    solve();
    return 0;
}
```

5.5 欧拉路

题意：
给出 m 条边，求出一条欧拉路，起点任意，终点任意，每条边只经过一次。要求给出的欧拉路字典序最小。($1 \le m \le 1024, 1 \le n \le 500$)

思路：
先总结一下 '有向图、无向图求欧拉路与欧拉回路的性质 '。
无向图：有且仅有两个点度数为奇数则有欧拉路，所有点度数均为偶数则有欧拉回路。
有向图：所有点入度 = 出度则有欧拉回路。有且仅有两个点入度不等于出度，且起点出度比入度大 1 ，终点入度比出度大 1 则有欧拉路。

算法——Hierholzer (解决无向图、有向图、欧拉路、欧拉回路问题)

选一个点 x 为起点，存在边 $<x,y>$，则删去边 $<x,y>$，若为无向图还需删除 $<y,x>$。若无边可走，则将 x 加入结果栈。最后输出结构栈即可。

回到此题，要求找到字典序最小的欧拉路。因此选择一个编号最小的奇数点进行递归，递归过程中优先走字典序更小的点，可以用 $multiset$ 维护。

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <set>
#include <stack>
#include <algorithm>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl
    ;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2
    << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const int N = 500+100;
const int M = 1e5+100;
const db EPS = 1e-9;
using namespace std;

multiset<int> st[N];
```

```
stack<int> stk;
int m,deg[N];

void dfs(int x){
    while(st[x].size()){
        int y = (*st[x].begin());
        st[x].erase(st[x].begin());
        st[y].erase(st[y].find(x));
        dfs(y);
    }
    stk.push(x);
}

int main()
{
    rep(i,0,500) st[i].clear();
    while(stk.size()) stk.pop();
    scanf("%d",&m);
    rep(i,1,m){
        int xx,yy; scanf("%d%d",&xx,&yy);
        st[xx].insert(yy);
        st[yy].insert(xx);
        deg[xx]++, deg[yy]++;
    }
    int x = -1;
    rep(i,0,500) if(deg[i]%2){x = i; break;}
    if(x == -1) x = 1;
    dfs(x);
    while(stk.size()){
        printf("%d\n",stk.top());
        stk.pop();
    }
    return 0;
}
```

5.6 双连通分量

5.6.1 边双缩点以及求割边

求边双连通分量、割边（e-DCC）
将每个 e-DCC 看成一个节点，进行缩点
求边双连通分量——即该分量中不存在割边

边双连通图的条件——当且仅当任意一条边都包含在至少一个简单环中
求边双连通分量的原理：
先求出无向图中所有的桥，把桥都删除后，无向图会分成若干个连通块，每一个连通块就是一个“边双连通分量”

```
#include <cstdio>
#include <iostream>
#include <algorithm>
#include <cstring>
#define rep(i,a,b) for(int i = a;i <= b;i++)
using namespace std;
const int N = 1e5+10, M = 5*1e5+10;

struct Edge{
    int to,next;
}e[2*M],ec[2*M]; //双向边，ec为缩点之后的图【森林/树】
```

```
int head[N],dfn[N],low[N],headc[N];  //dfn:记录该点的dfs序      low:记录该点所能达到的最小dfn值
//hc:缩点之后记录每一个点所连接的第一条边的边序
bool bridge[2*M];   //记录该边是否为割边
int tot,n,m,num,dcc,tc; //tot记录边序，num记录dfs序，dcc记录有多少个边双连通分量
//tc:缩点之后记录边序
int c[N];  //c[x]表示节点x所属的“边双连通分量”的编号

void init()
{
    num  = 0; tot = 1; dcc = 0;
    rep(i,0,n) head[i] = dfn[i] = low[i] = c[i] = 0;
    rep(i,0,2*m+2) bridge[i] = 0;
}

void initc()
{
    tc = 1; //缩点之后记录边序
    // memset(headc,0,sizeof headc); //缩点之后记录每一个点所连接的第一条边的边序
    rep(i,0,dcc) headc[i] = 0;
}

void add(int x,int y)
{
    e[++tot].to = y; e[tot].next = head[x]; head[x] = tot;
}

void addc(int x,int y)  //对缩点之后的图进行加边操作
{
    ec[++tc].to = y; ec[tc].next = headc[x]; headc[x] = tc;
}

void tarjan(int x,int in_edge)
{
    dfn[x] = low[x] = ++num;
    for(int i = head[x]; i ; i = e[i].next)
    {
        int y = e[i].to;
        if(!dfn[y]){
            tarjan(y,i);  //传入边(x,y)的序号
            low[x] = min(low[x],low[y]);
            if(low[y] > dfn[x])  //该边连接(x,y)，y点无法连接x点上面的点，因此该边是割边
                bridge[i] = bridge[i^1] = true;
        }
        else if(i != (in_edge^1)) //y点所连接的边 不能是(x,y)边的反向边
            low[x] = min(low[x],dfn[y]);
    }
}

void dfs(int x) //用于将图划分为多个边双连通分量
{
    c[x] = dcc;
    for(int i = head[x]; i ; i = e[i].next)
    {
        int y = e[i].to;
        if(c[y] || bridge[i]) continue;  //如果点y已经属于别的强连通分量，或者边i是割边，则continue
        dfs(y);
    }
```

```
}

int main()
{
    while(~scanf("%d%d",&n,&m))
    {
        init();
        rep(i,1,m){
            int x,y;
            scanf("%d%d",&x,&y);
            add(x,y); add(y,x);
        }
        rep(i,1,n)
            if(!dfn[i]) tarjan(i,0); //将边序号传进去
        rep(i,1,n)
            if(!c[i]){ //如果i点未被标记过
                ++dcc;
                dfs(i);
            }
        initc(); //对缩点之后的图进行初始化 这里很重要! 不要忘记! ! !
        for(int i = 2; i <= tot; i += 2){ //遍历所有边 2-tot
            int x = e[i^1].to, y = e[i].to; //记录该边连接的两个端点
            if(c[x] == c[y]) continue; //如果连接的两个点属于同一连通分量, 则continue
            //没必要用mp记录,不会重复, 可以用ct记录有多少条边
            addc(c[x],c[y]);  //用连通的分量的编号来代表整个连通分量, 以此来进行缩点
            addc(c[y],c[x]);
        }
        printf("缩点之后的森林, 点数%d, 边数%d(可能有重边)\n",dcc,tc/2);
    /*  printf("There are %d e-DCCs.\n",dcc);
        rep(i,1,n){
            printf("%d belongs to DCC %d.\n",i,c[i]); //输出每个点属于的e-DCC
        }*/
    /*  for(int i = 2; i <= tot; i+=2)
            if(bridge[i])
                printf("%d %d\n",e[i^1].to,e[i].to);  //该割边: (x,y)*/
    }
    return 0;
}
```

5.6.2 边双缩点 LCA

题意:
给出一个无向图，n 个点，m 条边，可能有重边与自环，也可能不连通。q 组询问，每组询问给出 3 个点，u、v、w，问是否存在两条路径不存在公共边，并且一条路径是 $v \to u$，另一条路径是 $w \to u$，存在输出 Yes，否则输出 No. $(1 \le n \le 10^5, 0 \le m \le 2*10^5, 1 \le q \le 10^5)$

思路:
既然是无向图上求两条互不相交的路径，比较直接的想法就是先求出边双连通分量进行缩点，然后在树上进行考虑。

求出边双连通分量之后，假如 u、v、w 三点在同一个双连通分量中，则答案必定为 Yes。若 v、w 在同一个双连通分量中，而 u 在另一个双连通分量中，则答案必定为 No。若 v 或 w 和 u 在同一个双连通分量中，另一个点不在其中，则答案也为 Yes。考虑完了一个和两个双连通分量的情况之后，我们来考虑三个的情况。

假如 u、v、w 分属于三个不同的双连通分量中，则需要进行分类。Yes 的情况只有两种，第一种情况是 v 和 w 都在 u 子树中，即 $lca(v,w) = u$ 即可，如图 (1)。第二种情况是 v、w 中有一个在 u 的子树中，另一个则不在。对于这种情况，我们先求出 $lca(v,w) = y$，再求出 $x_1 = lca(u,w), x_2 = lca(u,v)$，则 x_1 与 x_2 中一定有一个为 y，另一个为 u，才能输出 Yes，否则输出 No. 到此，这题分类讨论就结束了。
但是这一题还需要注意一些细节，因为图可能不连通，因此需要预先判断 u、v、w 三个点是否连通，如果不连通，直接

输出 $No.$ 还有一个细节，因为图可能是个森林，因此需要对每一个树进行 lca 处理。

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <cmath>
#include <queue>
#include <algorithm>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const int N = 2*1e5+100;
const int M = 5*1e5+100;
const db EPS = 1e-9;
using namespace std;

struct Edge{
    int to,next;
}e[M],ec[M];
int n,m,q,head[N],dfn[N],low[N],headc[N],dis[N],d[N],f[N][20],t;
bool bridge[M];
int tot,num,dcc,tc,DD[N];
int c[N];

void init(){
    tot = tc = 1;
    num = dcc = 0;
    rep(i,0,n) head[i] = headc[i] = 0;
    rep(i,0,2*m) bridge[i] = 0;
    rep(i,0,n) c[i] = dfn[i] = low[i] = 0;
    rep(i,0,n) dis[i] = d[i] = DD[i] = 0;
}

void add(int x,int y){
    e[++tot].to = y; e[tot].next = head[x]; head[x] = tot;
}

void addc(int x,int y)
{
    ec[++tc].to = y; ec[tc].next = headc[x]; headc[x] = tc;
}

void tarjan(int x,int in_edge)
{
    dfn[x] = low[x] = ++num;
    for(int i = head[x]; i ; i = e[i].next)
    {
        int y = e[i].to;
        if(!dfn[y]){
            tarjan(y,i);  //传入边(x,y)的序号
            low[x] = min(low[x],low[y]);
            if(low[y] > low[x])  //该边连接(x,y)，y点无法连接x点上面的点，因此该边是割边
                bridge[i] = bridge[i^1] = true;
```

```
        }
        else if(i != (in_edge^1)) //y点所连接的边 不能是(x,y)边的反向边
            low[x] = min(low[x],dfn[y]);
    }
}

void dfsD(int x,int hp){
    DD[x] = hp;
    for(int i = head[x]; i; i = e[i].next){
        int y = e[i].to;
        if(DD[y] == 0) dfsD(y,hp);
    }
}

void dfs(int x) //用于将图划分为多个边双连通分量
{
    c[x] = dcc;
    for(int i = head[x]; i ; i = e[i].next)
    {
        int y = e[i].to;
        if(c[y] || bridge[i]) continue;  //如果点y已经属于别的强连通分量，或者边i是割边，则continue
        dfs(y);
    }
}

void bfs(int s)
{
    queue<int> q;
    while(q.size()) q.pop();
    q.push(s); d[s] = 1; dis[s] = 0;    //把1当做树根
    while(q.size())
    {
        int x = q.front(); q.pop();
        for(int i = headc[x]; i ;i = ec[i].next){
            int y = ec[i].to;
            if(d[y]) continue;
            d[y] = d[x]+1;
            dis[y] = dis[x]+1;  //dist[y]:从1到y的距离
            f[y][0] = x;  //y走2^0步到达x
            for(int j = 1; j <= t;j++)
                f[y][j] = f[f[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 = t; i >= 0; i--)
        if(d[f[y][i]] >= d[x]) y = f[y][i];  //往上追溯，直至y和x位于同一深度
    if(x == y) return x;  //如果已经找到了，就返回x
    for(int i = t; i >= 0; i--)
        if(f[x][i] != f[y][i]) x = f[x][i], y = f[y][i];  //x和y同时往上走，一直到x和y恰好为lca的子节点
    return f[x][0];  //x和y共同的根节点就是lca
}

```

```

int main()
{
    int _; scanf("%d",&_);
    while(_--){
        scanf("%d%d%d",&n,&m,&q);
        init();
        rep(i,1,m){
            int xx,yy; scanf("%d%d",&xx,&yy);
            add(xx,yy); add(yy,xx);
        }
        int ctt = 0;
        rep(i,1,n)
            if(!DD[i]) dfsD(i,++ctt);
        rep(i,1,n)
            if(!dfn[i]) tarjan(i,0); //将边序号传进去
        rep(i,1,n)
            if(!c[i]){ //如果i点未被标记过
                ++dcc;
                dfs(i);
            }
        rep(i,2,tot){
            int x = e[i^1].to, y = e[i].to; //记录该边连接的两个端点
            if(c[x] == c[y]) continue; //如果连接的两个点属于同一连通分量，则continue
            addc(c[x],c[y]);  //用连通的分量的编号来代表整个连通分量，以此来进行缩点
            addc(c[y],c[x]);
        }
        t = (int)(log(dcc+1)/log(2))+1;
        rep(i,1,dcc)
            if(d[i] == 0) bfs(i);
        rep(i,1,q){
            int u,v,w; scanf("%d%d%d",&u,&v,&w);
            if(DD[u] == DD[v] && DD[u] == DD[w]){
                if(c[u] == c[v] && c[u] == c[w]) printf("Yes\n");
                else if(c[v] == c[w] && c[v] != c[u]) printf("No\n");
                else if(c[u] != c[v] && c[u] != c[w] && c[v] != c[w]){
                    int y = lca(c[v],c[w]);
                    if(y == c[u]) printf("Yes\n");
                    else{
                        int x1 = lca(c[v],c[u]);
                        int x2 = lca(c[w],c[u]);
                        if(x1 == y && x2 == c[u]) printf("Yes\n");
                        else if(x2 == y && x1 == c[u]) printf("Yes\n");
                        else printf("No\n");
                    }
                }
                else if(c[v] == c[u] && c[w] != c[v]) printf("Yes\n");
                else if(c[w] == c[u] && c[w] != c[v]) printf("Yes\n");
                else printf("No\n");
            }
            else printf("No\n");
        }
    }
    return 0;
}
```

5.6.3 点双缩点以及求割点

v-DCC 的缩点
由于一个割点可能属于多个 v-DCC，因此缩点之后，图中包含 p+t 个节点，p 个割点，t 个 v-DCC，将每个 v-DCC 和每个割点都作为新图中的节点
并将每个割点与包含它的所有 v-DCC 之间连边

此处需要注意：
缩点之后，所有的点双连通分量变成一个点，该点只与割点相连，不会与其他点双连通分量相连

无向图中求割点
x 是割点当且仅当搜索树上存在 x 的一个子节点 y，满足
dfn[x] <= low[y]
即 y 点无法访问 x 点之上的点

如果 x 不是根节点，则只需要一个点 y 即可
如果 x 是根节点，则需要两个点 y1、y2 存在，x 才能是割点，此时如果去掉 x，则 y1 与 y2 不连通

dfn: 记录该点的 dfs 序　low: 记录该点所能达到的最小 dfn 值

无向图中求点双连通分量
操作方法：
1. 当一个节点第一次被访问时，把该节点入栈。
2. 当割点判定法则中的条件 dfn[x] <= low[y] 成立时，无论 x 是否为根，都要：
(1) 从栈顶不断弹出节点，直至节点 y 被弹出。
(2) 刚才弹出的所有节点与节点 x 一起构成一个 v-DCC

割点：
若 x 为割点，则从图中删去节点 x 及所有与 x 相关联的边之后，G 分裂成了两个或两个以上不相连的子图

点双连通分量 (v-DCC) ——分量中无割点
注意: 点双连通分量与“删除割点后图中剩余的连通块”是不一样的概念
桥不属于任何 e-DCC，但是割点可能属于多个 v-DCC

无向连通图是点双连通图所需要满足的条件：【其中一个】
1. 图的顶点数不超过 2.
2. 图中任意两点都同时包含在至少一个简单环中。其中“简单环”指的是不自交的环，也就是我们通常画出的环

```
#include <cstdio>
#include <iostream>
#include <algorithm>
#include <cstring>
#include <vector>
#define rep(i,a,b) for(int i = a;i <= b;i++)
using namespace std;
const int N = 1e5+10, M = 5*1e5+10;

struct Edge{
    int to,next;
}e[M*2],ec[M*2];  //双向边
int head[N],dfn[N],low[N],stack[N];  //dfn:记录该点的dfs序      low:记录该点所能达到的最小dfn值
//stack:手动模拟堆栈，先进先出
int n,m,tot,num,root,top,cnt; //top:用来记录stack的堆顶位置 cnt:记录一共有多少个点双连通分量
bool cut[N];  //记录该点是否为割点
vector<int> dcc[N]; //dcc[i]保存编号为i的v-DCC中的所有节点
```

```
int new_id[2*N];  //用于记录缩点之后的新图的各个点的编号
int headc[2*N],tc; //tc:记录新图的边的序号       headc:记录新图中每个点的第一条边

void init()
{
    tot = 1;  //记录第一条边
    num = 0;  //记录dfs序
    cnt = 0;  //记录点双连通分量个数
    top = 0;  //初始化堆栈
    memset(head,0,sizeof head);
    memset(dfn,0,sizeof dfn);
    memset(low,0,sizeof low);
    memset(cut,0,sizeof cut);
    rep(i,1,n){  //n个点
        dcc[i].clear();
    }
}

void initc()
{
    //cnt记录一共有多少个点双连通分量
    num = cnt;  //割点的编号从cnt+1开始，1-cnt的点为点双连通分量的编号
    tc = 1;
    memset(headc,0,sizeof headc);
}

void add(int x,int y)
{
    e[++tot].to = y; e[tot].next = head[x]; head[x] = tot;
}

void addc(int x,int y)
{
    ec[++tc].to = y; e[tot].next = headc[x]; headc[x] = tc;
}

void tarjan(int x)
{
    dfn[x] = low[x] = ++num;
    stack[++top] = x;   //当一个节点第一次被访问时，把该节点入栈
    if(x == root && head[x] == 0){  //孤立点-无边连接
        dcc[++cnt].push_back(x);    //孤立点自己就是一个点双连通分量
        return;
    }
    int flag = 0;   //记录存在几个子节点，使得low[y]>=dfn[x]，用于判断割点
    for(int i = head[x]; i ; i = e[i].next)
    {
        int y = e[i].to;
        if(!dfn[y]){
            tarjan(y);
            low[x] = min(low[x],low[y]);
            if(low[y] >= dfn[x]){
                flag++;
                if(x != root || flag > 1) cut[x] = true;    //x为割点
                cnt++;  //出现了一个新的点双连通分量
                int z;
                do{
                    z = stack[top--];   //x、y以及y之后的所有点构成一个点双连通分量
                    dcc[cnt].push_back(z);
```

```
                }while(z != y);
                dcc[cnt].push_back(x);
            }
        }
        else low[x] = min(low[x],dfn[y]);
    }
}

int main()
{
    while(~scanf("%d%d",&n,&m))
    {
        init();
        rep(i,1,m){
            int x,y;
            scanf("%d%d",&x,&y);
            if(x == y) continue; //判断自环
            add(x,y); add(y,x);
        }
        rep(i,1,n)
            if(!dfn[i]) root = i, tarjan(i);  //因为原图可能不连通，因此需要对于每个点进行dfs
        //以下是缩点操作
        initc();
        rep(i,1,n)
            if(cut[i]) new_id[i] = ++num;   //cut:记录该点是否为割点，记录割点i在新图中的编号为num+1
        //建新图，从每个v-DCC到它包含的所有割点连边
        rep(i,1,cnt) //cnt表示点双连通分量个数
            for(int j = 0; j < dcc[i].size(); j++)
            {
                int x = dcc[i][j];
                if(cut[x]){ //如果x为割点，则将x与其所属的点强连通分量相连
                    addc(i,new_id[x]);
                    addc(new_id[x],i);
                }
            //  else c[x] = i;  此处可有可无，此处的作用为c[x]表示x所在的强连通分量的编号
            }
        printf("缩点之后的森林，点数为%d，边数为%d\n",num,tc/2);    //num为新图中点数
        printf("编号 1~%d 的为原图的v-DCC，编号 > %d 的为原图割点\n",cnt,cnt);  //cnt为点双连通分量个数
        for(int i = 2;i < tc;i+=2)
            printf("%d %d\n",ec[i^1].to,e[i].to);   //输出新图中的所有无向边

    /*  rep(i,1,cnt){
            printf("e-DCC #%d:",i);
            for(int j = 0; j < dcc[i].size(); j++)
                printf(" %d",dcc[i][j]);  //将第i个点双连通分量中的点全部输出
            puts("");
        } */

    /*  rep(i,1,n)
            if(cut[i]) printf("%d ",i); //cut[i]标记i节点是否为割点
        puts("are cut-vertexes"); */
    }
    return 0;
}
```

5.6.4 点双例题

题意:
n 个点，m 条边
如果把第 i 个点去掉，将有多少对点不能互通

```
#include <cstdio>
#include <iostream>
#include <algorithm>
#include <cstring>
#define rep(i,a,b) for(int i = a;i <= b;i++)
using namespace std;
const int N = 1e5+10, M = 5*1e5+10;
typedef long long ll;

struct Edge{
    int to,next;
}e[M*2];
int head[N],dfn[N],low[N],siz[N];   //dfn:记录该点的dfs序   low:记录该点所能达到的最小dfn值
ll ans[N];
bool cut[N];    //记录该点是否为割点
int n,m,tot,num,root;

void init()
{
    tot = 1;
    memset(head,0,sizeof head);
    memset(cut,0,sizeof cut);
    memset(low,0,sizeof low);
    memset(dfn,0,sizeof dfn);
    memset(ans,0,sizeof ans);
    memset(siz,0,sizeof siz);
}

void add(int x,int y)
{
    e[++tot].to = y; e[tot].next = head[x]; head[x] = tot;
}

void tarjan(int x)
{
    dfn[x] = low[x] = ++num;  //记录步长
    siz[x] = 1;  //记录该点之下有几个点
    int flag = 0,sum = 0;  //标记有几个子节点 low[y] >= dfn[x]
    for(int i = head[x]; i ; i = e[i].next)
    {
        int y = e[i].to;
        if(!dfn[y])
        {
            tarjan(y);  //每一次被dfs，都是因为该点未被访问过
            siz[x] += siz[y];
            low[x] = min(low[x],low[y]);
            if(low[y] >= dfn[x]){
                flag++; //根节点需要满足两次这个条件     非根节点只需满足一次这个条件 即为割点
                ans[x] += (ll)siz[y]*(n-siz[y]);  //flag++说明如果去掉x，则y与外界不再连通
                sum += siz[y];
                if(x != 1 || flag > 1) cut[x] = true;
            }
        }
        else{
            //siz[x] += siz[y]; //此处不能加这句话，因为y点之前已经被访问过了，说明就算去掉x，y仍
```

```
    然可以连通，因此不能算入
            low[x] = min(low[x],dfn[y]);  //因为该点之前已经被搜过了
        }
    }
    if(cut[x])
        ans[x] += (ll)(n-sum-1)*(sum+1)+(n-1);
    else
        ans[x] = 2*(n-1);
}

int main()
{
    while(~scanf("%d%d",&n,&m))
    {
        init();
        rep(i,1,m){
            int x,y;
            scanf("%d%d",&x,&y);
            if(x == y) continue;
            add(x,y); add(y,x);
        }
        tarjan(1);  //这里不需要从每一个点进行dfs，是因为题目保证所有城镇连通
        rep(i,1,n)
            printf("%lld\n",ans[i]);
    }
    return 0;
}
```

5.7 强连通分量

5.7.1 强连通分量及缩点

求有向图中的强连通分量
将 SCC 中的强连通分量进行缩点

求解算法:
1. 当节点 x 第一次被访问时，把 x 入栈，初始化 low[x] = dfn[x]
2. 扫描从 x 出发的每条边 (x,y)
(1) 若 y 没被访问过，则说明 (x,y) 是树枝边，递归访问 y，从 y 回溯之后，令 low[x] = min(low[x],low[y]).
(2) 若 y 被访问过并且 y 在栈中，即 y 被访问过，并且 y 不属于之前的强连通分量，则令 low[x] = min(low[x],dfn[y]).
3. 从 x 回溯之前，判断是否有 low[x] = dfn[x]。若成立，则不断从栈中弹出节点，直至 x 出栈.

```
#include <cstdio>
#include <iostream>
#include <algorithm>
#include <cstring>
#include <vector>
#define rep(i,a,b) for(int i = a;i <= b;i++)
using namespace std;
const int N = 1e5+10, M = 1e6+10;

struct Edge{
    int to,next;
}e[M],ec[M];
int head[N],dfn[N],low[N];  //dfn:dfs的序号 low:该点能够到达的最小的dfn值的点
int stack[N],vis[N],c[N];   //stack:维护通过点x到达的所有点  vis[x]:标记点x是否在堆栈中  c[x]:
    记录点x所在的强连通分量的编号
vector<int> scc[N]; //scc[i]:记录第i个强连通分量中的所有节点
```

```
int n,m,tot,num,top,cnt;    //tot记录边的序号，num记录dfs序，top记录stack堆栈的顶点，cnt记录图中强连通分量个数
int tc,headc[N];

void init()
{
    tot = 1; num = 0;
    top = 0; cnt = 0;
    memset(head,0,sizeof head);
    memset(c,0,sizeof c);
    memset(vis,0,sizeof vis);
    memset(dfn,0,sizeof dfn);
    memset(low,0,sizeof low);
    rep(i,1,n)
        scc[i].clear();
}

void initc()
{
    tc = 1;
    memset(headc,0,sizeof headc);
}

void add(int x,int y)
{
    e[++tot].to = y; e[tot].next = head[x]; head[x] = tot;
}

void addc(int x,int y)
{
    ec[++tc].to = y; ec[tc].next = headc[x]; headc[x] = tc;
}

void tarjan(int x)
{
    dfn[x] = low[x] = ++num;
    stack[++top] = x, vis[x] = 1;   //标记点x在栈中
    for(int i = head[x]; i ; i = e[i].next)
    {
        int y = e[i].to;
        if(!dfn[y]){    //如果点y没有被访问过
            tarjan(y);  //从y点一直递归下去
            low[x] = min(low[x],low[y]);
        }
        else if(vis[y]) low[x] = min(low[x],dfn[y]);
        //在求双连通分量时，此处为dfn[y]，即y点被访问过，则可用y的dfn值来更新x的low值
        //但是在强连通分量中，图是有向图，因此vis[y]==0的点，要么没被访问过，要么已经出栈了，即已经属于另一块强连通分量中
        //若y点已经属于另一块强连通分量，由于是有向图，因此y无法与x构成环，所以不能用y点dfn值来更新x的low值
    }
    if(dfn[x] == low[x])    //low[x] == dfn[x]成立，表明栈中从x到栈顶的所有节点构成一个强连通分量
    {                       //这些节点无法与其他节点一起构成环
        cnt++; int y;
        do{
            y = stack[top--],vis[y] = 0;  //节点出栈
            c[y] = cnt, scc[cnt].push_back(y);  //将这些节点存入新的强连通分量中
        }while(x != y);
```

```
    }
}

int main()
{
    while(~scanf("%d%d",&n,&m))
    {
        init();
        rep(i,1,m){
            int x,y;
            scanf("%d%d",&x,&y);
            add(x,y);
        }
        rep(i,1,n)
            if(!dfn[i]) tarjan(i);
        //将每个SCC缩成一个点，对于原图中的每条有向边
        //如果c[x]!=c[y]，则在编号为c[x]和编号为c[y]的SCC之间连边
        //最后结果将是一张有向无环图
        initc();
        rep(x,1,n){
            for(int i = head[x]; i ;i = e[i].next){
                int y = e[i].to;
                if(c[x] == c[y]) continue;
                addc(c[x],c[y]);
            }
        }
    }
    return 0;
}
```

5.7.2 kosaraju

用两次 dfs 求解 SCC 问题

补充一个 DAG 性质：
1. 选出最小的顶点集合，使得任意点该集合可达 = 入度为 0 的点
2. 至少加几条边，使得 DAG 成为一个 SCC = max(入度为 0 的点个数，出度为 0 的点个数)
3. 至多加几条边，使得 DAG 不是一个 SCC = 极大非连通子图

极大非连通子图：
图由 x 和 y 两部分构成，其中 x 是一个完全有向图，y 也是一个完全有向图，然后 x 中的点全部连向 y
因此极大非连通子图的边数为 n*(n-1)-x*y，且 x+y=n，因此 x 和 y 的差值越大，xy 越小，总边数越多

```
#include <bits/stdc++.h>
const int N = 2e5;
using namespace std;

int n,c[N],dfn[N],vis[N],dcnt,scnt;
// dcnt - dfs序计数，scnt - scc计数
// dfn[i] - dfs后序列中第i个点
// c[i] - i 号点所在 scc 编号
vector<int> G1[N], G2[N]; // G1-原图，G2-反图

void dfs1(int x){
    vis[x] = 1;
    for(auto y:G1[x])
        if(!vis[y]) dfs1(y);
```

```
    dfn[++dcnt] = x;
}

void dfs2(int x){
    c[x] = scnt;
    for(auto y:G2[x])
        if(!c[y]) dfs2(y);
}

void kosaraju(){
    // 初始化
    dcnt = scnt = 0;
    memset(c,0,sizeof c);
    memset(vis,0,sizeof vis);
    // 第一遍 dfs
    for(int i = 1; i <= n; i++)
        if(!vis[i]) dfs1(i);
    // 第二遍 dfs
    for(int i = n; i >= 1; i--)
        if(!c[dfn[i]]) ++scnt, dfs2(dfn[i]);
}

int main()
{
    int m; scanf("%d%d",&n,&m);
    for(int i = 1; i <= m; i++){
        int a,b; scanf("%d%d",&a,&b);
        G1[a].push_back(b);
        G2[b].push_back(a);
    }
    kosaraju();
    return 0;
}
```

5.7.3 2-SAT

题意：
有一个小镇上只有一个牧师。这个小镇上有一个传说，在九月一日结婚的人会受到爱神的保佑，但是要牧师举办一个仪式。这个仪式要么在婚礼刚刚开始的时候举行，要么举行完婚礼正好结束。
现在已知有 n 场婚礼，告诉你每一场的开始和结束时间，以及举行仪式所需要的时间。问牧师能否参加所有的婚礼，如果能则输出一种方案。

```
#include <cstdio>
#include <iostream>
#include <algorithm>
#include <cstring>
#include <vector>
#define rep(i,a,b) for(int i = a;i <= b;i++)
using namespace std;
const int N = 1e5+10, M = 1e7;

struct Edge{
    int to,next;
}e[M];
int head[N],dfn[N],low[N];  //dfn:dfs的序号 low:该点能够到达的最小的dfn值的点
int stack[N],vis[N],c[N];   //stack:维护通过点x到达的所有点  vis[x]:标记点x是否在堆栈中  c[x]:
    记录点x所在的强连通分量的编号
```

```
vector<int> scc[N]; //scc[i]:记录第i个强连通分量中的所有节点
int n,m,tot,num,top,cnt;    //tot记录边的序号，num记录dfs序，top记录stack堆栈的顶点，cnt记录图中强连通分量个数
int s[N],t[N],d[N];

void init()
{
    tot = 1; num = 0;
    top = 0; cnt = 0;
    memset(head,0,sizeof head);
    memset(c,0,sizeof c);
    memset(vis,0,sizeof vis);
    memset(dfn,0,sizeof dfn);
    memset(low,0,sizeof low);
    rep(i,1,n)
        scc[i].clear();
}

void add(int x,int y)
{
    e[++tot].to = y; e[tot].next = head[x]; head[x] = tot;
}

void tarjan(int x)
{
    dfn[x] = low[x] = ++num;
    stack[++top] = x, vis[x] = 1;   //标记点x在栈中
    for(int i = head[x]; i ; i = e[i].next)
    {
        int y = e[i].to;
        if(!dfn[y]){    //如果点y没有被访问过
            tarjan(y);  //从y点一直递归下去
            low[x] = min(low[x],low[y]);
        }
        else if(vis[y]) low[x] = min(low[x],dfn[y]);
    }
    if(dfn[x] == low[x])    //low[x] == dfn[x]成立，表明栈中从x到栈顶的所有节点构成一个强连通分量
    {                       //这些节点无法与其他节点一起构成环
        cnt++; int y;
        do{
            y = stack[top--],vis[y] = 0;  //节点出栈
            c[y] = cnt, scc[cnt].push_back(y);  //将这些节点存入新的强连通分量中
        }while(x != y);
    }
}

int overlap(int a1,int b1,int a2,int b2)
{
    if(a1>=a2&&a1<b2 || b1>a2&&b1<=b2 || a1<=a2&&b1>=b2) return 1;
    return 0;
}

void solve()
{
    rep(i,1,n){
        if(c[i] == c[i+n]){
            printf("NO\n");
            return;
```

```
        }
    }
    printf("YES\n");
    int val[N];
    memset(val,0,sizeof val);
    rep(i,1,n){
        if(c[i] < c[i+n])   //Tarjan之后的图就已经进行了拓扑排序
            val[i] = 0;     //如果! a -> a, 则a为真
        else val[i] = 1;
    }
    rep(i,1,n){
        if(val[i] == 0){
            printf("%02d:%02d %02d:%02d\n",s[i]/60,s[i]%60,(s[i]+d[i])/60,(s[i]+d[i])%60);
        }
        else
            printf("%02d:%02d %02d:%02d\n",(t[i]-d[i])/60,(t[i]-d[i])%60,t[i]/60,t[i]%60);
    }
}

int main()
{
    while(~scanf("%d",&n))
    {
        init();
        rep(i,1,n){
            int x1,y1,x2,y2;
            scanf("%d:%d %d:%d %d",&x1,&y1,&x2,&y2,&d[i]);
            s[i] = x1*60+y1;
            t[i] = x2*60+y2;
        }
        //每个点都有两个状态，一个为真，一个为假，因此共2*n个状态
        rep(i,1,n){
            rep(j,i+1,n){
                if(overlap(s[i],s[i]+d[i],s[j],s[j]+d[j]))
                    add(i,j+n),add(j,i+n);
                if(overlap(s[i],s[i]+d[i],t[j]-d[j],t[j]))
                    add(i,j),add(j+n,i+n);
                if(overlap(t[i]-d[i],t[i],s[j],s[j]+d[j]))
                    add(i+n,j+n),add(j,i);
                if(overlap(t[i]-d[i],t[i],t[j]-d[j],t[j]))
                    add(i+n,j),add(j+n,i);
            }
        }
        rep(i,1,2*n)
            if(!dfn[i]) tarjan(i);
        solve();
    }
    return 0;
}
```

5.8 二分图匹配

```
#include <cstdio>
#include <iostream>
#include <cstring>
#define rep(i,a,b) for(int i = a;i <= b;i++)
```

```
using namespace std;
const int MAXN = 5010;  //点数最大值
const int MAXM = 50010; //边数最大值

struct Edge{
    int to,next; //next表示该边的上一条边的编号
}e[MAXM];
int head[MAXN],tot,uN;  //uN表示一共有多少个点
//head[u]表示以u为起点的边的编号
int girl[MAXN],used[MAXN];
//girl表示每个点的匹配情况
//used表示改点有没有被标记过

void init()
{
    tot = 0;
    memset(head,-1,sizeof head);
}

void addedge(int u,int v)
{
    tot++; //边的编号从1~n
    e[tot].to = v; e[tot].next = head[u]; //next表示与这条边共点的上一条边
    //如果next为-1，则表示该边无上一条边
    head[u] = tot;  //head[u]表示以u为起点的第一条边的编号
}

bool dfs(int u)
{
    for(int i = head[u]; i != -1;i = e[i].next){
        int v = e[i].to;
        if(!used[v])
        {
            used[v] = 1;
            if(!girl[v] || dfs(girl[v]))
            {
                girl[v] = u;
                return true;
            }
        }
    }
    return false;
}

int solve()
{
    int res = 0;
    memset(girl,0,sizeof girl);
    //点的编号 0 ~ uN-1
    for(int u = 0; u < uN;u++)
    {
        memset(used,0,sizeof used);
        if(dfs(u)) res++;
    }
    return res;
}

int main()
{
```

```
    //读入图
    printf("%d\n",solve());
    return 0;
}
```

5.9 第 K 小问题

5.9.1 S 到 T 的第 K 短路

```
#include <cstdio>
#include <iostream>
#include <algorithm>
#include <cstring>
#include <queue>
#define rep(i,a,b) for(int i = a;i <= b;i++)
using namespace std;
const int M = 1e5+100;
const int N = 2000;
const int inf = 0x3f3f3f3f;

int n,m,s,t,k;
struct Edge{
    int to,next,w;
}e[M],ef[M];
int head[N],headf[N],tot,totf;
int dis[N],book[N],dist[N];
struct Po{
    int id; //到达哪一个点
    int fw; //到该点已经走过的距离
    int w; //到该点已经走过的距离+该点距离终点的距离
}tmp;

bool operator < (Po x,Po y) //优先队列只能重载 <
{
    return x.w>y.w; //如果放在结构体内重载，需要加const
}
void init()
{
    tot = 1; totf = 1;
    memset(head,0,sizeof head);
    memset(headf,0,sizeof headf);
}

void add(int x,int y,int w)
{
    e[++tot].to = y; e[tot].next = head[x]; head[x] = tot; e[tot].w = w;
}

void addf(int x,int y,int w)
{
    ef[++totf].to = y; ef[totf].next = headf[x]; headf[x] = totf; ef[totf].w = w;
}

void spfa(int x)
{
    queue<int> q;
    memset(book,0,sizeof book);
    rep(i,1,n) dis[i] = inf;
    while(!q.empty()) q.pop();
```

```
    q.push(x);
    book[x] = 1;
    dis[x] = 0;
    while(!q.empty())
    {
        int p = q.front();
        for(int i = headf[p]; i ; i = ef[i].next)
        {
            if(dis[ef[i].to] > dis[p]+ef[i].w)
            {
                dis[ef[i].to] = dis[p]+ef[i].w;
                if(book[ef[i].to] == 0)
                {
                    q.push(ef[i].to);
                    book[ef[i].to] = 1;
                }
            }
        }
        book[p] = 0;
        q.pop();
    }
}

int bfs()
{
    if(dis[s] == inf) return -1;    //如果不加这句话，就是wa，确保连通性
    int val = 0;
    priority_queue<Po> q;
    while(!q.empty()) q.pop();
    tmp.fw = 0; //走到id这个点目前所需要的距离
    tmp.id = s;
    tmp.w = tmp.fw+dis[tmp.id];
    q.push(tmp);
    if(t==s)k++;    //这里也是一个大坑点
    while(!q.empty())
    {
        Po p = q.top();
        q.pop();    //优先队列，必须在push之前就pop，不然队列顶的那个点会被调到底下去
        int id = p.id;  //p所在的点
        if(id == t) val++;
        if(val == k) return p.fw;   //求第k短路
        for(int i = head[id]; i ; i = e[i].next)    //加入新边
        {
            tmp.fw = p.fw+e[i].w;
            tmp.id = e[i].to;
            tmp.w = tmp.fw+dis[tmp.id]; //此处为估价函数
            q.push(tmp);
        }
    }
    if(val!=k){return -1;}
}

int main()
{
    while(~scanf("%d%d",&n,&m))
    {
        init();
        rep(i,1,m)
        {
```

```
            int x,y,z;
            scanf("%d%d%d",&x,&y,&z);
            add(x,y,z);
            addf(y,x,z);
        }
        scanf("%d%d%d",&s,&t,&k);
        spfa(t);
        printf("%d\n",bfs());
    }
    return 0;
}
```

5.9.2 有向全图第 k 小路径

题意：给出一个 n 个点 m 条边的有向图，每条边都有一个权重。一共有 q 组询问，每次询问 k，表示询问图中第 k 小的路径。$(1 \le n, m, q, k \le 5 * 10^4)$

思路：q 组询问，我们离线下来，只用求出最大的那个 k 的答案即可。

然后一样的套路，定状态。一条路径的表示，无非是起点和终点，(u, v)，但是如果这样定状态的话，会发现不好更新，难道更新 v 的所有出边，如果更新 v 的所有出边很容易 T？

因此我们令 u 为 v 的前驱，状态定义为 (u, v, id, w) 表示 u 是 v 的前驱，且 v 是 u 的第 id 条出边，当前路径权重为 w。

然后每次更新 u 的第 $id + 1$ 条出边，或者更新 v 的第一条出边即可，记得将每个点的出边排序。

```
#include <bits/stdc++.h>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl
    ;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2
    << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const db EPS = 1e-9;
const int N = 5*1e4+10;
using namespace std;

int n,m,qn,Q[N],maxn;
ll ans[N];
vector<pair<ll,int> > v[N];
struct Node{
    int u,v,id;
    ll w;
    Node() {}
    Node(int x,int y,int d,ll z):u(x),v(y),id(d),w(z) {}
    bool operator < (Node xx) const {
        return w > xx.w;
    }
};

priority_queue<Node> q;

void solve(){
```

```
    while(q.size()) q.pop();
    int ct = 0;
    rep(i,1,n) sort(v[i].begin(),v[i].end());
    rep(i,1,n){
        if(v[i].size())
            q.push({i,v[i][0].second,0,v[i][0].first});
    }
    while(q.size()){
        Node x = q.top(); q.pop();
        ct++;
        ans[ct] = x.w;
        if(ct == maxn) break;
        if(v[x.v].size()){
            q.push({x.v,v[x.v][0].second,0,x.w+(v[x.v][0].first)});
        }
        if((int)v[x.u].size() > (int)(x.id+1)){
            q.push({x.u,v[x.u][x.id+1].second,x.id+1,x.w-v[x.u][x.id].first+v[x.u][x.id
    +1].first});
        }
    }
    rep(i,1,qn){
        printf("%lld\n",ans[Q[i]]);
    }
}

int main()
{
    int _; scanf("%d",&_);
    while(_--){
        scanf("%d%d%d",&n,&m,&qn);
        rep(i,0,n) v[i].clear();
        rep(i,1,m){
            int x,y; ll w; scanf("%d%d%lld",&x,&y,&w);
            v[x].push_back(make_pair(w,y));
        }
        maxn = 0;
        rep(i,1,qn){
            scanf("%d",&Q[i]);
            maxn = max(maxn,Q[i]);
        }
        solve();
    }
    return 0;
}
```

5.9.3 无向全图第 k 小最短路

题意：n 个点，m 条边的有权无向图，询问图中第 k 大的最短路，注意路径 (u,v) 被计算，仅当这条路径是 (u,v) 之间的最短路，保证有解。$(2 \le n, m \le 2 * 10^5, 1 \le k \le 400)$

思路：无向图中的问题，主要难点在于如何才能避免算重。

我们注意到此题是两点之间的最短路，而两点之间只会有一条最短路，路径端点相同且长度相同算一条。因此我们将路径两点存入 map，当且仅当路径两点第一次在一条路径中出现时，才更新答案。

因此我们令 u 为 v 的前驱，状态定义为 (u,v,id,w) 表示 u 是 v 的前驱，且 v 是 u 的第 id 条出边，当前路径权重为 w。

因此状态就是路径的起点和终点以及路径的权重，每次选择终点的出边进行状态更新即可。

```
#include <bits/stdc++.h>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const int N = 2*1e5+100;
const int M = 4*1e5+100;
const db EPS = 1e-9;
using namespace std;

int n,m,k;
vector<pair<ll,int> > v[N];
struct Node{
    ll w; int u,v;
    Node(ll a,int b,int c) : w(a), u(b), v(c) {}
    bool operator < (Node xx) const {
        return w > xx.w;
    }
};
map<pair<int,int>,int> mp;
vector<Node> temp;
priority_queue<Node> q;

void dijsktra(){
    int cnt = 0;
    mp.clear();
    while(q.size()) q.pop();
    rep(i,1,n) q.push({0,i,i});
    while(q.size()){
        Node tp = q.top(); q.pop();
        if(mp.find(make_pair(tp.u,tp.v)) != mp.end()) continue;
        if(tp.u != tp.v && mp.find(make_pair(tp.u,tp.v)) == mp.end() && mp.find(make_pair(tp.v,tp.u)) == mp.end()){
            cnt++;
            if(cnt == k){
                printf("%lld\n",tp.w);
                return;
            }
        }
        mp[make_pair(tp.u,tp.v)] = 1;
        int rt = 0;
        for(pair<ll,int> &thp:v[tp.v]){
            rt++;
            if(rt+cnt > k) break;
            int y = thp.second;
            if(mp.find(make_pair(tp.u,y)) != mp.end()) continue;
            q.push({tp.w+thp.first,tp.u,y});
        }
    }
}

int main()
```

```
{
    scanf("%d%d%d",&n,&m,&k);
    rep(i,1,m){
        int x,y; ll w; scanf("%d%d%lld",&x,&y,&w);
        temp.push_back({w,y,x});
    }
    sort(temp.begin(),temp.end());
    int siz = temp.size();
    for(int i = siz-1; i >= max(siz-1-k,0); i--){
        int x = temp[i].u, y = temp[i].v, w = temp[i].w;
        v[x].push_back(make_pair(w,y));
        v[y].push_back(make_pair(w,x));
    }
    rep(i,1,n) sort(v[i].begin(),v[i].end());
    dijsktra();
    return 0;
}
```

5.9.4 无向全图第 k 小团

题意：给出一个 n 个点的无向图，每个点都有一个权重，求出图中第 k 小的团，找不到输出 -1。团的定义为完全图。$(1 \le n \le 100, 1 \le k \le 10^6)$

思路：此类的问题的思路比较统一，先定一个状态，然后再将这个状态丢入堆中，去更新其它状态。

我们定状态为团的权重，并用 $bitset$ 记录团中所有出现的点，并记录每个团中编号最大的节点。

我们更新时每次选一个编号比当前团中任意节点都大的一个节点加入团中。判断一个点是否可以加入团，只需判断这个点的邻接矩阵和记录团中点的 $bitset$ 进行与运算，如果最终结果仍然是团中点的 $bitset$，则证明当前点和团中所有点都有连边，因此加入这个点仍可以构成一个团。

```
#include <bits/stdc++.h>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl
    ;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2
    << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const int N = 100+10;
const db EPS = 1e-9;
using namespace std;

char s[N];
int n,k;
ll w[N];
bitset<105> mp[N];
struct Node{
    ll w; int id;
    bitset<105> vis;
    Node() {w = id = 0;vis.reset();}
    bool operator < (Node xx) const {
        return w > xx.w;
    }
};
```

```

priority_queue<Node> q;

int main()
{
    scanf("%d%d",&n,&k);
    rep(i,1,n) scanf("%lld",&w[i]);
    rep(i,1,n){
        scanf("%s",s+1);
        rep(j,1,n)
            if(s[j] == '1') mp[i].set(j);
    }
    while(q.size()) q.pop();
    Node empty; ll ans = 0;
    q.push(empty);
    while(q.size()){
        Node x = q.top(); q.pop();
        k--;
        if(k == 0) {ans = x.w; break;}
        rep(i,x.id+1,n)
            if(x.vis[i] == 0 && (x.vis&mp[i]) == x.vis){
                Node nw = x;
                nw.w += w[i];
                nw.vis[i] = 1;
                nw.id = i;
                q.push(nw);
            }
    }
    if(k != 0) printf("-1\n");
    else printf("%lld\n",ans);
    return 0;
}
```

5.10 简单图构造

题意：
给出 n 个点，再给出每个点的出度，问能否构造一个没有自环的简单图，如果可以判断构造是否唯一，输出构造方案。$(2 \leq n \leq 100)$

思路：
运用 $Havel_Hakimi$ 定理，将点的度数排序，每次让度数最大的点向度数小的点连边，连完之后重新排序再次让度数最大的点向后连边，过程中不出现度数为负的点则可以构造。

然后问题就变成了如何判断是否有多种构造方式。我们只需在度数最大的点向后连边时，连最后一条边的时候查看后一个点的度数是否与该点度数相同，如果相同则说明有多种构造方案。

```
#include<stdio.h>
#include<string.h>
#include<algorithm>
using namespace std;
const int N=105;
struct node
{
    int d,id;
} e[N],e1[N]; //存点
int n,tot;
pair<int,int>p1[N*N],p2[N*N]; //存图的边
```

```
bool cmp(node a,node b)  //按度从大到小排序
{
    if(a.d==b.d) return a.id>b.id;
    return a.d>b.d;
}
bool Havel_Hakimi()  //判断是否可图化
{
    tot=0;
    for(int i=0;i<n-1;i++)
    {
        sort(e+i,e+n,cmp);
        if(i+e[i].d>=n) return 0;
        for(int j=i+1;j<=i+e[i].d;j++)
        {
            p1[tot++]=make_pair(e[i].id,e[j].id); //记录边
            e[j].d--;
            if(e[j].d<0) return 0;
        }
    }
    if(e[n-1].d!=0) return 0;
    return 1;
}
bool fun() //判断是否有其他成图方式
{
    int flag=0;
    int tot1=0;
    for(int i=0;i<n-1;i++)
    {
        sort(e1+i,e1+n,cmp);
        int tmp=e1[i].d+i;
        if(tmp<n-1&&e1[tmp].d==e1[tmp+1].d&&e1[tmp].d!=0)
        {
            flag=1;
            swap(e1[tmp].id,e1[tmp+1].id);
        }
        for(int j=i+1;j<=tmp;j++)
            {
                p2[tot1++]=make_pair(e1[i].id,e1[j].id);
                e1[j].d--;
            }
    }
    return flag;
}
int main()
{
    while(~scanf("%d",&n))
    {
        int sum=0;
        for(int i=0; i<n; i++)
        {
            scanf("%d",&e[i].d);
            e1[i].id=e[i].id=i+1;
            e1[i].d=e[i].d;
            sum+=e[i].d;
        }
        if(sum%2) printf("IMPOSSIBLE\n"); //如果度数和为奇数一定不能成图，一条边连两个点
        else
        {
            if(!Havel_Hakimi()) printf("IMPOSSIBLE\n"); //不可图化
```

```
            else
            {
                if(fun()) //可图化且图不唯一，输出两个图的信息
                {
                    printf("MULTIPLE\n");
                    printf("%d %d\n",n,tot);
                    for(int i=0;i<tot;i++)
                        printf("%d%c",p1[i].first,i==tot-1?'\n':' ');
                    for(int i=0;i<tot;i++)
                        printf("%d%c",p1[i].second,i==tot-1?'\n':' ');
                    printf("%d %d\n",n,tot);
                    for(int i=0;i<tot;i++)
                        printf("%d%c",p2[i].first,i==tot-1?'\n':' ');
                    for(int i=0;i<tot;i++)
                        printf("%d%c",p2[i].second,i==tot-1?'\n':' ');
                }
                else //图唯一
                {
                    printf("UNIQUE\n");
                    printf("%d %d\n",n,tot);
                    for(int i=0;i<tot;i++)
                        printf("%d%c",p1[i].first,i==tot-1?'\n':' ');
                    for(int i=0;i<tot;i++)
                        printf("%d%c",p1[i].second,i==tot-1?'\n':' ');
                    if(tot==0) printf("\n\n"); //特判图只有一个节点没有边，要输出两个换行
                }
            }
        }
    }
    return 0;
}
```

6 网络流

6.1 最大流

6.1.1 最小点覆盖

题意:
$N*N$ 的矩阵中有 K 个小行星，现在每行每列都有一个武器，可以消除此行或者此列中的所有小行星，问最少需要多少个武器可以将矩阵中所有小行星消除。

思路:
这是一个典型的最小点覆盖问题，可以用二分图匹配算法或者最大流算法进行解决。我们主要来讨论网络流的做法。

先来回顾一下最大流 $Dinic$ 的基础性质，先在残量网络上 BFS 求出所有节点的层次，构造了一个分层图。然后在分层图上 DFS 寻找增广路，在回溯时实时更新剩余容量。当在残量网络中 S 不能到达 T 时，算法结束。

因此最大流这个算法有个很重要的性质，即 == 最大流 = 最小割 = 最大匹配 = 最小点覆盖 ==，基本覆盖了大部分最大流应用的题目。

现在我们再回过头来看这题如何建图，这题就是用最少的点覆盖所有的边，典型的最小点覆盖题目。因此我们将其转化为最大匹配，即每个点只能选择一次，最多选择多少条边。因此对于一个在 (i,j) 位置的行星，将左边第 i 个点和右边第 j 个点相连，图中所有边流量均设为 1，跑最大流即可。

最后再来一个不严谨的”最大匹配 = 最小点覆盖”的证明。即当图中选的边已经达到了最大匹配之后，一定不存在一条未选的边，边的两个端点同时都没有被选。并且由于每个点只能被选择一次，因此每条边都代表只选了一个点，而这些点的集合覆盖了所有的边。不太严谨的证明，只是有利于记忆。详细证明可以网上继续查阅题解。

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#include <queue>
#define rep(i,a,b) for(int i = a;i <= b;i++)
using namespace std;
const int inf = 1<<29,N = 1000+10,M = 300500;  //处理1e4-1e5规模的网络

struct Edge{
    int to,next,v;
}e[M];
int n,m,s,t,k;  //顶点个数 边数 源点 汇点
int head[N],tot,dis[N],mp[N][N];
queue<int> q;

void init()   //千万别忘了初始化!
{
    tot = 1; memset(head,0,sizeof head);  //点的编号是2~n, 因为2^1 = 3, 3^1 = 2;  符合后续代码的操作
}

void add(int x,int y,int v)
{
    e[++tot].to = y; e[tot].next = head[x]; e[tot].v = v; head[x] = tot;
    e[++tot].to = x; e[tot].next = head[y]; e[tot].v = 0; head[y] = tot;  //反向边与正向边的流量之和为v
}

bool bfs()
{
```

```
    memset(dis,0,sizeof dis);
    while(!q.empty()) q.pop();
    q.push(s); dis[s] = 1;
    while(!q.empty())
    {
        int x = q.front(); q.pop();
        for(int i = head[x];i;i = e[i].next)
        {
            if(e[i].v && !dis[e[i].to]){
                q.push(e[i].to);


                dis[e[i].to] = dis[x]+1;
                if(e[i].to == t) return 1;  //找到一条路就return
            }
        }
    }
    return 0;
}

int dinic(int x,int flow) //找增广路
{
    if(x == t) return flow;
    int rest = flow,k;  //rest为输入的流量
    for(int i = head[x];i && rest; i = e[i].next)
    {
        if(e[i].v && dis[e[i].to] == dis[x]+1){
            k = dinic(e[i].to,min(rest,e[i].v));
            if(!k) dis[e[i].to] = 0;  //剪枝，去掉增广完毕的点
            e[i].v -= k;
            e[i^1].v += k;  //反向边加上flow，相当于我们可以反悔从这条路流过
            rest -= k; //k为能够被送出去的流量
        }
    }
    return flow-rest;  //总共被送出去了多少流量
}

int solve()
{
    int flow = 0,maxflow = 0;
    while(bfs())
        while((flow = dinic(s,inf))) maxflow += flow;
    return maxflow;
}

int main()
{
    while(~scanf("%d%d",&n,&k))
    {
        init();
        s = 1, t = 1+n+n+1;
        rep(i,1,k){
            int xx,yy; scanf("%d%d",&xx,&yy);
            add(xx+1,yy+1+n,1);
        }
        rep(i,1,n) add(s,i+1,1), add(1+n+i,t,1);
        printf("%d\n",solve());
    }
    return 0;
```

```
}
```

6.1.2 最大流 + 二分

题意:
给出了一个 $Y*X$ 的地图，在地图的四个边缘有门，用 $'D'$ 表示，$'X'$ 表示障碍物即不能走，$'.'$ 表示这个位置初始有一个人，现在地图中的所有人要逃出门外，每个 $'.'$ 这个点可以站好多人，但是出门的时候只能一个一个出，每个人移动一格的时间为 1，问最少需要多少时间，所有人可以撤出场地。如果不能撤出，输出 $impossible$ 。

思路:
因为是最少的时间，所以一开始想到的是最小割和最小费用最大流，然后发现均无法解决这个问题，因为难以解决每个时间每个门只能被经过一次这个问题。

后来发现我们可以 == 对每一个时刻的门分别建点 ==，然后使这个门的容量为 1，因此就可以满足每个时刻每个门只能出去一个人这个条件。

但是如果对门的所有时刻都建点的话，那么无疑跑出来的结果是不对的，即我们现在只能验证当前每一个时刻上限的情况下，所有人能不能出来。因此发现这个题的答案，即所需要的时间是单调的，因此可以进行二分答案。二分所有人出门的时间，然后用最大流建图来验证

当二分的时间为 x 时，则对每个门都建 x 个点，然后每个人到达这个门所需的时间如果为 y 的话，则将这个人与这个门所有 $>=y$ 的时刻连边。至此即可完成本题。

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#include <map>
#include <queue>
#define rep(i,a,b) for(int i = a;i <= b;i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl
    ;
using namespace std;
const int inf = 1<<29,N = 1e5+100,M = 5*1e6+100;  //处理1e4-1e5规模的网络

struct Edge{
    int to,next,v;
}e[M];
int n,m,s,t;  //顶点个数 边数 源点 汇点
int head[N],tot,dis[N];
queue<int> q;
char mp[20][20];
int numdoor,mandoor[200][200],numman,hp[200],vis[300];
int dir[4][2] = {{1,0},{-1,0},{0,1},{0,-1}};

void bbfs(int x)
{
    // printf("x:%d\n",x);
    queue<pair<int,int> > qp;
    while(qp.size()) qp.pop();
    memset(vis,0,sizeof vis);
    vis[x] = 1;
    qp.push(make_pair(x,0));
    while(qp.size()){
        int xx = qp.front().first, dist = qp.front().second;
        int yy = xx%m;
```

```
        qp.pop();
        xx = xx/m+1;
        rep(i,0,3){
            int xxx = xx+dir[i][0], yyy = yy+dir[i][1];
            int tp = (xxx-1)*m+yyy;
            if(xxx < 1 || xxx > n || yyy < 1 || yyy > m || mp[xxx][yyy] == 'X') continue;
            if(vis[tp]) continue;
            if(mp[xxx][yyy] == 'D'){
                // printf("oops!\n");
                // LOG2("x",x,"tp",tp);
                // LOG1("dist",dist);
                mandoor[hp[x]][hp[tp]] = dist+1;
                continue;
            }
            vis[tp] = 1;
            qp.push(make_pair(tp,dist+1));
        }
    }
}

void init_solve()
{
    memset(mandoor,0,sizeof mandoor);
    numman = numdoor = 0;
    memset(hp,0,sizeof hp);
    rep(i,1,n)
        rep(j,1,m)
            if(mp[i][j] == '.') hp[(i-1)*m+j] = ++numman;
            else if(mp[i][j] == 'D') hp[(i-1)*m+j] = ++numdoor;
    rep(i,1,n)
        rep(j,1,m)
            if(mp[i][j] == '.'){
                // printf("i:%d,j:%d\n",i,j);
                bbfs((i-1)*m+j);
            }
}

void init()   //千万别忘了初始化!
{
    tot = 1; memset(head,0,sizeof head);  //点的编号是2~n, 因为2^1 = 3, 3^1 = 2;  符合后续代码的操作
}

void add(int x,int y,int v)
{
    e[++tot].to = y; e[tot].next = head[x]; e[tot].v = v; head[x] = tot;
    e[++tot].to = x; e[tot].next = head[y]; e[tot].v = 0; head[y] = tot;  //反向边与正向边的流量之和为v
}

bool bfs()
{
    memset(dis,0,sizeof dis);
    while(!q.empty()) q.pop();
    q.push(s); dis[s] = 1;
    while(!q.empty())
    {
        int x = q.front(); q.pop();
```

```
        for(int i = head[x];i;i = e[i].next)
        {
            if(e[i].v && !dis[e[i].to]){
                q.push(e[i].to);
                dis[e[i].to] = dis[x]+1;
                if(e[i].to == t) return 1;  //找到一条路就return
            }
        }
    }
    return 0;
}

int dinic(int x,int flow) //找增广路
{
    if(x == t) return flow;
    int rest = flow,k;  //rest为输入的流量
    for(int i = head[x];i && rest; i = e[i].next)
    {
        if(e[i].v && dis[e[i].to] == dis[x]+1){
            k = dinic(e[i].to,min(rest,e[i].v));
            if(!k) dis[e[i].to] = 0;  //剪枝，去掉增广完毕的点
            e[i].v -= k;
            e[i^1].v += k;  //反向边加上flow，相当于我们可以反悔从这条路流过
            rest -= k; //k为能够被送出去的流量
        }
    }
    return flow-rest;  //总共被送出去了多少流量
}


int solve()
{
    int flow = 0,maxflow = 0;
    while(bfs())
        while(flow = dinic(s,inf)) maxflow += flow;
    return maxflow;
}

void mainsolve()
{
    int l = 1, r = 200, ans = -1;
    while(l <= r){
        int mid = (l+r)>>1;
        init();
        s = 1, t = 1+numman+mid*numdoor+1;
        rep(i,1,numman) add(s,i+1,1);
        rep(i,1,numdoor)
            rep(j,1,mid) add(1+numman+(i-1)*mid+j,t,1);
        rep(i,1,numman)
            rep(j,1,numdoor)
                if(mandoor[i][j] <= mid && mandoor[i][j])
                    rep(k,mandoor[i][j],mid) add(1+i,1+numman+(j-1)*mid+k,1);
        if(solve() == numman) r = mid-1, ans = mid;
        else l = mid+1;
    }
    if(ans == -1) printf("impossible\n");
    else printf("%d\n",ans);
}

```

```
int main()
{
    int _; scanf("%d",&_);
    while(_--)
    {
        scanf("%d%d",&n,&m);
        rep(i,1,n) scanf("%s",mp[i]+1);
        init_solve();
        mainsolve();
    }
    return 0;
}
```

6.1.3 最大流路径输出

题意：
给出 n 个起重机，每个起重机有两个属性，$W[i]$ 表示这个起重机的重量，$L[i]$ 表示这个起重机能够拉起的最大重量（可以拉重物也可以拉起重机）。现在有 m 栋楼以及 m 个重物，要求给出每栋楼起重机的分配方案，使得每栋楼最后留下的起重机可以拉起对应的重物。$(1 \le n \le 100, 1 \le M \le 100)$

思路：
比赛的时候以为所有的起重机都得用上... 然后就想到了费用流... 然后就是 $wa\ wa\ wa$... 其实这是一道比较简单的最大流问题，主要难点应该在路径输出。

我们先来考虑下如何建图。首先对于 n 个起重机进行拆点，拆点的目的是化边权为点权来限制流量。然后将源点 s 和所有 $W[i] = 0$ 的点的入点相连，流量为 1。每个拆开的点的入点和出点连一条流量为 1 的边。对于点 x 的出点与点 y 的入点，x 与 y 相连（流量为 1），当且仅当 $W[y] \le L[x]$。

再将 m 个重物与汇点相连，流量为 1。最后对于每个重物，将所有 $L[i]$ 大于重物重量的点与重物相连，流量为 1。然后直接跑最大流即可。如果最大流为 m，则输出路径，如果小于 m，则输出 $impossible$。

最后考虑本题最关键的步骤，求出最大流之后如何输出路径。

对于 m 个点，通过反向边往回跑，仅当反向边流量不为 0，而正向边流量为 0 时才走这条边，跑到源点即停止，进行 m 遍 dfs 即可。

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#include <queue>
#include <vector>
#define rep(i,a,b) for(int i = a;i <= b;i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl
    ;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2
    << " , " << z1 << ": " << z2 << endl;
using namespace std;
const int inf = 1<<29,N = 1000,M = 1e5;  //处理1e4-1e5规模的网络

struct Edge{
    int to,next,v,f;
}e[M];
int n,m,s,t;  //顶点个数 边数 源点 汇点
int head[N],tot,dis[N],L[N],W[N],T[N];
queue<int> q;
```

```
vector<int> ans[N];

void init()   //千万别忘了初始化!
{
    tot = 1; memset(head,0,sizeof head);  //点的编号是2~n, 因为2^1 = 3, 3^1 = 2;  符合后续代码的操作
}

void add(int x,int y,int v)
{
    e[++tot].to = y; e[tot].next = head[x]; e[tot].v = v; head[x] = tot; e[tot].f = 1;
    e[++tot].to = x; e[tot].next = head[y]; e[tot].v = 0; head[y] = tot; e[tot].f = -1;  //反向边与正向边的流量之和为v
}

bool bfs()  //判断是否有增广路径
{
    memset(dis,0,sizeof dis);
    while(!q.empty()) q.pop();
    q.push(s); dis[s] = 1;
    while(!q.empty())
    {
        int x = q.front(); q.pop();
        for(int i = head[x];i;i = e[i].next)
        {
            if(e[i].v && !dis[e[i].to]){
                q.push(e[i].to);
                dis[e[i].to] = dis[x]+1;
                if(e[i].to == t) return 1;  //找到一条路就return
            }
        }
    }
    return 0;
}

int dinic(int x,int flow) //dfs找增广路, 找到一条路, 就把增广的流量加到答案中
{
    if(x == t) return flow;
    int rest = flow,k;  //rest为输入的流量
    for(int i = head[x];i && rest; i = e[i].next)
    {
        if(e[i].v && dis[e[i].to] == dis[x]+1){
            k = dinic(e[i].to,min(rest,e[i].v));
            if(!k) dis[e[i].to] = 0;  //剪枝, 去掉增广完毕的点
            e[i].v -= k;
            e[i^1].v += k;  //反向边加上flow, 相当于我们可以反悔从这条路流过
            rest -= k; //k为能够被送出去的流量
        }
    }
    return flow-rest;  //总共被送出去了多少流量
}

int solve()
{
    int flow = 0,maxflow = 0;
    while(bfs())
        while((flow = dinic(s,inf))) maxflow += flow;   //while 循环内容 —— 判断是否还有增广路
    return maxflow;
```

```
}

void dfs(int x,int flag){ //路径输出
    if(1 < x && x <= 1+n) ans[flag].push_back(x-1);
    if(x == s) return;
    for(int i = head[x]; i; i = e[i].next)
        if(e[i^1].v == 0 && e[i].to <= 1+2*n && e[i].f == -1) dfs(e[i].to,flag);
}

int main()
{
    scanf("%d",&n); tot = 1;
    rep(i,1,n) scanf("%d%d",&W[i],&L[i]);
    scanf("%d",&m);
    rep(i,1,m) scanf("%d",&T[i]);
    s = 1, t = 2+2*n+m;
    rep(i,1,m) add(1+2*n+i,t,1);
    rep(i,1,n)
        if(W[i] == 0) add(s,i+1,1);
    rep(i,1,n) add(i+1,1+n+i,1);
    rep(i,1,n)
        rep(j,1,n)
            if(i != j && L[i] >= W[j]) add(i+1+n,1+j,1);
    rep(i,1,m)
        rep(j,1,n)
            if(L[j] >= T[i]) add(1+n+j,1+2*n+i,1);
    int maxflow = solve();
    if(maxflow != m) printf("impossible\n");
    else{
        rep(i,1,m)
            dfs(1+2*n+i,i);
        rep(i,1,m){
            for(int j = ans[i].size()-1; j >= 0; j--){
                printf("%d",ans[i][j]);
                if(j == 0) printf("\n");
                else printf(" ");
            }
        }
    }
    return 0;
}
```

6.1.4 最大权闭合子图

一、算法介绍
有一个有向图，每一个点都有一个权值（可以为正或负或 0），选择一个权值和最大的子图，使得每个点的后继都在子图里面，这个子图就叫最大权闭合子图。

能选的子图有 Ø,4,3,4,2,4,1,2,3,4, 它们的权值分别为 0,-1,5,-6,4.
所以最大权闭合子图为 3,4，权值为 5.

· 解法
这个问题可以转化为最小割问题，用网络流解决。
从源点 s 向每个正权点连一条容量为权值的边，每个负权点向汇点 t 连一条容量为权值的绝对值的边，有向图原来的边容量全部为无限大。

求它的最小割，割掉后，与源点 s 连通的点构成最大权闭合子图，权值为（正权值之和-最小割）。

· 如何理解
割掉一条边的含义
由于原图的边都是无穷大，那么割边一定是与源点 s 或汇点 t 相连的。

割掉 s 与 i 的边，表示不选择 i 点作为子图的点；
割掉 i 与 t 的边，表示选择 i 点为子图的点。

如果 s 与 i 有边，表示 i 存在子图中；
如果 i 与 t 有边，表示 i 不存在于子图中。

· 合法性
只有 s 与 t 不连通时，才能得到闭合子图。

如果 s 与 t 连通，则存在点 i,j，使得 s 到 i 有边，i 到 j 连通，j 到 t 有边，所以 j 一定是 i 的后继，但选择了 i，没有选择 j，不是闭合子图。

如果 s 与 t 不连通，选择了正权点 i，一定选择了 i 后继中的所有负权点。设 j 是 i 的后继中的正权点，则割掉 s 到 j 的边是没有意义的，最小割不会割掉它，则 j 一点被选中，所以 i 的所有后继都被选中，符合闭合图的定义。

· 最优性
最小割 =(不选的正权之和 + 要选的负权绝对值之和）
最大权闭合子图 =（正权之和-不选的正权之和-要选的负权绝对值之和）= 正权值和-最小割
因为正权值和，是定值，而最小割保证值最小，所以最大权闭合子图一定最优。

二、例题
题意: 一共有 m 个实验，n 个仪器，每个实验完成之后能得到 p_i 的报酬，每个仪器购买需要 v_i 的花费，每个实验需要依托某几个对应的器材才能完成。现要求给出购买仪器以及做实验的方案，使得净收益最大，给出具体方案与结果。$(1 \le n, m \le 50)$

思路: 以往普通费用流的问题，其最终的总流量都是固定的，因此可以依据总流量固定这一特点，对流量赋予费用来求解。

而此题的难点恰好在于此，涉及了费用，但最终总流量不固定，如果总流量不固定是很难用费用流算法来建图的，因此我们来考虑最大流。

而最大流通常解决的都是总流量问题，或者有时二分一下答案用最大流来判断是否合法，基本不涉及费用的问题，由此可以判断此题非之前遇到的普通最大流问题，是根据一种特殊的建图方式用最大流来求解出了费用问题。

其实这是一道最大权闭合子图问题，应用最大流最小割定理，通过求解最大流来获得最小割。建图方式如下：
1. 最大权闭合子图问题，针对的是在一个有向图中，每个点都有权值，现要在这个有向图中找到一个闭合子图，即子图中不存在出边不在子图中的点，使该子图中点权值和最大。
2. 具体建图方式则为，有向图中的边流量赋为无穷，源点连向权值为正的点，流量为其权值；权值为负的点连向汇点，流量为其权值绝对值。最终的答案为（正权值之和-最小割）

证明该定理的话，可以通过最小割只能断源点的连边或者汇点的连边来证明，即放弃正权点，或者接纳负权点。最大流结束后，仍与源点相连的点极为最大权闭合子图中的点。

如上建图即可跑出答案，最后的方案输出即 bfs 一遍找到仍与源点相连的点，此处可以依据最大流最后 bfs 的 dis 数组来判断。

总结: 通过此题我们学会了一个用最大流算法解决总流量不固定时的最大收益问题。

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#include <queue>
#define rep(i,a,b) for(int i = a;i <= b;i++)
using namespace std;
```

```cpp
const int inf = 1<<29,N = 2000+10,M = 300500;  //处理1e4-1e5规模的网络

struct Edge{
    int to,next,v;
}e[M];
int n,m,s,t;  //顶点个数 边数 源点 汇点
int head[N],tot,dis[N],mp[N][N];
queue<int> q;

void init()   //千万别忘了初始化!
{
    tot = 1; memset(head,0,sizeof head);  //点的编号是2~n, 因为2^1 = 3, 3^1 = 2;  符合后续代码的操作
}

void add(int x,int y,int v)
{
    e[++tot].to = y; e[tot].next = head[x]; e[tot].v = v; head[x] = tot;
    e[++tot].to = x; e[tot].next = head[y]; e[tot].v = 0; head[y] = tot;  //反向边与正向边的流量之和为v
}

bool bfs()
{
    memset(dis,0,sizeof dis);
    while(!q.empty()) q.pop();
    q.push(s); dis[s] = 1;
    while(!q.empty())
    {
        int x = q.front(); q.pop();
        for(int i = head[x];i;i = e[i].next)
        {
            if(e[i].v && !dis[e[i].to]){
                q.push(e[i].to);
                dis[e[i].to] = dis[x]+1;
                if(e[i].to == t) return 1;  //找到一条路就return
            }
        }
    }
    return 0;
}

int dinic(int x,int flow) //找增广路
{
    if(x == t) return flow;
    int rest = flow,k;  //rest为输入的流量
    for(int i = head[x];i && rest; i = e[i].next)
    {
        if(e[i].v && dis[e[i].to] == dis[x]+1){
            k = dinic(e[i].to,min(rest,e[i].v));
            if(!k) dis[e[i].to] = 0;  //剪枝, 去掉增广完毕的点
            e[i].v -= k;
            e[i^1].v += k;  //反向边加上flow, 相当于我们可以反悔从这条路流过
            rest -= k; //k为能够被送出去的流量
        }
    }
    return flow-rest;  //总共被送出去了多少流量
}

```

```
int solve()
{
    int flow = 0,maxflow = 0;
    while(bfs())
        while((flow = dinic(s,inf))) maxflow += flow;
    return maxflow;
}

vector<int> v1,v2;
string S;

int main()
{
    scanf("%d%d",&m,&n); getchar();
    init();
    s = n+m+1, t = n+m+2;
    int ans = 0;
    rep(i,1,m){
        getline(cin,S);
        int len = S.length(), cnt = 0, base = 0;
        rep(j,0,len-1){
            if(S[j] >= '0' && S[j] <= '9'){
                if(base == 0) cnt++;
                base = base*10+S[j]-'0';
            }
            else{
                if(cnt == 1) add(s,i,base), ans += base;
                else add(i,m+base,inf);
                base = 0;
            }
        }
        if(cnt == 1) add(s,i,base);
        else add(i,m+base,inf);
    }
    rep(i,1,n){
        int v; scanf("%d",&v);
        add(m+i,t,v);
    }
    ans -= solve();
    rep(i,1,m+n)
        if(dis[i] != 0){
            if(i <= m) v1.push_back(i);
            else v2.push_back(i-m);
        }
    int sz1 = v1.size(), sz2 = v2.size();
    rep(i,0,sz1-1) printf("%d%c",v1[i]," \n"[i==sz1-1]);
    rep(i,0,sz2-1) printf("%d%c",v2[i]," \n"[i==sz2-1]);
    printf("%d\n",ans);
    return 0;
}
```

6.1.5 最小路径覆盖

题意:
给出一个 n 个点, m 条边的 DAG, 现要在该图中找到最少的不相交路径, 求出具体路径数。$(1 \le n \le 150, 1 \le m \le 6000)$

思路:
最小路径覆盖问题，如果是第一次遇到该问题，可以先排除费用流的做法，因为总流量不固定，也正因为总流量不固定，

我们采取最大流算法。

将一个点拆成两个点，若存在边 (x,y)，则将左边的 x 连向右边的 y，最后答案为 $ans = n - maxflow$。接下来给出该算法的大致证明，在构建的图中，若选择了边 (x,y)，则意味着在原图中将这两点放在了一条路径中，相当于这两点所在的路径发生了合并，因此总路径数 -1，即最终 $ans = n - maxflow$。

此题还可以扩展到最少相交路径问题（POJ-2594）。而处理最少相交路径只需要将连边条件 (x,y) 从原图中存在边 (x,y) 改成原图中 (x,y) 可达即可，即可以跨越一个点进行路径合并。

```cpp
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#include <queue>
#define rep(i,a,b) for(int i = a;i <= b;i++)
using namespace std;
const int inf = 1<<29,N = 2000+10,M = 300500;  //处理1e4-1e5规模的网络

struct Edge{
    int to,next,v;
}e[M];
int n,m,s,t;  //顶点个数 边数 源点 汇点
int head[N],tot,dis[N],mp[N][N];
queue<int> q;

void dbg() {cout << "\n";}
template<typename T, typename... A> void dbg(T a, A... x) {cout << a << ' '; dbg(x...);}
#define logs(x...) {cout << #x << " -> "; dbg(x);}

void init()   //千万别忘了初始化!
{
    tot = 1; memset(head,0,sizeof head);  //点的编号是2~n，因为2^1 = 3, 3^1 = 2;  符合后续代码的操作
}

void add(int x,int y,int v)
{
    e[++tot].to = y; e[tot].next = head[x]; e[tot].v = v; head[x] = tot;
    e[++tot].to = x; e[tot].next = head[y]; e[tot].v = 0; head[y] = tot;  //反向边与正向边的流量之和为v
}

bool bfs()
{
    memset(dis,0,sizeof dis);
    while(!q.empty()) q.pop();
    q.push(s); dis[s] = 1;
    while(!q.empty())
    {
        int x = q.front(); q.pop();
        for(int i = head[x];i;i = e[i].next)
        {
            if(e[i].v && !dis[e[i].to]){
                q.push(e[i].to);
                dis[e[i].to] = dis[x]+1;
                if(e[i].to == t) return 1;  //找到一条路就return
            }
        }
```

```
    }
    return 0;
}

int dinic(int x,int flow) //找增广路
{
    if(x == t) return flow;
    int rest = flow,k;  //rest为输入的流量
    for(int i = head[x];i && rest; i = e[i].next)
    {
        if(e[i].v && dis[e[i].to] == dis[x]+1){
            k = dinic(e[i].to,min(rest,e[i].v));
            if(!k) dis[e[i].to] = 0;  //剪枝，去掉增广完毕的点
            e[i].v -= k;
            e[i^1].v += k;  //反向边加上flow，相当于我们可以反悔从这条路流过
            rest -= k; //k为能够被送出去的流量
        }
    }
    return flow-rest;  //总共被送出去了多少流量
}

int solve()
{
    int flow = 0,maxflow = 0;
    while(bfs())
        while((flow = dinic(s,inf))) maxflow += flow;
    return maxflow;
}

int to[N],deg[N];

int main()
{
    scanf("%d%d",&n,&m);
    init();
    s = 2*n+1, t = 2*n+2;
    int ans = 0;
    rep(i,1,m){
        int x,y; scanf("%d%d",&x,&y);
        add(x,n+y,1);
    }
    rep(i,1,n) add(s,i,1), add(i+n,t,1);
    ans = solve();
    rep(i,2,tot){
        int x = e[i^1].to, y = e[i].to;
        if(x <= n && y <= 2*n && e[i].v == 0){
            to[x] = y-n;
            deg[y-n]++;
        }
    }
    rep(i,1,n){
        if(!deg[i]){
            int pos = i;
            while(pos){
                printf("%d",pos);
                pos = to[pos];
                if(pos == 0) printf("\n");
                else printf(" ");
            }
```

```
        }
    }
    printf("%d\n",n-ans);
    return 0;
}
```

6.2 费用流

```
#include <cstdio>
#include <iostream>
#include <algorithm>
#include <cstring>
#include <queue>
#define rep(i,a,b) for(int i = a;i <= b;i++)
using namespace std;
const int N = 1100, M = 1e5+100;

struct Edge{
    int to,next,cap,cost; //cap为该边的容量，cost为该边的花费
}e[M];
int head[N],tot,s,t; //tot记录边数，s为源点，t为汇点
int dis[N],incf[N],pre[N],vis[N]; //dis为到每一个点的最短距离，incf为每一个点流入的流量
//pre为每一个点的最短距离是由哪一条边更新而来，vis标记该点有没有被访问过
int n,m,maxflow,ans;

void init()
{
    tot = 1; //从2开始“成对存储”，2和3是一对，4和5是一对
    memset(head,0,sizeof head);
}

void add(int x,int y,int z,int c) //z为容量，c为花费
{
    //正向边，初始容量为z，单位费用为c
    e[++tot].to = y; e[tot].next = head[x]; head[x] = tot; e[tot].cap = z; e[tot].cost = c;
    //反向边，初始容量为0，单位费用为-c，与正向边“成对存储”
    e[++tot].to = x; e[tot].next = head[y]; head[y] = tot; e[tot].cap = 0; e[tot].cost = -c;
}

bool spfa()
{
    //用最短路找增广路
    queue<int> q;
    //建议不要用memset赋值，太容易出错了，当dis设置为long long时
    //tmp还是一个int，就特别容易出错
    fill(dis,dis+N,1e8);
//  memset(dis,0x3f,sizeof dis); //memset是按字节赋值，一个字节8位，一个16进制位表示4个二进制位，因此0x3f为一个字节
    //此处dis赋值为inf，寻找最短路（最小费用流）
    //如果dis赋值为0xcf，则为寻找最长路（最大费用流）
    //0xcf为11001111，因此按位赋值之后为负数
    memset(vis,0,sizeof vis); //每个点都没被访问过
    q.push(s); dis[s] = 0; vis[s] = 1;
    incf[s] = 1<<30; //incf为每一个点流入的流量
    while(q.size())
    {
```

```
        int x = q.front(); vis[x] = 0; q.pop();
        for(int i = head[x]; i ; i = e[i].next) //通过与x相连的各边遍历与x相连的点
        {
            if(!e[i].cap) continue; //该边剩余流量为0，不在残量网络中，无法通过
            int y = e[i].to;
            if(dis[y] > dis[x] + e[i].cost)  //如果此处为 < ，则为最长路算法【最大费用流】
            {
                dis[y] = dis[x]+e[i].cost;
                incf[y] = min(incf[x],e[i].cap);
                pre[y] = i; //记录y点的最短路是由哪一条边更新而来的
                if(!vis[y]) vis[y] = 1, q.push(y);
            }
        }
    }
//  int tmp = 0x3f3f3f3f;
    if(dis[t] == 1e8) return false; //汇点不可达，已求出最大流
    return true;
}

void update()
{
    int x = t;
    while(x!=s)
    {
        int i = pre[x];
        e[i].cap -= incf[t];
        e[i^1].cap += incf[t];
        x = e[i^1].to; //相反边的终点，即为x的上一个点
    }
    maxflow += incf[t];
    ans += dis[t]*incf[t];  //dis[t]为这条最短路上的总花费，incf为这条最短路上的流
}

void solve()
{
    ans = 0; maxflow = 0;
    while(spfa()) update();
    printf("%d\n",ans);
}

int main()
{
    while(~scanf("%d%d",&n,&m))
    {
        //建图过程
        s = 1; t = 2+n;
        init();
        add(s,2,2,0);
        add(n+1,t,2,0);
        rep(i,1,m){
            int a,b,c;
            scanf("%d%d%d",&a,&b,&c);
            add(a+1,b+1,1,c); add(b+1,a+1,1,c);
        }
        solve();
    }
    return 0;
}

```

```
/*
常见错误:
    1.没有init();
    2.spfa()函数部分dis是long long型的，用memset赋值，再用int型的tmp比较...
    3.建议spfa()函数部分的dis用fill来进行赋值【不易出错】
*/
```

6.3 线性规划

题意:
从 N 个团队中挑人,其中第 i 个团队中的人能力值为 c[i]。现有 m 条约束,每条约束为 [l, r, c],表示从区间 [l, r] 的团队中选的总人数要小于等于 c。求在 m 条约束下,选出的人的能力值之和的最大值。($1 \le N \le 200, 1 \le M \le 4000, 0 \le c[i] \le 10^6$)

思路:
这是一个典型的线形规划问题，因此直接用单纯形算法求解即可。

```
#include <bits/stdc++.h>
using namespace std;

namespace simplex{
    using T = long double;
    const int N = 210, M = 4010;
    const T eps = 1e-7;
    int n, m;
    int Left[M], Down[M];
    //Ax<=b, max c^T x
    //result : v,xi=sol[i],1 based
    T a[M][N], b[M], c[N], v, sol[N];
    bool eq(T a, T b) {return fabs(a-b) < eps;}
    bool ls(T a, T b) {return a<b&&!eq(a,b);}
    void init(int p,int q)
    {
        n=p, m=q, v=0;
        for(int i = 1; i <= m; i++){
            for(int j = 1; j <= n; j++)
                a[i][j]=0;
        }
        for (int i=1;i<=m;i++) b[i]=0;
        for (int i=1;i<=n;i++) c[i]=sol[i]=0;
    }
    void input(){
        scanf("%d%d",&n,&m);
        for(int i = 1; i <= n; i++){
            int tp; scanf("%d",&tp);
            c[i] = (long double)tp;
        }
        for(int i = 1; i <= m; i++){
            int l,r,c; scanf("%d%d%d",&l,&r,&c);
            for(int j = l; j <= r; j++) a[i][j] = 1;
            b[i] = (long double)c;
        }
    }
    void pivot(int x, int y){
        swap(Left[x], Down[y]);
        T k = a[x][y]; a[x][y] = 1;
        vector<int> nz;
        for(int i = 1; i <= n; i++){
```

```
            a[x][i] /= k;
            if(!eq(a[x][i],0)) nz.push_back(i);
        }
        b[x] /= k;
        for(int i = 1; i <= m; i++){
            if(i==x||eq(a[i][y],0)) continue;
            k = a[i][y]; a[i][y] = 0;
            b[i] -= k*b[x];
            for(int j:nz) a[i][j] -= k*a[x][j];
        }
        if(eq(c[y],0)) return;
        k = c[y]; c[y] = 0;
        v += k*b[x];
        for(int i:nz) c[i] -= k*a[x][i];
    }
    //0:find solution,1:no feasible solution,2:unbounded
    int solve(){
        for(int i = 1; i <= n; i++) Down[i] = i;
        for(int i = 1; i <= m; i++) Left[i] = n+i;
        while(1){
            int x = 0, y = 0;
            for(int i = 1; i <= m; i++)
                if(ls(b[i],0)&&(x==0||b[i]<b[x])) x = i;
            if(x == 0) break;
            for(int i = 1; i <= n; i++)
                if(ls(a[x][i],0)&&(y==0||a[x][i]<a[x][y])) y = i;
            if(y == 0) return 1;
            pivot(x,y);
        }
        while(1){
            int x = 0, y = 0;
            for(int i = 1; i <= n; i++)
                if(ls(0,c[i])&&(!y||c[i]>c[y])) y = i;
            if(y == 0) break;
            for(int i = 1; i <= m; i++)
                if(ls(0,a[i][y]) && (!x || b[i]/a[i][y]<b[x]/a[x][y])) x = i;
            if(x == 0) return 2;
            pivot(x,y);
        }
        for(int i = 1; i <= m; i++) if(Left[i] <= n) sol[Left[i]] = b[i];
        return 0;
    }
}

int main()
{
    simplex::input();
    simplex::solve();
    printf("%lld\n", (long long)(simplex::v+0.5));
    return 0;
}
```

7 计算几何

7.1 计算几何板子

```cpp
#include <cstdio>
#include <iostream>
#include <cstring>
#include <vector>
#include <cmath>
#include <algorithm>
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x) cout << "x: " << x << endl;
#define LOG2(x,y) cout << "x: " << x << ", y: " << y << endl;
#define pi acos(-1.0)
#define cross(p1,p2,p3) ((p2.x-p1.x)*(p3.y-p1.y)-(p3.x-p1.x)*(p2.y-p1.y)) //向量(p1,p2)与(p1,p3)叉乘
#define crossOp(p1,p2,p3) sign(cross(p1,p2,p3)) //判断正负，顺时针为负，为0则代表三点共线
using namespace std;

//实数比较
typedef double db;
const db EPS = 1e-9;
inline int sign(db a) {return a < -EPS ? -1 : a > EPS; } //返回-1表示a < 0, 1表示a > 0, 0表示a = 0
inline int cmp(db a, db b) {return sign(a-b); } //返回-1表示a < b, 1表示a > b, 0表示 a==b

//点类
struct Point {
    db x,y;
    Point() {}
    Point(db _x, db _y) : x(_x), y(_y) {}
    Point operator+(Point p) { return {x+p.x, y+p.y}; }
    Point operator-(Point p) { return {x-p.x, y-p.y}; }
    Point operator*(db d) { return {x*d, y*d}; }
    Point operator/(db d) { return {x/d, y/d}; }
    Point rotleft() { return Point(-y,x); } //逆时针旋转90度
    Point rotright() { return Point(y,-x); } //顺时针旋转90度
    db dot(Point p) { return x*p.x+y*p.y; } //点积
    db det(Point p) { return x*p.y-y*p.x; } //叉积
    Point rot(db an) { return {x*cos(an)-y*sin(an),x*sin(an)+y*cos(an)}; }  //旋转
    db abs() { return sqrt(abs2()); }
    db abs2() { return x*x+y*y; }
    db disTo(Point p) { return (*this-p).abs(); }
    //此时在x负半轴上的点，排序结果是最小的。如果去掉sign(x)>=0，则排序结果是最大的
    int quad() const { return sign(y) == 1 || (sign(y) == 0 && sign(x) >= 0); } //判断该点是否在x轴上方或x轴上

    bool operator<(Point p) const {
        int c = cmp(x, p.x);
        if (c) return c == -1;  //先判断x大小
        return cmp(y, p.y) == -1;   //再判断y大小
    }
    bool operator==(Point p) const {
        return cmp(x, p.x) == 0 && cmp(y, p.y) == 0;
    }
    bool operator!=(Point p) const{
        return (cmp(x, p.x) || cmp(y,p.y));
    }
};
```

```cpp

db area(vector<Point> ps){  //凸包面积
    db ret = 0; rep(i,0,ps.size()-1) ret += ps[i].det(ps[(i+1)%ps.size()]);
    return ret/2;
}

db perimeter(vector<Point> ps){ //凸包周长
    db ret = 0; rep(i,0,ps.size()-1) ret += ps[i].disTo(ps[(i+1)%ps.size()]);
    return ret;
}

db dot(Point A, Point B, Point C){  //三点点积
    return (B-A).dot(C-A);
}

db rad(Point p1, Point p2){ //返回两个向量的夹角，范围为-2*pi~2*pi，顺时针转为负数，逆时针转为正数
    return atan2l(p1.det(p2),p1.dot(p2));
    //取绝对值就是旋转角度，否则会有正负区别，返回幅度制
}

/*
坐标系变换 —— 选中点集中的两点，两点构成的向量为x轴
    Point p1 = {AA[2].x-AA[1].x,AA[2].y-AA[1].y}; //选定两点作为向量
    rep(i,1,n){
        Point p2 = {AA[i].x-AA[1].x,AA[i].y-AA[1].y};
        db ang = rad(p1,p2);        //返回两个向量角度
        db len = p2.abs();
        Point tmp = {len*cos(ang),len*sin(ang)};    //得到新坐标系下的点坐标
        mp[tmp] = 1;  //可以用map存变更坐标轴之后的点
    }
*/

//求凸包
vector<Point> convexHull(vector<Point> ps) {
    int n = ps.size(); if(n <= 1) return ps;
    sort(ps.begin(),ps.end());
    vector<Point> qs(n*2); int k = 0;
    for(int i = 0; i < n; qs[k++] = ps[i++])
        while(k > 1 && crossOp(qs[k-2],qs[k-1],ps[i]) <= 0) --k;    //把 <= 改成 <，即可将凸包边上的点也包括在凸包中，不稳定凸包问题
    for(int i = n-2, t = k; i >= 0; qs[k++] = ps[i--])
        while(k > t && crossOp(qs[k-2],qs[k-1],ps[i]) <= 0) --k;
    qs.resize(k-1);
    return qs;
}

//最小矩形覆盖
db minRectangleCover(vector<Point> ps){
    //凸包点集顺序按逆时针
    int n = ps.size();
    if(n < 3) return 0.0;
    ps.push_back(ps[0]);
    db ans = -1;
    int r = 1, p = 1, q;
    for(int i = 0; i < n; i++){
        //求出离边 ps[i]-ps[i+1] 最远的点 r
        while(sign(cross(ps[i],ps[i+1],ps[r+1])-cross(ps[i],ps[i+1],ps[r])) >= 0) //叉积最大即为到点r到ps[i+1]-ps[i]这条边的距离最大
```

```
            r = (r+1)%n;
        //卡出 ps[i]-ps[i+1] 方向上正向 n 最远的点 p
        while(sign(dot(ps[i],ps[i+1],ps[p+1])-dot(ps[i],ps[i+1],ps[p])) >= 0)   //正向最远即为点积最大
            p = (p+1)%n;
        if(i == 0) q = p;
        //卡出 ps[i]-ps[i+1]方向上负向最远的点 q
        while(sign(dot(ps[i],ps[i+1],ps[q+1]) - dot(ps[i],ps[i+1],ps[q])) <= 0) //负向最大即为点积最小
            q = (q+1)%n;
        db d = ps[i].disTo(ps[i+1]);    //线段长度
        d = d*d;
        //叉积求出高，点积求出底边
        db tmp = cross(ps[i],ps[i+1],ps[r]) *
            (dot(ps[i],ps[i+1],ps[p])-dot(ps[i],ps[i+1],ps[q]))/d;
        if(ans < 0 || ans > tmp) ans = tmp;
    }
    return ans;
}

bool cmp1(Point a, Point b){        //按照角度排序，第四象限-第三象限-第二象限-第一象限
    if(a.quad() != b.quad()) return a.quad() < b.quad();
    else return sign(a.det(b)) > 0;
}

bool cmp2(Point a, Point b){    //第二种极角排序方式，用角度直接排
    return a.ang < b.ang;   //tp[j].ang = atan2(am[j].y-am[i].y,am[j].x-am[i].x)，返回幅度制，范围是-pi~pi
}

struct Line {
    Point s,e;
    Line() {}
    Line(Point _s, Point _e) : s(_s), e(_e) {}
    bool operator == (Line v) { return (s == v.s) && (e == v.e); }
    // 根据一个点和倾斜角 angle 确定直线，0 <= angle < pi
    Line(Point p, double angle) {
        s = p;
        if(sign(angle-pi/2) == 0) e = (s+Point(0,1));
        else e = (s+Point(1,tan(angle)));
    }
    // ax+by+c = 0
    Line(db a, db b, db c){
        if(!sign(a)) s = Point(0,-c/b), e = Point(1,-c/b);
        else if(!sign(b)) s = Point(-c/a,0), e = Point(-c/a,1);
        else s = Point(0,-c/b), e = Point(1,(-c-a)/b);
    }
    void adjust() { if(e<s) swap(s,e); }
    db length() { return s.disTo(e); }  //求线段长度
    db angle() {    //返回直线倾斜角 0 <= angle < pi，弧度制
        db k = atan2(e.y-s.y,e.x-s.x);
        if(sign(k) < 0) k += pi;
        if(sign(k-pi) == 0) k -= pi;
        return k;
    }
    Point crosspoint(Line v){ //求两直线交点
        db a1 = (v.e-v.s).det(s-v.s);
        db a2 = (v.e-v.s).det(e-v.s);
        return Point((s.x*a2-e.x*a1)/(a2-a1),(s.y*a2-e.y*a1)/(a2-a1));
```

```
    }
    // 点和直线关系，1在上方，2在下方，3在直线上
    int relation(Point p) {
        int c = sign((p-s).det(e-s));
        if(c < 0) return 1;
        else if(c > 0) return 2;
        else return 3;
    }
    // 点在线段上的判断
    bool pointonseg(Point p) { return sign((p-s).det(e-s)) == 0 && sign((p-s).dot(p-e)) <= 0; }
    // 两向量平行（对应直线平行或重合）
    bool parallel(Line v) { return sign((e-s).det(v.e-v.s)) == 0; }
    // 线段比较，用于map存直线
    bool operator<(Line l) const {
        if(s != l.s) return s < l.s;
        else return e < l.e;
    }
    //两线段相交判断
    //2 规范相交
    //1 非规范相交
    //0 不相交
    int segcrossseg(Line v){
        int d1 = sign((e-s).det(v.s-s));
        int d2 = sign((e-s).det(v.e-s));
        int d3 = sign((v.e-v.s).det(s-v.s));
        int d4 = sign((v.e-v.s).det(e-v.s));
        if((d1^d2) == -2 && (d3^d4) == -2) return 2;
        return (d1 == 0 && sign((v.s-s).dot(v.s-e)) <= 0) ||
            (d2 == 0 && sign((v.e-s).dot(v.e-e)) <= 0) ||
            (d3 == 0 && sign((s-v.s).dot(s-v.e)) <= 0) ||
            (d4 == 0 && sign((e-v.s).dot(e-v.e)) <= 0);
    }
    //点在线段上的判断
    bool pointonseg(Point p){
        return sign((p-s).det(e-s)) == 0 && sign((p-s).dot(p-e)) <= 0;
    }
};

struct circle{
    Point p; //圆心
    db r; //半径
    circle() {}
    circle(Point _p, db _r){
        p = _p;
        r = _r;
    }
    circle(Point a,Point b,Point c){
        Line u = Line((a+b)/2,((a+b)/2)+((b-a).rotleft()));
        Line v = Line((b+c)/2,((b+c)/2)+((c-b).rotleft()));
        p = u.crosspoint(v);
        r = p.disTo(a);
    }
};

int main()
{
    Point p1(1,3), p2(2,3), p3(1.5,1), p4(1.5,5);
    Line l1(p1,p2); l1.adjust();
```

```
    printf("%f\n",l1.angle());

    return 0;
}


/*
例题1: [hihocoder 1879]
    有n个点, 求锐角三角形的个数/总面积 (2018 北京icpc )
    n <= 2000, 可能三点共线
解法1:
    (n,3)-直角-钝角, O((n^2)logn), 因为锐角三角形会出现重复, 因此计算直角和钝角, 直角和钝角不会重复


例题2:
    点旋转, 用复数乘法来操作


例题3:
    n个点, 查看是否有三点共线
    枚举一个点, 将其它点进行极角排序, 查看是否有三点共线 O((n^2)logn)
    如果是整数点, 还可以进行哈希
    (x1,y1) (x2,y2)
    d1 = gcd(x1,y1)
    d2 = gcd(x2,y2)
    hash (x1/d1, y1/d1), (x2/d2,y2/d2)
*/
```

7.2 凸包三分

题意:
给出一个 N 个点的凸包，Q 次询问，每次给出一个 idx 和 k，表示现有一根长度无限的垂直 $(P[idx], P[idx+1])$ 向量且距离 $P[idx]$ 为 k 的垂直线。先问这根垂直线倒落时凸包上第一个被碰到的点，如果第一个被碰到的是直线，则输出两个点。$(1 \le n \le 10^5, 1 \le Q \le 10^5)$

思路:
稍微画图模拟一下就可以发现，最终倒在哪个点上取决于凸包上的点与这个点连线，连线之后与 $(P[idx], P[idx+1])$ 之间的夹角大小，夹角最大的点即为第一个碰到的点。若有两个相同大的点，即碰到的是直线。

这个夹角是先增后减的，因此不能进行二分，而是三分，而且是整数三分。而整数三分的精髓就是先三分到大致区间，然后在确定的小区间内暴力求解。

但是直接判断角度不好处理，我们转化成向量的叉乘进行计算，判断大小是变大还是变小即可。

总结:
1. 计算几何中判断大小的问题，尽量转化成叉积或者点积进行计算。

2. 整数三分是先三分到一个小范围，然后在小范围内直接暴力。

```
#include <bits/stdc++.h>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl
    ;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2
    << " , " << z1 << ": " << z2 << endl;
```

```
typedef long long ll;
typedef double db;
const int N = 1e5+100;
const int M = 1e5+100;
const db EPS = 1e-9;
#define cross(p1,p2,p3) ((p2.x-p1.x)*(p3.y-p1.y)-(p3.x-p1.x)*(p2.y-p1.y))
#define crossOp(p1,p2,p3) sign((cross(p1,p2,p3)))
using namespace std;

inline int sign(db a) {return a < -EPS ? -1 : a > EPS; }
inline int cmp(db a,db b) {return sign(a-b);}

struct Point{
    db x,y;
    Point() {}
    Point(db _x, db _y) : x(_x), y(_y) {}
    Point operator-(Point p) {return {x-p.x,y-p.y};}
    db dot(Point p) {return x*p.x+y*p.y;}
    db det(Point p) { return x*p.y-y*p.x; } //叉积
    db abs() { return sqrt(abs2()); }
    db abs2() { return x*x+y*y; }
    db disTo(Point p) { return (*this-p).abs(); }
}P[N];

int T,Q;

int calc(Point p2,Point p3,Point p4){
    Point t1 = p3-p2, t2 = p4-p2;
    return sign(t2.det(t1));
}

int main()
{
    scanf("%d%d",&T,&Q);
    rep(i,0,T-1) scanf("%lf%lf",&P[i].x,&P[i].y);
    int n = T;
    rep(i,1,Q){
        int pos; db x; scanf("%d%lf",&pos,&x);
        db len = P[pos].disTo(P[(pos+1)%n]);
        db tx = ((db)(P[(pos+1)%n].x-P[pos].x)*(db)x/len)+P[pos].x;
        db ty = ((db)(P[(pos+1)%n].y-P[pos].y)*(db)x/len)+P[pos].y;
        int l = 1, r = T-1, lmid, rmid;
        //三分先确定一个小范围
        while(r-l >= 15){
            lmid = l+(r-l)/3;
            rmid = r-(r-l)/3;
            if(calc({tx,ty},P[(pos+lmid)%n],P[(pos+rmid)%n]) == 1){
                l = lmid;
            }
            else r = rmid;
        }
        int indx1 = 1, indx2 = 0;
        rep(k,l,r){
            int jud = calc({tx,ty},P[(pos+k)%n],P[(pos+indx1)%n]);
            if(jud == -1) indx1 = k, indx2 = 0;
            else if(jud == 0) indx2 = k;
        }
        if(indx2 == 0) printf("%d\n",(indx1+pos)%n);
        else printf("%d %d\n",(indx1+pos)%n,(indx2+pos)%n);
```

```
    }
    return 0;
}
```

7.3 锐角三角形计数

题意：
给出 n 个点，先从中任意选取三个点，使得这三个点为一个锐角三角形，问一共有多少个锐角三角形。$(3 \le n \le 2000)$

思路：
正难则反，很明显正向思考这题难度非常高，因此考虑逆向思考，即能够构造出的直角和钝角三角形个数。

我们枚举一个点，然后按这个点对其它所有点进行极角排序，然后进行双指针枚举，枚举一个刚好变成直角的点，再枚举一个刚好不是钝角的点，然后直接计算答案即可，具体内容可以看代码。

```
#include <bits/stdc++.h>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl
    ;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2
    << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const int N = 2000+100;
const int M = 1e5+100;
const db EPS = 1e-9;
using namespace std;

inline int sign(db a) {return a < -EPS ? -1 : a > EPS;}

struct Point{
    ll x,y;
    Point() {}
    Point(ll _x, ll _y) : x(_x), y(_y) {}
    Point operator+(Point p) {return {x+p.x, y+p.y};}
    Point operator-(Point p) {return {x-p.x, y-p.y};}
    ll dot(Point p) {return x*p.x+y*p.y;} //点积
    ll det(Point p) {return x*p.y-y*p.x;} //叉积
    int quad() const { return sign(y) == 1 || (sign(y) == 0 && sign(x) >= 0);}
}P[N],Q[2*N];

db rad(Point p1, Point p2){
    return atan2l(p1.det(p2), p1.dot(p2));
}

bool cmp1(Point a, Point b){ //极角排序
    if(a.quad() != b.quad()) return a.quad() < b.quad();
    else return sign(a.det(b)) > 0;
}

int n;

int main()
{
    int _; scanf("%d",&_);
```

```
    while(_--){
        ll ans = 0;
        scanf("%d",&n);
        rep(i,1,n){
            scanf("%lld%lld",&P[i].x,&P[i].y);
        }
        int l = 0, r = 0;
        rep(i,1,n){
            int tot = 0;
            rep(j,1,n){
                if(j == i) continue;
                Q[++tot].x = P[j].x-P[i].x, Q[tot].y = P[j].y-P[i].y;
            }
            sort(Q+1,Q+1+tot,cmp1);
            // rep(j,tot+1,2*tot) Q[j].x = Q[j-tot].x, Q[j].y = Q[j-tot].y;
            int l = 1, r = 1;
            rep(j,1,tot){
                while(l <= tot && Q[j].det(Q[l]) >= 0 && Q[j].dot(Q[l]) > 0) l++;
                while(r <= tot && (Q[j].det(Q[r]) >= 0 || Q[j].dot(Q[r]) <= 0)) r++;
                ans += r-l;
            }
        }
        ans = ((n*(n-1ll)*(n-2ll))/6ll)-ans;
        printf("%lld\n",ans);
    }
    return 0;
}
```

7.4 点与凸包的切线

题意：
给定一个 n 个点的凸包，再给出 m 个光照点，每个光照点的照射范围为 360 度，问最少选取几个光照点可以照亮整个凸包，要求输出方案，保证不会出现一个光照点位于凸包的延长线上，共 200 组数据。$(1 \le n, m \le 1000)$

思路：
其实这题思路比较明显，就是先求出每个光照点所能照射到的一段连续范围，然后问题就变成了给定一个长度为 n 的环形区域，以及 m 段区间，要求选取最少的区间覆盖整个区域。

首先求每个光照点对应的一段连续区域，比赛时的思路是极角排序，然后选择最两边的边。这样的复杂度是 $O(n^2logn)$，再加上 200 组数据，成功 TLE... 其实仔细思考一下可以发现并不需要进行极角排序，利用叉积即可解决该问题。因为照射范围的两个端点一定是光照点与凸包点连线形成的直线中最两边的两个点。

因此其它所有端点与光照点连线形成的直线都是顺时针或逆时针才能到达两个端点，即照射范围的两个端点是顺时针旋转的两个边界点，因此直接用叉积判断即可。

求出每个光照点范围之后，就只需要处理区间覆盖问题了。环形区域，我们只需要枚举起始点，$O(n^2)$ 求解即可。我们记录 $add[i]$ 数组表示所有覆盖第 i 条的光照点中所能覆盖的最远距离，$id[i]$ 记录光照点编号，然后直接贪心即可解决该问题。

```
#include <bits/stdc++.h>
#define mem(a,b) memset(a,b,sizeof a);
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define per(i,a,b) for(int i = a; i >= b; i--)
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
typedef long long ll;
typedef double db;
#define pi acos(-1.0)
```

```
const int N = 1e3+100;
const db EPS = 1e-9;
using namespace std;

void dbg() {cout << "\n";}
template<typename T, typename... A> void dbg(T a, A... x) {cout << a << ' '; dbg(x...)
    ;}
#define logs(x...) {cout << #x << " -> "; dbg(x);}
inline int sign(db a) {return a < -EPS ? -1 : a > EPS;}
inline int cmp(db a,db b) {return sign(a-b);}

struct Point{
    ll x,y;
    Point operator-(Point p) {return {x-p.x,y-p.y};}
    ll dot(Point p) {return x*p.x+y*p.y;}
    ll det(Point p) {return x*p.y-y*p.x;} //叉积
}P[N],H[N];

int n,m,L[N],R[N],add[N],id[N],ans;
vector<int> base;

void init(){
    scanf("%d%d",&n,&m);
    rep(i,0,n-1) scanf("%lld%lld",&P[i].x,&P[i].y);
    rep(i,0,m-1) scanf("%lld%lld",&H[i].x,&H[i].y);
    rep(i,0,n-1) add[i] = id[i] = 0;
    rep(i,0,m-1){
        L[i] = 0, R[i] = 0;
        rep(j,1,n-1){
            //利用叉积求切线
            if((P[L[i]]-H[i]).det(P[j]-H[i]) > 0) L[i] = j;
            if((P[R[i]]-H[i]).det(P[j]-H[i]) < 0) R[i] = j;
        }
        int len = (R[i]-L[i]+n)%n;
        rep(j,0,len-1){
            int now = (L[i]+j)%n;
            int tlen = (R[i]-now+n)%n;
            if(tlen > add[now]) add[now] = tlen, id[now] = i;
        }
    }
}

void solve(){
    ans = 1e5; base.clear();
    rep(i,0,n-1){
        int pos = i, num = 0, left = n;
        vector<int> hp; hp.clear();
        while(left > 0){
            if(add[pos] == 0) {num = 1e5; break;}
            left -= add[pos];
            hp.push_back(id[pos]);
            pos = (pos+add[pos])%n;
            num++;
        }
        if(num < ans){
            ans = num;
            base = hp;
        }
    }
```

```
    if(ans == 1e5) printf("-1\n");
    else{
        printf("%d\n",ans);
        rep(i,0,ans-1) printf("%d%c",base[i]+1," \n"[i==ans-1]);
    }
}

int main()
{
    int _; scanf("%d",&_);
    while(_--){
        init();
        solve();
    }
    return 0;
}
```

7.5 动态维护上凸壳

题意：
主人公小智一共会捕捉 n 只宝可梦，宝可梦有两个属性，攻击值 A，防御值 B。每当捕捉到一只新的宝可梦 T，小智有两种方法判断这只宝可梦是否是无用的。
1. 存在一只宝可梦 X，使得 $X.A > T.A$ 并且 $X.B > T.B$
2. 存在两只宝可梦 X, Y，使得 $c * X.A + (1-c) * Y.A > T.A$ 并且 $c * X.B + (1-c) * Y.B > T.B$，$0 \le c \le 1$
每当捕捉到一只新宝可梦，需要输出当前共有几只无用宝可梦。$(1 \le n \le 10^5, 0 \le A_i, B_i \le 10^9)$

思路：
参数 c 的约束其实为线段的参数方程，可以转换成动态维护上凸壳进行求解。

```
#include <bits/stdc++.h>
#define rep(i,a,b) for(int i = a; i <= b; i++)
typedef long long ll;
using namespace std;

struct Node{
    ll x,y;
    Node operator - (Node p) const {return {x-p.x,y-p.y};}
    ll operator ^ (Node p) const {return x*p.y-y*p.x;}
    bool operator < (Node p) const {return (x == p.x ? y > p.y : x < p.x);}
};

struct Hull : public multiset<Node>{
    bool inside(iterator p){
        auto t2 = next(p);
        if(t2 == end()) return 0;
        if(p == begin()) return t2->x > p->x && t2->y > p->y;
        auto t1 = prev(p);
        if(((*t1-*p)^(*t2-*p)) < 0) return 1;
        else return 0;
    }
    void ins(Node p){
        auto t = insert(p);
        if(inside(t)) { erase(t); return; }
        while(t != begin() && inside(prev(t))) erase(prev(t));
        while(next(t) != end() && inside(next(t))) erase(next(t));
    }
};
```

```

int main()
{
    int n; scanf("%d",&n);
    Hull hull;
    rep(i,1,n){
        ll x,y; scanf("%lld%lld",&x,&y);
        hull.ins({x,y});
        printf("%d\n",i-(int)hull.size());
    }
    return 0;
}
```

7.6 欧拉平面图公式

题意：
给出 n 条线段，求平面中一共有多少个闭合区域。保证不存在三条线段相交与一点，并且线段都是规范相交，即不存在相交点在线段端点。$(1 \le n \le 1000)$

思路：
此题是一个公式题，因此直接上公式。

基本术语：
平面图：不存在边交叉的二维平面上的图
连通平面图：平面图上的图形相互连通，即可以仅依靠图中的边从任意一个节点到达另外一个节点

欧拉平面图公式：
1. 基本公式（针对连通平面图）
· $V - E + F = 2$，其中 V 为点数，E 为边数，F 为面数（即闭合区域）
· 此处的 F 包括最外层的面
· 此公式可用数学归纳法证明，证明过程简单易懂，读者可自行查阅
2. 广义公式（针对平面图）
· $V - E + F = 1 + C$，其中 C 为平面图中的连通块个数
· 此公式可根据基本公式推出，推导过程较为简单，读者可自行思考

本题做法：
由于本题仅给出了线段，并不满足平面图的限制，因此我们可以将线段的交点当作一个新的点，将原来的图形转化成平面图。
因此在转化后的图中，点数变为 $(2 * n + inter)$，边数变为 $(n + 2 * inter)$，其中 $inter$ 为直线交点个数，$comp$ 为连通块个数，即最终答案如下：（本题忽略最外层的点）

$$F = C + E - V = comp + (n + 2 * inter) - (2 * n + inter) = comp + inter - n$$

```
#include <bits/stdc++.h>
#define mem(a,b) memset(a,b,sizeof a);
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define per(i,a,b) for(int i = a; i >= b; i--)
typedef long long ll;
const int N = 1e4+100;
using namespace std;

int n, inter, vis[N], comp;
vector<int> G[N];

inline int sgn(ll x) { return x >= 0 ? x > 0 : -1;}

```

```
struct Point{
    ll x,y;
    Point() {}
    Point(ll _x, ll _y) : x(_x), y(_y) {}
    Point operator-(Point p) { return {x-p.x, y-p.y}; }
    ll det(Point p) { return x*p.y-y*p.x; } //叉积
    ll dot(Point p) { return x*p.x+y*p.y; } //点积
}P1[N],P2[N];

struct Line{
    Point s,e;
    Line() {}
    Line(Point _s, Point _e){
        s = _s;
        e = _e;
    }
    int segcrosseg(Line v){
        int d1 = sgn((e-s).det(v.s-s));
        int d2 = sgn((e-s).det(v.e-s));
        int d3 = sgn((v.e-v.s).det(s-v.s));
        int d4 = sgn((v.e-v.s).det(e-v.s));
        if(((d1^d2) == -2) && (d3^d4) == -2) return 2;
        return (d1==0 && sgn((v.s-s).dot(v.s-e)) <= 0) ||
               (d2==0 && sgn((v.e-s).dot(v.e-e)) <= 0) ||
               (d3==0 && sgn((s-v.s).dot(s-v.e)) <= 0) ||
               (d4==0 && sgn((e-v.s).dot(e-v.e)) <= 0);
    }
};

void dfs(int x){
    if(vis[x]) return;
    vis[x] = 1;
    for(auto y:G[x]) dfs(y);
}

int main()
{
    scanf("%d",&n);
    rep(i,1,n) scanf("%lld%lld%lld%lld",&P1[i].x,&P1[i].y,&P2[i].x,&P2[i].y);
    rep(i,1,n){
        Line l1 = {P1[i], P2[i]};
        rep(j,i+1,n){
            Line l2 = {P1[j], P2[j]};
            if(l1.segcrosseg(l2) != 0){
                G[i].push_back(j);
                G[j].push_back(i);
                inter++;
            }
        }
    }
    rep(i,1,n)
        if(!vis[i]){
            comp++;
            dfs(i);
        }
    printf("%d\n", inter+comp-n);
    return 0;
}
```

8 其他

8.1 三分

```
#include <cstdio>
#include <iostream>
#include <cstring>
#include <algorithm>
#define rep(i,a,b) for(int i = a; i <= b; i++)
using namespace std;
const int N = 20;
const double eps = 1e-7;
double coe[N];
int n;

double func(double x)
{
    double ans = 0;
    double base = 1;
    rep(i,0,n)
    {
        ans += base*coe[i];
        base *= x;
    }
    return ans;
}

void solve(double l, double r)
{
    while(l + eps < r)
    {
        double m1 = (2*l+r)/3, m2 = (2*r+l)/3;
        double r1 = func(m1), r2 = func(m2);
        if(r1 >= r2)
            r = m2;
        else
            l = m1;
    }
    printf("%.5f\n",r);
}

int main()
{
    double l,r;
    scanf("%d",&n);
    scanf("%lf%lf",&l,&r);
    for(int i = n; i >= 0; i--)
        scanf("%lf",&coe[i]);
    solve(l,r);
    return 0;
}
```

8.2 bitset 优化暴力

题意：
给出一个 $n*m$ 的矩阵，矩阵每个点为 1 或 0，每行每列均可翻转且只能翻转一次。问能否将矩形通过翻转变成一个从 $a_{1.1}...a_{1.m}$ $a_{2.1}...a_{n.m}$ 数值不下降的状态，如果可以给出每行每列的翻转状态，否则输出 NO。$(1 \le n, m \le 200)$

思路：
稍微考虑一下这个题目，就可以发现此题难点主要在于一个点是否翻转由该点所代表的行列同时决定，因此我们难以进行判断。

所以我们可以思考能否事先确定行或列的状态，这样判断起来就很方便了。然后就可以发现只要确定了矩阵的最终状态，再确定了第一行的状态，我们就可以确定每一列的状态。确定了每一列状态和矩形最终状态，就可以不断向下循环确定每一行的状态，如果一直都不发生冲突，则为 YES。

我们来考虑一下复杂度。首先枚举矩形的最终状态，即从 (n,m) 向 $(1,1)$ 不断填 1，因此一共有 $n*m$ 个最终状态。对于每个最终状态，我们枚举第一行是否翻转，然后确定每一列的状态，再对剩下的每一行进行判断。因此复杂度为 $O(n*m*n*m)$。

这样的复杂度肯定是会 T 的，因此考虑 $bitset$ 优化，对于每一行直接按位异或得到答案，可以将复杂度优化到 $O(n*m*n*m)/32$，因此可以通过此题。具体异或细节见代码。

```
#include <cstdio>
#include <cstdlib>
#include <iostream>
#include <cstring>
#include <bitset>
#include <algorithm>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
typedef double db;
const int N = 200+10;
const int M = 1e5+100;
const db EPS = 1e-9;
using namespace std;

bitset<N> row,col,norml[2];
bitset<N> mp[N],jud;
bitset<N> base[N];
int n,m;

void test()
{
    col = mp[0]^base[0]^norml[row[0]]; //枚举第一行状态，确定每一列是否翻转
    rep(i,1,n-1){
        jud = base[i]^mp[i]^col; //判断这一行的row是否翻转
        if(jud.count() == m) row.set(i,1);
        else if(jud.count() == 0) row.set(i,0);
        else return;
    }
    printf("YES\n");
    rep(i,0,n-1) printf("%d",row[i]==1); printf("\n");
    rep(i,0,m-1) printf("%d",col[i]==1); printf("\n");
    exit(0);
}

int main()
{
```

```
    scanf("%d%d",&n,&m);
    rep(i,0,n-1)
        rep(j,0,m-1){
            int xx; scanf("%d",&xx);
            if(xx == 1) mp[i].set(j);
        }
    rep(i,0,m-1) norml[1].set(i);
    for(int i = n-1; i >= 0; i--)
        for(int j = m-1; j >= 0; j--){
            base[i].set(j);
            row.set(0,1);
            test();
            row.set(0,0);
            test();
        }
    printf("NO\n");
    return 0;
}
```

8.3 IDA*

```
#include <cstdio>
#include <iostream>
#include <algorithm>
#include <cmath>

int n, shell[21];

// swap [x1, x2] and [x2 + 1, x3]
void move (int x1, int x2, int x3)
{
     int tmp[21], i, j;
     for (i = x2 + 1, j = 0; i <= x3; i++, j++)
         tmp[j] = shell[i];
     for (i = x1; i <= x2; i++, j++)
         tmp[j] = shell[i];
     for (i = x1, j = 0; i <= x3; i++, j++)
         shell[i] = tmp[j];
     return;
}

int hfunc ()
{
    int i, ans = 0;
    for (i = 0; i < n - 1; i++)
        if (shell[i + 1] != shell[i] + 1) ans++;
    if (shell[n - 1] != n) ans++;
    return ans;
}

int maxdepth;
int dfs (int depth)
{
    int x1, x2, x3, h;
    for (x1 = 0; x1 <= n - 2; x1++) //枚举移动区间的左端点
    {
        for (x2 = x1; x2 <= n - 2; x2++)  //枚举移动区间的右端点
        {
```

```
            for (x3 = x2 + 1; x3 <= n - 1; x3++)  //枚举插入点
            {
                move(x1, x2, x3);   //进行移动
                h = hfunc();
                if (h == 0) return 1;
                else if (3 * depth + h <= 3 * maxdepth) //IDA*, 限制深度
                {
                     if (dfs(depth + 1)) return 1;
                }
                move(x1, x1 - x2 + x3 - 1, x3); //如果不可行, 则返回原状
            }
        }
    } return 0;
}

int main ()
{
    int kase, i;
    scanf("%d", &kase);
    for (; kase > 0; kase--)
    {
        scanf("%d", &n);       //几本书
        for (i = 0; i < n; i++)
            scanf("%d", &shell[i]); //各个位置的序号
        maxdepth = (int)ceil((double)hfunc() / 3);
        if (maxdepth) while (maxdepth < 5 && dfs(1) == 0) maxdepth++;   //不断加大搜索深度
        if (maxdepth == 5) printf("5 or more\n");
        else printf("%d\n", maxdepth);
    }
    return 0;
}

/*
    题意:
1-n本书, 可以选择任意一段连续的书, 再将这一段书插入到其他某个位置。
目标是将书按照1-n的顺序依次排列, 求最少需要多少次操作。
如果更换次数 >= 5, 则直接输出  “5 or more”

    解法:
限制dfs的搜索深度, 并且设计一个估价函数。
可以发现, 将一段书插入到某一个位置, 最多改变3个数的后继值, 因此对于每一个状态, 记录该状态下i与i+1不匹配的个数
然后ceil(这个个数/3)的值就是该状态达到目标状态所至少需要的操作数

对于每个状态, 当前深度+目标操作数 < 5才继续进行深度搜索, 由此大大降低了搜索的复杂度
*/
```

8.4 左右手路径

```
#include <bits/stdc++.h>
#define __ ios::sync_with_stdio(0);cin.tie(0);cout.tie(0)
#define rep(i,a,b) for(int i = a; i <= b; i++)
#define LOG1(x1,x2) cout << x1 << ": " << x2 << endl;
#define LOG2(x1,x2,y1,y2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << endl;
#define LOG3(x1,x2,y1,y2,z1,z2) cout << x1 << ": " << x2 << " , " << y1 << ": " << y2 << " , " << z1 << ": " << z2 << endl;
typedef long long ll;
```

```
typedef double db;
const int N = 1e5+100;
const int M = 1e5+100;
const db EPS = 1e-9;
using namespace std;

//左手路径，优先序循环为顺时针，第一个优先选择为前一个方位
//右手路径，优先序循环为逆时针，第一个优先选择为后一个方位
string choice[4] = {"UP","RIGHT","DOWN","LEFT"},op;
int now = 2;

int main()
{
    //右手路径，初始 now = "DOWN"
    while(1){
        rep(i,0,3){
            string next = choice[(now-i+5)%4];
            cout << "LOOK " << next << endl;
            cin >> op;
            if(op == "SAFE"){
                cout << "GO " << next << endl;
                cin >> op;
                now = (now-i+5)%4;
                if(op == "YES") exit(0);
                break;
            }
        }
    }

    //左手路径，初始 now = "RIGHT"
    // while(1){
    //  rep(i,0,3){
    //      string next = choice[(now+i+3)%4];
    //      cout << "LOOK " << next << endl;
    //      cin >> op;
    //      if(op == "SAFE"){
    //          cout << "GO " << next << endl;
    //          cin >> op;
    //          now = (now+i+3)%4;
    //          if(op == "YES") exit(0);
    //          break;
    //      }
    //  }
    // }
    return 0;
}
```

8.5 圆周率小数点后第 n 位

```
// 求小数点后n位
#include <iostream>
#define MAX_C 56000
int a = 10000, b, c = MAX_C, d, e, f[MAX_C + 1], g, n, ans, cnt;
using namespace std;
int main() {
    while(~scanf("%d", &n)){
    for (; b - c; )
        f[b++] = a / 5;
```

```
    for (; d = 0, g = c * 2;
         c -= 14, ans = e + d / a, e = d % a, cnt++)
    { if (cnt * 4 > n) break;
     for (b = c; d += f[b]*a, f[b] = d % --g, d /= g--, --b; d *= b); }
    if (n % 4 == 0) cout << (ans / 1000);
    else if (n % 4 == 1) cout << ((ans / 100) % 10);
    else if (n % 4 == 2) cout << ((ans / 10) % 10);
    else if (n % 4 == 3) cout << (ans % 10);
    printf("\n");}
    return 0;
}
```

8.6 BM 板子

8.6.1 BM 板子

```
#include <cstdio>
#include <cstring>
#include <cmath>
#include <algorithm>
#include <vector>
#include <string>
#include <map>
#include <set>
#include <cassert>
#include<bits/stdc++.h>
using namespace std;
#define rep(i,a,n) for (ll i=a;i<n;i++)
#define per(i,a,n) for (ll i=n-1;i>=a;i--)
#define pb push_back
#define mp make_pair
#define all(x) (x).begin(),(x).end()
#define fi first
#define se second
#define SZ(x) ((int)(x).size())
typedef long long ll;
typedef vector<ll> VI;

typedef pair<ll,ll> PII;
const ll mod=1000000007;
ll powmod(ll a,ll b) {ll res=1;a%=mod; assert(b>=0); for(;b;b>>=1){if(b&1)res=res*a%mod
    ;a=a*a%mod;}return res;}
// head

ll _,n;
namespace linear_seq {
    const ll N=10010;
    ll res[N],base[N],_c[N],_md[N];
    vector<ll> Md;
    void mul(ll *a,ll *b,ll k) {
        rep(i,0,k+k) _c[i]=0;
        rep(i,0,k) if (a[i]) rep(j,0,k) _c[i+j]=(_c[i+j]+a[i]*b[j])%mod;
        for (ll i=k+k-1;i>=k;i--) if (_c[i])
            rep(j,0,SZ(Md)) _c[i-k+Md[j]]=(_c[i-k+Md[j]]-_c[i]*_md[Md[j]])%mod;
        rep(i,0,k) a[i]=_c[i];
    }
    ll solve(ll n,VI a,VI b) { // a 系数 b 初值 b[n+1]=a[0]*b[n]+...
//        printf("%lld\n",SZ(b));
        ll ans=0,pnt=0;
```

```
        ll k=SZ(a);
        assert(SZ(a)==SZ(b));
        rep(i,0,k) _md[k-1-i]=-a[i];_md[k]=1;
        Md.clear();
        rep(i,0,k) if (_md[i]!=0) Md.push_back(i);
        rep(i,0,k) res[i]=base[i]=0;
        res[0]=1;
        while ((1ll<<pnt)<=n) pnt++;
        for (ll p=pnt;p>=0;p--) {
            mul(res,res,k);
            if ((n>>p)&1) {
                for (ll i=k-1;i>=0;i--) res[i+1]=res[i];res[0]=0;
                rep(j,0,SZ(Md)) res[Md[j]]=(res[Md[j]]-res[k]*_md[Md[j]])%mod;
            }
        }
        rep(i,0,k) ans=(ans+res[i]*b[i])%mod;
        if (ans<0) ans+=mod;
        return ans;
    }
    VI BM(VI s) {
        VI C(1,1),B(1,1);
        ll L=0,m=1,b=1;
        rep(n,0,SZ(s)) {
            ll d=0;
            rep(i,0,L+1) d=(d+(ll)C[i]*s[n-i])%mod;
            if (d==0) ++m;
            else if (2*L<=n) {
                VI T=C;
                ll c=mod-d*powmod(b,mod-2)%mod;
                while (SZ(C)<SZ(B)+m) C.pb(0);
                rep(i,0,SZ(B)) C[i+m]=(C[i+m]+c*B[i])%mod;
                L=n+1-L; B=T; b=d; m=1;
            } else {
                ll c=mod-d*powmod(b,mod-2)%mod;
                while (SZ(C)<SZ(B)+m) C.pb(0);
                rep(i,0,SZ(B)) C[i+m]=(C[i+m]+c*B[i])%mod;
                ++m;
            }
        }
        return C;
    }
    ll gao(VI a,ll n) {
        VI c=BM(a);
        c.erase(c.begin());
        rep(i,0,SZ(c)) c[i]=(mod-c[i])%mod;
        return solve(n,c,VI(a.begin(),a.begin()+SZ(c)));
    }
};
ll dp[10005];
int main() {
    ll t;
    scanf("%lld",&t);
    ll k;
    ll n;
    while (t--) {
        scanf("%lld%lld",&k,&n);
        ll p=powmod(k,mod-2);
        if(n==-1){
            printf("%lld\n",2ll*powmod(k+1,mod-2)%mod);
```

```
            continue;
        }
        vector<ll>v;
        dp[0]=1;
        rep(i,1,11){
            dp[i]=0;
            rep(j,max(0ll,i-k),i){
                dp[i]=(dp[i]+dp[j]*p)%mod;
            }
            v.push_back(dp[i]);
            //cout<<i<<"!!"<<dp[i]<<endl;
        }
        int len=v.size();
        //for(int i=0;i<len;i++)cout<<v[i]<<endl;
        //VI{1,2,4,7,13,24}
        printf("%lld\n",linear_seq::gao(v,n-1));
    }
}
```

8.6.2 BM 递推式

```
#include<bits/stdc++.h>
using namespace std;
#define inf 0x3f3f3f3f
#define rep(i,a,b) for(int i=a;i<=b;i++)
#define per(i,a,b) for(int i=a;i>=b;i--)
#define ll long long
#define VI vector<ll>
const int MAXN = 105;
const int MOD = 1e9+7;
int a[5]={2,3,7};
int b[MAXN];
const int N = 50010;///多项式系数最大值
int64_t res[N],c[N],md[N],COEF[N]/**COEF是多项式系数*/,Mod=10000007;
vector<ll> Md;

inline static int64_t gcdEx(int64_t a, int64_t b, int64_t&x, int64_t& y)
{
    if(!b) {x=1;y=0;return a;}
    int64_t d = gcdEx(b,a%b,y,x);
    y -= (a/b)*x;
    return d;
}

static int64_t Inv(int64_t a, int64_t Mod) {
    int64_t x, y;
    return gcdEx(a, Mod, x, y)==1?(x%Mod+Mod)%Mod:-1;
};

VI BM(VI s) {///BM算法求模数是质数的递推式子的通项公式,可以单独用
        VI C(1,1),B(1,1);
        int L(0),m(1),b(1);
        for(size_t n(0);n<s.size();++n) {
            int64_t d(0);
            for(int i(0);i<=L;++i) d=(d+C[i]*s[n-i])%Mod;
            if (!d) ++m;
            else {
                VI T(C);
```

```
                int64_t c(Mod-d*Inv(b,Mod)%Mod);
                while (C.size()<B.size()+m) C.push_back(0);
                for (size_t i(0);i<B.size();++i)
                    C[i+m]=(C[i+m]+c*B[i])%Mod;
                if (2*L<=(int)n) {L=n+1-L; B=T; b=d; m=1;}
                else ++m ;
            }
        }
        ///下边这样写能够输出递推式的系数.
        printf("F[n] = ") ;
        for(size_t i(0);i<C.size();++i) {
            COEF[i+1] = min((int64_t)C[i],Mod-C[i]) ;
            if(i>0) {
                if(i != 1) printf(" + ") ;
                printf("%lld*F[n-%d]",COEF[i+1],i) ;
                putchar(i+1==C.size()?'\n':' ') ;
            }
        }
        return C;
    }
int main()
{
    VI v = {0,1,2,4,11,25} ;
    BM(v);
    return 0;
}
```

8.7 高精度

8.7.1 高精度

```
#include <cstdio>

struct HighPrec{
    int L , A[10001];
};
inline HighPrec Init(){
    // 函数作用: 返回一个初值为 1 的高精度
    HighPrec H;
    H.L = 1 , H.A[1] = 1;
    return H;
}
HighPrec Mul(HighPrec A , int k){
    // 函数作用: 高精度乘上单精度
    HighPrec H;
    for(int i = 1 ; i <= A.L ; i++)
        H.A[i] = A.A[i] * k; // 乘
    for(int i = 2 ; i <= A.L ; i++)
        H.A[i] += H.A[i - 1] / 10,
        H.A[i - 1] %= 10; // 进位
    H.L = A.L;
    while(H.A[H.L] > 10) // 看原先最高位能不能进位
        H.A[H.L + 1] = H.A[H.L] / 10,
        H.A[H.L] %= 10, // 进位
        H.L++; // 增加长度
    return H;
}
HighPrec Div(HighPrec A , int k){
    // 函数作用: 高精度除以单精度
```

```
    HighPrec H;
    int t = 0;
    for(int i = A.L ; i >= 1 ; i--)
        t = t * 10 + A.A[i], // 小学竖式
        H.A[i] = t / k , t %= k;
    H.L = A.L;
    while(H.A[H.L] == 0)
        H.L--; // 看原先最高位有没有被除为 0 并不断退位
    return H;
}
void OutPut(HighPrec A){
    // 函数作用：输出高精度
    for(int i = A.L ; i >= 1 ; i--)
        printf("%d" , A.A[i]); // 正常套路，逆序输出
}

HighPrec T;
int n;

int main(){
    scanf("%d" , &n);
    T = Init(); // 赋初值 1
    // 下面的乘除是原公式简化的结果，大家请手动去简化
    for(int i = n + 2 ; i <= n * 2 ; i++)
        T = Mul(T , i);
    for(int i = 1 ; i <= n ; i++)
        T = Div(T , i);
    OutPut(T); // 输出
    return 0;
}
```

8.7.2 大数模板

```
#include <bits/stdc++.h>
typedef long long ll;
using namespace std;

struct BigInteger{
    ll A[50];
    enum{MOD = 1000000};
    BigInteger(){memset(A, 0, sizeof(A)); A[0]=1;}
    void set(ll x){memset(A, 0, sizeof(A)); A[0]=1; A[1]=x;}
    void print(){
        printf("%lld", A[A[0]]);
        for (ll i=A[0]-1; i>0; i--){
            if (A[i]==0){printf("000000"); continue;}
            for (ll k=10; k*A[i]<MOD; k*=10ll) printf("0");
            printf("%lld", A[i]);
        }
        printf("\n");
    }
    ll& operator [] (int p) {return A[p];}
    const ll& operator [] (int p) const {return A[p];}
    BigInteger operator + (const BigInteger& B){
        BigInteger C;
        C[0]=max(A[0], B[0]);
        for (ll i=1; i<=C[0]; i++)
            C[i]+=A[i]+B[i], C[i+1]+=C[i]/MOD, C[i]%=MOD;
```

```
        if (C[C[0]+1] > 0) C[0]++;
        return C;
    }
    BigInteger operator * (const BigInteger& B){
        BigInteger C;
        C[0]=A[0]+B[0];
        for (ll i=1; i<=A[0]; i++)
            for (ll j=1; j<=B[0]; j++){
                C[i+j-1]+=A[i]*B[j], C[i+j]+=C[i+j-1]/MOD, C[i+j-1]%=MOD;
            }
        if (C[C[0]] == 0) C[0]--;
        return C;
    }
};

int main(){
    BigInteger A,B,C,D;
    A.set(10000);
    B.set(10000);
    C = A+B;
    D = A*B;
    C.print();
    D.print();
}
```

8.8 二叉树

题意:
给定一个前序遍历、后序遍历，求有多少个符合条件的中序遍历，即有多少个不同的二叉树?

思路:
本题的关键在于识别这样一对 (i, j) 满足，a[i] == a[j] 且 a[i+1] == a[j-1]，则 a[i] 可作为子树根节点，而 a[i+1] 可为左儿子，也可为右儿子，即总方案数 * 2。

```
#include <bits/stdc++.h>
using namespace std;

char a[30], b[30];

int main() {
    int ans = 1;
    scanf("%s%s", a, b);
    for(int i = 0; a[i]; i++)
        for(int j = 1; b[j]; j++)
            if(a[i] == b[j] && a[i+1] == b[j-1])
                ans *= 2;
    printf("%d\n", ans);
    return 0;
}
```

8.9 全排列

```
#include <bits/stdc++.h>
using namespace std;

void Permutation(int step, int len, int a[])
```

```
{
    if(step == (len-1)) //当选取到最后一个元素时，输出当前排列
    {
        for(int i = 0; i < len; i++) cout<<a[i]<<" ";
        cout<<endl;
        return;
    }
    for(int i = step; i < len; i++)
    {
        int tmp; tmp = a[step]; a[step] = a[i]; a[i] = tmp; //交换a[i]和a[step]
        Permutation(step+1, len, a);
        tmp = a[step]; a[step] = a[i]; a[i] = tmp; //交换a[i]和a[step]
    }
}

int a[20];

int main()
{
    for(int i = 0; i < 6; i++) a[i] = i;
    Permutation(0, 6, a);
    return 0;
}
```