【typora常用公式】

加粗　　下标1_1　　上标2^2　　斜体

分式：$\frac{1}{2}$

【算法知识整理】

【前置小知识点】

关系运算符优先级大于符号运算符！

& 按位与操作，按二进制位进行"与"运算。运算规则：（有 **0** 则为 **0**）

```
0&0=0;
0&1=0;
1&0=0;
1&1=1;
```

| (or)按位或运算符，按二进制位进行"或"运算。运算规则：（有 **1** 则为 **1**）

```
0|0=0;
0|1=1;
1|0=1;
1|1=1;
```

XOR，异或运算符。运算规则：（不同为**1**，同为**0**）

```
0^0=0;
0^1=1;
1^0=1;
1^1=0;
```

>>右移运算符　　n/2

<<左移运算符　　n*2

取到某个数的二进制最低位：　　result=n&（-n);

取到某个数的二进制最高位：

```
//关系运算符优先级大于符号运算符！所以必须要有括号
```

```
n|=(n>>1);  //前2位变为1
n|=(n>>2);  //前4位变为1
n|=(n>>4);  //前8位变为1
n|=(n>>8);  //前16位变为1
n|=(n>>16); //前32位变为1
n|=(n>>32); //前64位变为1
//超过int最大位数，足够大，能够保证所有位都变为1.
n^=(n>>1);  //右移一位异或后n取到最高位的大小
//n即为所求
//例:
//1001
// 100  1101
//1101
//  11  1111
//1111
//.....
//1111
// 111  1000
//异或后求出1000
```

保留任意位小数输出

```
#include<iomanip>        //需要头文件
cout<<fixed<<setprecision(n)<<number<<endl;  //n为保留小数位数
```

【赛前输入输出模板】

```
//__int128类型输入输出
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;
}
void print(__int128 x){
    if(x<0)putchar('-'),x=-x;
```

```
    if(x>9)print(x/10);
    putchar(x%10+'0');
}
```

【STL用法整合】

一、【Vector】

数组

```
1.创建vecotr对象：
(1) vector<int> v1;
(2) vector<int> v2(10);
2.基本操作：
v.capacity();  //容器容量
v.size();      //容器大小
v.at(int idx); //用法和[]运算符相同
v.push_back(); //尾部插入
v.pop_back();  //尾部删除
v.front();     //获取头部元素
v.back();      //获取尾部元素
v.begin();     //头元素的迭代器
v.end();       //尾部元素的迭代器
v.insert(pos,elem); //pos是vector的插入元素的位置
v.insert(pos, n, elem) //在位置pos上插入n个元素elem
v.insert(pos, begin, end);
v.erase(pos);   //移除pos位置上的元素，pos为迭代器
v.erase(begin, end); //移除[begin, end)区间的数据，
reverse(pos1, pos2); //将vector中的pos1~pos2的元素逆序存储
```

二、【Queue】

queue为单端队列，deque为双端队列。

```
1. 创建deque对象
```

```
(1) deque<int> d1;
(2) deque<int> d2(10);
2. 基本操作:
(1) 元素访问:
d[i];
d.at[i];
d.front();
d.back();
d.begin();
d.end();
(2) 添加元素:
d.push_back();
d.push_front();
d.insert(pos,elem); //pos是vector的插入元素的位置
d.insert(pos, n, elem) //在位置pos上插入n个元素elem
d.insert(pos, begin, end);
(3) 删除元素:
d.pop_back();
d.pop_front();
d.erase(pos);   //移除pos位置上的元素，返回下一个数据的位置
d.erase(begin, end); //移除[begin, end)区间的数据，返回下一个元素的位置
```

优先队列:

```
priority_queue<int>q;
//默认从大到小，需要重载<运算符
struct node{
    int from,to,val;
  friend bool operator<(node a, node b)  //从小到大
    {
        return a.key > b.key;
    }
};
```

三、【Stack】

栈

```
1. 创建stack对象
(1) stack<int> s1;
2. 基本操作：
empty() 堆栈为空则返回真
pop() 移除栈顶元素
push() 在栈顶增加元素
size() 返回栈中元素数目
top() 返回栈顶元素
```

单调栈：

一种思想，把栈中的数据按照递增或递减的顺序放置

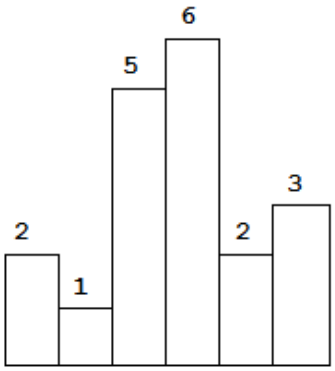
```
//伪代码
stack<int> st;
//此处一般需要给数组最后添加结束标志符，具体下面例题会有详细讲解
for (遍历这个数组)
{
    if (栈空 || 栈顶元素大于等于当前比较元素)
    {
        入栈;
    }
    else
    {
        while (栈不为空 && 栈顶元素小于当前元素)
        {
            栈顶元素出栈;
            更新结果;
        }
        当前数据入栈;
    }
}
```

例题：

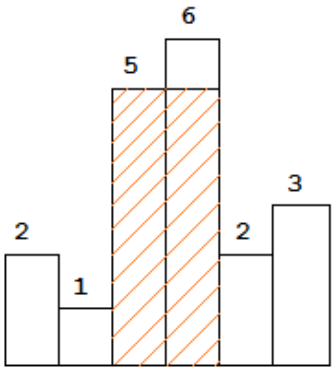
柱状图中的最大矩形

给定 *n* 个非负整数，用来表示柱状图中各个柱子的高度。每个柱子彼此相邻，且宽度为 1 。

求在该柱状图中，能够勾勒出来的矩形的最大面积。



以上是柱状图的示例，其中每个柱子的宽度为 1，给定的高度为 `[2,1,5,6,2,3]`。



图中阴影部分为所能勾勒出的最大矩形面积，其面积为 `10` 个单位。


```
//首先我们结合暴力法来分析一下为什么使用单调栈做本题，我们计算每个柱子出的最大面积的时候是分别找到左边第一个比它小的柱子位置x和右边第一个比它小的柱子y来计算的。当我们弹出一个元素的时候，因为是单调递增栈因此栈顶元素值一定是它左边第一个小于它的元素，而**使它弹出的那个元素一定是它右边第一个小于它的值**，这样我们只需要使用弹出的高度 * 宽度就可计算出当前位置的面积
int largestRectangleArea(vector<int>& heights) {
    heights.push_back(-1);//同理，我们希望栈中所有数据出栈，所以给数组最后添加一个负数
    stack<int> st;
    int ret = 0, top;
    for (int i = 0; i < heights.size(); i++)
    {
        if (st.empty() || heights[st.top()] <= heights[i])
        {
            st.push(i);
        }
        else
        {
            while (!st.empty() && heights[st.top()] > heights[i])
```

```
            {
                top = st.top();
                st.pop();
                //i-top指的是当前矩形的宽度，heights[top]就是当前的高度
                int tmp = (i - top)*heights[top];
                if (tmp > ret)
                    ret = tmp;
            }
            st.push(top);
            heights[top] = heights[i];
        }
    }
    return ret;
}
```

四、【Map】

(1) map为单重映射、multimap为多重映射；

(2) 主要区别是map存储的是无重复键值的元素对，而multimap允许相同的键值重复出现，既一个键值可以对应多个值。

(3) map内部自建了一颗红黑二叉树，可以对数据进行自动排序，所以map里的数据都是有序的，这也是我们通过map简化代码的原因。

(4)自动建立key-value的对应关系，key和value可以是你需要的任何类型。

(5) key和value一一对应的关系可以去重。

注意：map会以key的大小从小到大的顺序自动排序（内部为红黑树，与set相同）

```
1. 创建Map对象
map<T1,T2> m;
map<T1,T2, op> m;  //op为排序规则，默认规则是less<T>,从小到大
2. 基本操作：
m.at(key);
m[key];
m.find(key)          //存在返回迭代器，不存在返回m.end()
m.count(key);        //返回bool，1存在，0不存在
m.max_size();        //求算容器最大存储量
m.size();            //容器的大小
m.begin();
m.end();
```

```
m.insert(elem);
m.insert(pos, elem);
m.insert(begin, end);
```

五、【List】

双向链表

```
1.创建List对象:
list<int> L1;
list<int> L2(10);
3.基本操作:
(1) 元素访问:
lt.front();
lt.back();
lt.begin();
lt.end();
(2) 添加元素:
lt.push_back();
lt.push_front();
lt.insert(pos, elem);
lt.insert(pos, n , elem);
lt.insert(pos, begin, end);
(3) 删除元素:
lt.pop_back();
lt.pop_front();
lt.erase(begin, end);
lt.erase(elem);
```

六、【Set】

1. 特点:

构造set集合的主要目的是为了快速检索,去重与排序

(1) set存储的是一组无重复的元素，而multiset允许存储有重复的元素;0

(2) 如果要修改某一个元素值，必须先删除原有的元素，再插入新的元素。

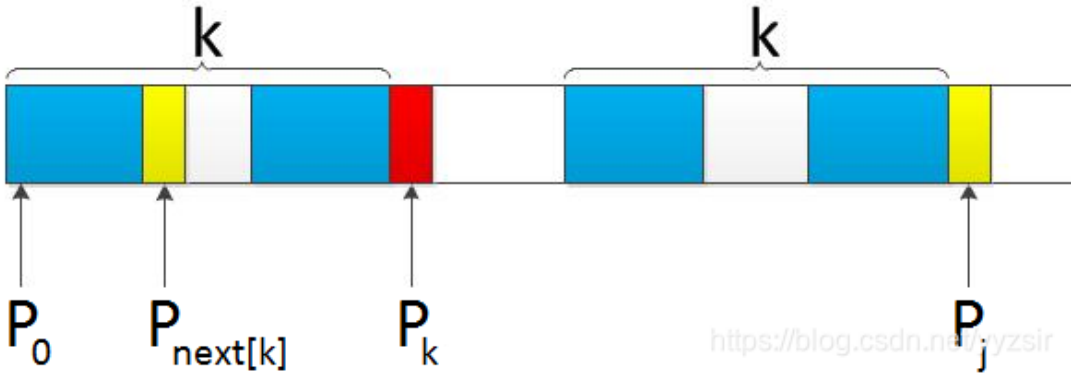
```
1.创建Set对象：
set<T> s;
set<T, op(比较结构体)> s;      //op为排序规则，默认规则是less<T>(升序排列)，或者是greater<T>(降序规则)。
3. 基本操作：
s.size();       //元素的数目
s.max_size();   //可容纳的最大元素的数量
s.empty();      //判断容器是否为空
s.find(elem);   //存在返回迭代器，不存在返回s.end()
s.count(elem); //elem的个数，要么是1，要么是0，multiset可以大于一
s.begin();
s.end();
s.rbegin();      //返向迭代器的首迭代器
s.rend();        //返向迭代器的末尾后一个迭代器==反向后的s.end（）
s.insert(elem);
s.insert(pos, elem);
s.insert(begin, end);
s.erase(pos);
s.erase(begin,end);
s.erase(elem);
s.clear();       //清除a中所有元素；
```

【字符串】

一、【字符串匹配】

1、【KMP算法】

获取next数组时，k=next[k]数组的原理。



模板：

```
//next[j]代表j之前的最长前后缀长度
//AB‘C’DAB‘D’E
//next[6]=2；//D之前的最长前后缀长度为2‘AB’
int Next[1000000];
void getNext(string p)
{
    Next[0] = -1;         //初始化next[0]，相当于把前后缀相同长度的表整体向右移一位。
    int j = 0;
    int k = -1;
    while (j < (int)p.length() - 1)
    {
        if (k == -1 || p[j] == p[k])//没有匹配的或找到相等
        {
            if(p[++j]==p[++k])Next[j]=Next[k];
            else Next[j]=k;
        }
        else
        {
            k = Next[k];//见上图
        }
    }
}
int KMP(string T, string p)//T为母串，p为子串
{
    int i = 0;
    int j = 0;
    getNext(p);
    while (i < (int)T.length() && j < (int)p.length())
    {
        if (j == -1 || T[i] == p[j])
        {
            i++;
            j++;
        }
```

```
        else
        {
            j = Next[j];//相当于右移j-next[j]位
        }
    }
    if (j == (int)p.length())
    {
        return i - j;//返回该子串在母串中的起始位置
    }
    return -1;//不匹配
}
```

2、【FFT】

模板：

```
/*
FFT板子
*/
void fft_01Match(string &s,string &t)
{
    int n=s.size(),m=t.size();
    reverse(t.begin(),t.end());
    vector<int> ans(n+m-1,0);
    vector<int> A(n,0),B(m,0),C;

    //26次FFT，非常有超时风险
    for(char ch='a';ch<='z';ch++)
    {
        for(int i=0;i<n;i++) if(s[i]==ch || s[i]=='*') A[i]=1; else
 A[i]=0;
        for(int i=0;i<m;i++) if(t[i]==ch)              B[i]=1; else
 B[i]=0;
        C = fft::multiply(A,B);
        for(int i=0;i<(int)C.size();i++) ans[i]+=C[i];
    }
    int cnt_tongpei = 0;
    for(int i=0;i<m;i++) if(t[i]=='*') cnt_tongpei++;
    for(int i=m-1;i<n;i++)
    {
        int ct = ans[i] + cnt_tongpei; //对位匹配个数
        int ck = m - ct;               //对位失配个数
        if(ct==m)
            cout<<(i-m+1)+1<<"\n"; //匹配位置
    }
}
```

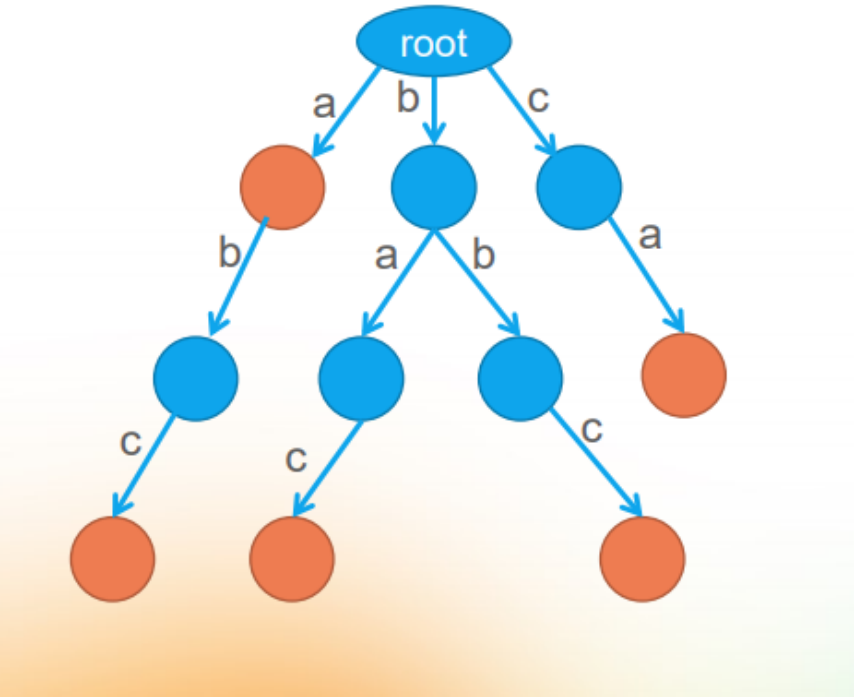
3、【Sunday算法】

模板：

```
#include <iostream>
#include <string>
using namespace std;
const int maxNum = 1005;
int shift[maxNum];
int Sunday(const string& T, const string& P) {
    int n = T.length();
    int m = P.length();

    // 默认值，移动m+1位
    for(int i = 0; i < maxNum; i++) {
        shift[i] = m + 1;
    }
    // 模式串P中每个字母出现的最后的下标
    // 所对应的主串参与匹配的最末位字符的下一位字符移动到该位，所需要的移动位数
    for(int i = 0; i < m; i++) {
        shift[P[i]] = m - i;
    }
    // 模式串开始位置在主串的哪里
    int s = 0;
    // 模式串已经匹配到的位置
    int j;
    while(s <= n - m) {
        j = 0;
        while(T[s + j] == P[j]) {
            j++;
            // 匹配成功
            if(j >= m) {
                return s;
            }
        }
        // 找到主串中当前跟模式串匹配的最末字符的下一个字符
        // 在模式串中出现最后的位置
        // 所需要从(模式串末尾+1)移动到该位置的步数
        s += shift[T[s + m]];
    }
    return -1;
}
```

二、【字典树】



字典树的功能：

1、维护字符串集合（即字典）。

2、向字符串集合中插入字符串（即建树）。

3、查询字符串集合中是否有某个字符串（即查询）。

4、统计字符串在集合中出现的个数（即统计）。

5、将字符串集合按字典序排序（即字典序排序）。

6、求集合内两个字符串的LCP（Longest Common Prefix，最长公共前缀）（即求最长公共前缀）。

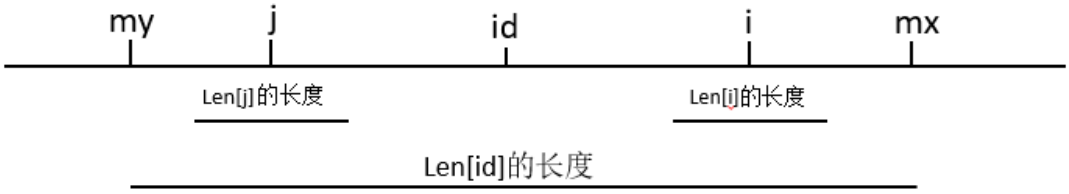
模板：

```
//int trie[MAX_NODE][CHARSET];
//int k;
```

```
//其中MAX_NODE是trie中最大能存储的节点数目，CHARSET是字符集的大小，k是当前trie中包含有多少个节点。Trie[i][j]的值是0表示trie树中i号节点，并没有一条连出去的边，满足边上的字符标识是字符集中第j个字符（从0开始）；trie[i][j]的值是正整数x表示trie树中i号节点，有一条连出去的边，满足边上的字符标识是字符集中第j个字符，并且这条边的终点是x号节点。
//color[p]=1标记p点为终结点，0则非终结点
const int maxn = 5e5 + 10;//最大节点数
int tire[maxn][26], isend[maxn];
int k = 1;
void build(string str)
{
    int p = 0;
    for (int i = 0; i < str.size(); i++)
    {
        int c = str[i] - 'a';
        if (!tire[p][c])tire[p][c] = k++;
        p = tire[p][c];
    }
    isend[p] = 1;
}
bool search(string str)
{
    int p = 0;
    for (int i = 0; i < str.size(); i++)
    {
        int c = str[i] - 'a';
        if (!tire[p][c])return false;
        p = tire[p][c];
    }
    return isend[p];
}
```

三、【最长回文串（Manacher）】

时间：O（n）



原字符串s	a b a b c
预处理后str	% # a # b # a # b # c #
Len[i]	0 1 2 1 4 1 4 1 2 1 2 1

模板：

```
const int maxn=10000010;
char str[maxn];//原字符串
char tmp[maxn<<1];//转换后的字符串
int Len[maxn<<1];//每个点的最长回文子串
//转换原始串
int INIT(char *st)
{
    int i,len=strlen(st);
    tmp[0]='%';//字符串开头增加一个特殊字符，防止越界
    for(i=1;i<=2*len;i+=2)
    {
        tmp[i]='#';
        tmp[i+1]=st[i/2];
    }
    tmp[2*len+1]='#';
    tmp[2*len+2]='$';//字符串结尾加一个字符，防止越界
    tmp[2*len+3]=0;
    return 2*len+1;//返回转换字符串的长度
}
//Manacher算法计算过程
int MANACHER(char *st,int len)
{
     int mx=0,ans=0,po=0;//mx即为当前计算回文串最右边字符的最大值
     for(int i=1;i<=len;i++)
     {
         if(mx>i)Len[i]=min(mx-i,Len[2*po-i]);//在Len[j]和mx-i中取个
小
         else Len[i]=1;//如果i>=mx，要从头开始匹配
         while(st[i-Len[i]]==st[i+Len[i]])Len[i]++;
         if(Len[i]+i>mx)//若新计算的回文串右端点位置大于mx，要更新po和mx的值
         {
```

```
                mx=Len[i]+i;
                po=i;
                ans=max(ans,Len[i]);
            }

        }
        return ans-1;//返回Len[i]中的最大值-1即为原串的最长回文子串额长度
    }
```

四、【AC自动机】

模板：

```
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;
typedef long long ld;
const int mod = 1e9 + 7;
const int maxn = 1e6 + 5;
const int inf = 99999999;
int n;
int tr[maxn][26],tot;
int e[maxn], fail[maxn];
void insert(char* s) {
    int u = 0;
    for (int i = 1; s[i]; ++i) {
        if (!tr[u][s[i] - 'a']) tr[u][s[i] - 'a'] = ++tot;
        u = tr[u][s[i] - 'a'];
    }
    e[u]++;
}
queue<int> q;
void build() {
    for (int i = 0; i < 26; ++i)if (tr[0][i]) q.push(tr[0][i]);
    while (!q.empty()) {
        int u = q.front();q.pop();
        for (int i = 0; i < 26; ++i)
            if (tr[u][i]) fail[tr[u][i]] = tr[fail[u]][i],
q.push(tr[u][i]);
            else tr[u][i] = tr[fail[u]][i];
    }
}
int query(char* t) {
    int u = 0, res = 0;
    for (int i = 1; t[i]; ++i) {
        u = tr[u][t[i] - 'a'];
```

```
        for (int j=u; j && e[j] != -1; j = fail[j]) res += e[j],
e[j] = -1;
    }
    return res;
}
char s[maxn];
int main() {
    scanf("%d", &n);
    for (int i = 1; i <= n; ++i) scanf("%s", s + 1), insert(s);
    scanf("%s", s + 1);
    build();
    printf("%d",query(s));
    return 0;
}
```

五、【字符串hash】

自然溢出方法

Hash公式

unsigned long long Hash[n]

$$hash[i] = hash[i-1] * p + id(s[i])$$

利用unsigned long long的范围自然溢出，相当于自动对$2^{64}-1$取模

给你一个字符串 text ， 请你返回满足下述条件的 不同 非空子字符串的数目：

- 可以写成某个字符串与其自身相连接的形式。

例如，abcabc 就是 abc 和它自身连接形成的。

示例 1：

```
输入：text = "abcabcabc"
输出：3
解释：3 个子字符串分别为 "abcabc" ， "bcabca" 和 "cabcab" 。
```

示例 2：

```
输入：text = "leetcodeleetcode"
输出：2
解释：2 个子字符串为 "ee" 和 "leetcodeleetcode" 。
```

【例题：】

```
#define ull unsigned long long   // 自然溢出用 unsigned long long
```

```cpp
const int MAXN = 2e4 + 50;
class Solution {
public:
    unordered_set<ull> H;
    ull base = 29;
    ull hash[MAXN], p[MAXN];

    int distinctEchoSubstrings(string text) {
        int n = text.size();
        hash[0] = 0, p[0] = 1;
        for(int i = 0;i < n;i++)
            hash[i+1] = hash[i]*base + (text[i] - 'a' + 1);

        for(int i = 1;i < n;i++)
            p[i] = p[i-1]*base;

        for(int len = 2; len <= n; len += 2)
        {
            for(int i = 0;i + len -1 < n;i++)
            {
                int x1 = i, y1 = i + len/2 - 1;
                int x2 = i + len/2, y2 = i + len - 1;
                ull left = hash[y1 + 1] - hash[x1] * p[y1 + 1 -
x1];
                ull right = hash[y2 + 1] - hash[x2] * p[y2 + 1 -
x2];

                if(left == right) H.insert(left);
            }
        }
        return H.size();
    }
};
```

字符串Hash的应用

题型一

描述

问题：给两个字符串S1，S2，求S2是否是S1的子串，并求S2在S1中出现的次数

数据范围：1=<|S1|，|S2|<=10000

解法

求出S1和S2的Hash值，并且n^2的求解出S1所有子串的Hash值，放入map中，查询即可。复杂度$n^2 \log n$

题型二

描述

问题：给N个单词串，和一个文章串，求每个单词串是否是文章串的子串，并求每个单词在文章中出现的次数。

数据范围：文章串长度：[1,10^5]，N个单词串总长：[1,10^6]

解法

设单词串总长为$|S|$，文章串总长为$|A|$。

此题和第一题做法相同。复杂度$|A|^2 \log |A| + |S|$

题型三

描述

问题：给两个字符串S1,S2，求它们的最长公共子串的长度。

数据范围：1=<|S1|，|S2|<=10^5

解法

将S1的每一个子串都hash成一个整数

将S2的每一个子串都hash成一个整数

两堆整数，相同的配对，并且找到所表示的字符串长度最大的即可。

复杂度：$O(|S1|^2 + |S2|^2)$

PS：为觉得开数组不保险，所以上面的题一和题二都用的map存，这里我也不知道能不能实现$O(1)$的存储和查询。

题型四

描述

问题：给一个字符串S，求S的最长回文子串。

比如abcbbabbc的最长回文子串是cbbabbc，bbabb也是回文串，但不是最长的

数据范围： 1 = <|S|<=10^5

解法

先求子串长度位奇数的，再求偶数的。枚举回文子串的中心位置，然后二分子串的长度，直到找到一个该位置的最长回文子串，不断维护长度最大值即可。

复杂度：$O(|S| * \log |S|)$

六、【回文树】

例题：https://ac.nowcoder.com/acm/contest/view-submission?submissionId=53365438

```
#include<bits/stdc++.h>
#define MN 300005
char s[MN];
int now,num[MN];
bool col[MN];
std::vector<int> vis;
struct PAM{
    int lst=1,len[MN],fail[MN],son[MN][26],N=1;
    void init() {
        fail[0]=fail[1]=1;
        len[0]=0;len[1]=-1;
        N=1;
        lst=1;
    }
    void add(int id) {
        int x=lst,c=s[id]-'a';
        for(;s[id]!=s[id-len[x]-1];x=fail[x]) ;
        if(!son[x][c]) {
            len[++N]=len[x]+2;
            int y=fail[x];
            for(;s[id]!=s[id-len[y]-1];y=fail[y]);
            fail[N]=son[y][c];
            son[x][c]=N;
        }
        lst=son[x][c];
        if(!col[lst]) {
            vis.push_back(lst);
            col[lst]=1;
        }
    }
}P;
int T;
int main()
{
    P.init();
    scanf("%d",&T);
    for(now=1;now<=T;now++) {
        scanf("%s",s+1);
        for(int i=1;s[i]!='\0';i++) P.add(i);
        P.lst=1;
```

```
        for(int j=0;j<vis.size();j++) {
            col[vis[j]]=0;
            num[vis[j]]++;
        }
        vis.clear();
    }
    int Ans=0;
    for(int i=2;i<=P.N;i++) if(num[i]==T) Ans++;
    printf("%d\n",Ans);
}
```

```
//方法二：
#include<iostream>
#define maxn 300005
using namespace std;
string s,s1;
int len[maxn],fail[maxn],tr[maxn][30],cnt[maxn],idex=1,cur;
int get_fail(int x,int i){
    while(i-len[x]-1<0||s[i]!=s[i-len[x]-1]){
        x=fail[x];
    }
    return x;
}
void build(int k){
    int llen=s.size();
    len[1]=-1,fail[0]=1,len[0]=0,fail[1]=1;
    for(int i=0;i<llen;i++){
        int u=get_fail(cur,i);
        if(!tr[u][s[i]-'a']){
            fail[++idex]=tr[get_fail(fail[u],i)][s[i]-'a'];
            tr[u][s[i]-'a']=idex;
            len[idex]=len[u]+2;
        }
        cur=tr[u][s[i]-'a'];
        //cur为以u节点该种回文的数量
        if(cnt[cur]==k-1){
            //若为k-1个，算上这一种便能够达到k个，
            cnt[cur]++;
        }
    }
}
int main()
```

```
{
    int k;
    cin >> k;
    for(int i=1;i<=k;i++){
        cin >> s;
        build(i);
    }
    long long ans=0;
    for(int i=idex;i>0;i--){
        if(cnt[i]==k){
            //计算同种回文串出现次数是否为k次。
            ans++;
        }
    }
    cout << ans << endl;
}
```

【「APIO2014」回文串】

```
#include <bits/stdc++.h>
using namespace std;
const int maxn = 300000 + 5;

namespace pam {
int sz, tot, last;
int cnt[maxn], ch[maxn][26], len[maxn], fail[maxn];
char s[maxn];

int node(int l) {  // 建立一个新节点，长度为 l
  sz++;
  memset(ch[sz], 0, sizeof(ch[sz]));
  len[sz] = l;
  fail[sz] = cnt[sz] = 0;
  return sz;
}

void clear() {  // 初始化
  sz = -1;
  last = 0;
  s[tot = 0] = '$';
  node(0);
  node(-1);
  fail[0] = 1;
}

int getfail(int x) {  // 找后缀回文
```

```
  while (s[tot - len[x] - 1] != s[tot]) x = fail[x];
  return x;
}

void insert(char c) {  // 建树
  s[++tot] = c;
  int now = getfail(last);
  if (!ch[now][c - 'a']) {
    int x = node(len[now] + 2);
    fail[x] = ch[getfail(fail[now])][c - 'a'];
    ch[now][c - 'a'] = x;
  }
  last = ch[now][c - 'a'];
  cnt[last]++;
}

long long solve() {
  long long ans = 0;
  for (int i = sz; i >= 0; i--) {
    cnt[fail[i]] += cnt[i];
  }
  for (int i = 1; i <= sz; i++) {  // 更新答案
    ans = max(ans, 1ll * len[i] * cnt[i]);
  }
  return ans;
}
}  // namespace pam

char s[maxn];

int main() {
  pam::clear();
  scanf("%s", s + 1);
  for (int i = 1; s[i]; i++) {
    pam::insert(s[i]);
  }
  printf("%lld\n", pam::solve());
  return 0;
}
```

【动态规划（dp）】

一、【数位dp】

适用范围：求出在给定区间 [A,B] 内，符合条件 f(i) 的数 i 的个数。条件 f(i) 一般与数的大小无关，而与数的组成有关

```
typedef long long ll;
int a[20];
ll dp[20][state];//不同题目状态不同
ll dfs(int pos,/*state变量*/,bool lead/*前导零*/,bool limit/*数位上界变量*/)//不是每个题都要判断前导零
{
    //递归边界，既然是按位枚举，最低位是0，那么pos==-1说明这个数我枚举完了
    if(!pos) return 1;/*这里一般返回1，表示你枚举的这个数是合法的，那么这里就需要你在枚举时必须每一位都要满足题目条件，也就是说当前枚举到pos位，一定要保证前面已经枚举的数位是合法的。不过具体题目不同或者写法不同的话不一定要返回1 */
    //第二个就是记忆化(在此前可能不同题目还能有一些剪枝)
    if(!limit && !lead && dp[pos][state]!=-1) return dp[pos][state];
    int up=limit?a[pos]:9;
    ll ans=0;
    for(int i=0;i<=up;i++)
    {
        if() ...
        else if()...
        ans+=dfs(pos-1,,lead && i==0,limit && i==a[pos])
        /*大概就是说，我当前数位枚举的数是i，然后根据题目的约束条件分类讨论去计算不同情况下的个数，还有要根据state变量来保证i的合法性，比如题目要求数位上不能有62连续出现,那么就是state就是要保存前一位pre,然后分类，前一位如果是6那么这意味就不能是2，这里一定要保存枚举的这个数是合法*/
    }
    if(!limit && !lead) dp[pos][state]=ans;
    /*这里对应上面的记忆化，在一定条件下时记录，保证一致性，当然如果约束条件不需要考虑lead，这里就是lead就完全不用考虑了*/
    return ans;
}
ll solve(ll x)
{
    int pos=0;
    while(x)
    {
        a[++pos]=x%10;//取出低位到高位的所有,a[2]=3表示第二位上的数字为3
        x/=10;
```

```
    }
    return dfs(pos,true,true);
}
int main()
{
    ll le,ri;
    while(~scanf("%lld%lld",&le,&ri))
    {
        printf("%lld\n",solve(ri)-solve(le-1));
    }
}
```

二、【01背包问题】

注意事项：用一维数组去滚动时，第二层（容量）需要逆序遍历，这样才能防止一个物品被放入多次，因为如果顺序取，那么，在更新

```
dp[j]=max(dp[j-w[i]]+val[i],dp[j])
```

之前，dp[j-w[i]]可能已经被更新过了，代表的是i下的最优解，而我们需要的是i-1下的最优解，顺序去取必定会导致重复取物品。

有 N 件物品和一个容量是 V 的背包。每件物品只能使用一次。

第 i 件物品的体积是 v_i，价值是 w_i。

求解将哪些物品装入背包，可使这些物品的总体积不超过背包容量，且总价值最大。
输出最大价值。

模板：

```
//可改为一维数组进行滚动，也要逆序
typedef long long ll;
int cost[101], val[101];
int dp[101][1010];
//dp[i][j],当容量为j时，放入前i个物品价值最大;
//dp[i][j]=max(dp[i-1][j],dp[i-1][j-cost[i]]+val[i]);
int main() {
    ios::sync_with_stdio(false); cin.tie(0); cout.tie(0);
    int t, num;
    cin >> t >> num;
    for (int i = 1; i <= num; i++)
    {
        cin >> cost[i] >> val[i];
    }
```

```
    for (int i = 1; i <= num; i++)
    {
        for (int j = t; j >= 0; j--)
        {
            if (j >= cost[i])
                dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - cost[i]]
+ val[i]);
            else dp[i][j] = dp[i - 1][j];
        }
    }
    cout << dp[num][t] << endl;
    return 0;
}
```

三、【完全背包】

注意事项:在变成一维进行滚动时，第二层循环要顺序，这样才能确保重复取，符合完全背包的定义。

每种可拿任意个，无数量限制。

有 N 种物品和一个容量是 V 的背包，每种物品都有无限件可用。

第 i 种物品的体积是 v_i，价值是 w_i。

求解将哪些物品装入背包，可使这些物品的总体积不超过背包容量，且总价值最大。
输出最大价值。

模板：

```
typedef long long ll;
ll cost[10100], val[10100], dp[10000010];
int main() {
    ios::sync_with_stdio(false); cin.tie(0); cout.tie(0);
    int t, num;
    cin >> t >> num;
    for (int i = 1; i <= num; i++)
    {
        cin >> cost[i] >> val[i];
    }
    for (int i = 1; i <= num; i++)
    {
        for (int j = cost[i]; j <= t; j++)
```

```
        {
            dp[j] = max(dp[j], dp[j - cost[i]] + val[i]);
        }
    }
    cout << dp[t] << endl;
    return 0;
}
```

四、【多重背包】

每种最多拿k个

有 N 种物品和一个容量是 V 的背包。

第 i 种物品最多有 s_i 件，每件体积是 v_i，价值是 w_i。

求解将哪些物品装入背包，可使物品体积总和不超过背包容量，且价值总和最大。
输出最大价值。

输入格式

第一行两个整数，N，V，用空格隔开，分别表示物品种数和背包容积。

接下来有 N 行，每行三个整数 v_i, w_i, s_i，用空格隔开，分别表示第 i 种物品的体积、价值和数量。

输出格式

输出一个整数，表示最大价值。

暴力：

```
#include <iostream>
#include <algorithm>
using namespace std;

const int N = 110;
int v[N], w[N], s[N];
int f[N][N];

int main() {
    int n, m;
    cin >> n >> m;

    for (int i = 1; i <= n; i++) {
        cin >> v[i] >> w[i] >> s[i];
    }

    // 三重循环
    for (int i = 1; i <= n; i++) {
        for (int j = 0; j <= m; j++) {
```

```
            for (int k = 0; k <= s[i] && k * v[i] <= j; k++) {
                f[i][j] = max(f[i][j], f[i - 1][j - k * v[i]] + k *
w[i]);
            }
        }
    }

    cout << f[n][m] << endl;

    return 0;
}
```

二进制优化：

```
#include <iostream>
#include <algorithm>
using namespace std;
const int N = 25000;
int v[N], w[N],f[N];
int main() {
    int n, m;
    cin >> n >> m;

    // 核心是划分s[i],然后对所有划分出来的数用01背包思想
    // 20: 1 2 4 8  5
    int cnt = 0;
    for (int i = 1; i <= n; i ++) {
        int a, b, s;
        cin >> a >> b >> s;

        int k = 1;
        while(k <= s) {
            cnt ++;
            v[cnt] = a * k;
            w[cnt] = b * k;
            s -= k;
            k *= 2;
        }
        if (s > 0) {
            cnt ++;
            v[cnt] = a * s;
            w[cnt] = b * s;
        }
    }
    n = cnt;
```

```
    // 01背包模板
    for (int i = 1; i <= n; i ++) {
        for (int j = m; j >= v[i]; j --) {
            f[j] = max(f[j], f[j - v[i]] + w[i]);
        }
    }

    cout << f[m] << endl;

    return 0;
}
```

五、【LCS最长公共子序列】

注：序列可以不连续

```
string S,T;
int dp[maxn+1][maxn+1];
int LCS(string S,string T)
{
    int n=S.size(),m=T.size();
    for(int i=0;i<n;i++){
        for(int j=0;j<m;j++){
            if(S[i]==T[j])dp[i][j]=dp[i-1][j-1]+1;
            else dp[i][j]=max(dp[i-1][j],dp[i][j-1]);
        }
    }
    return dp[n][m]
}
```

LCS存在相同元素转化为LIS，需要把相同元素的下标逆置，这样在更新时才不会对结果造成影响，才是对的，可以用map+vector存下标

参考博客：https://blog.csdn.net/guogaoan/article/details/38539851

六、【LIS最长上升子序列】

Dilworth定理：序列的不上升子序列最少划分数等于序列的最长上升子序列长度

(下降子序列最小个数等于最长上升子序列的长度)

```
int n;//原序列长度
int a[maxn],dp[maxn];//dp[i]:以i结尾的最长上升子序列
int LIS()
{
   int res=0;
    for(int i=0;i<n;i++)
    {
        dp[i]=1;
        for(int j=0;j<i;j++)
        {
            if(a[j]<a[i])
            {
                dp[i]=max(dp[i],dp[j]+1);
            }
        }
        res=max(res,dp[i]);
    }
    return res;
}
```

七、【状态压缩dp】

https://www.luogu.com.cn/problem/P1896

```
#include <iostream>
#include<bits/stdc++.h>
#define int long long
using namespace std;
const int mod = 998244353;
const int maxn = 2e5 + 10;
const int inf = 1e18 + 7;
int n, m, t, k, r, c,x;
int dp[10][1024][100];
int st[maxn];       //每一种状态
int num[maxn];      //每一种状态下放置的数量
//dp方程
//dp[i][j][k]=cigema(dp[i-1][z][k-num[j]]);
signed main()
{
    ios::sync_with_stdio(false); cin.tie(0), cout.tie(0);
    cin >> n >> k;
    int cnt = 0;
    for (int i = 0; i < (1 << n); i++) {
        if (i & (i << 1))continue;      //筛选状态，不能出现两个1相邻的状
态
        int sum = 0;                    //记录该种状态下1的数量
```

```
        for (int j = 0; j < n; j++) {
            if (i & (1 << j))sum++;
        }
        st[++cnt] = i;
        num[cnt] = sum;
    }
    dp[0][1][0] = 1;
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= cnt; j++) {
            for (int c = 0; c <= k; c++) {
                if (c >= num[j]) {
                    for (int t = 1; t <= cnt; t++) {
                        if (!(st[t] & st[j]) && !(st[j] & (st[t] <<
1)) && !(st[j] & (st[t] >> 1))) {   // 该行不能和上一行冲突，原理和预处理
相似
                            dp[i][j][c] += dp[i - 1][t][c -
num[j]];
                        }
                    }
                }
            }
        }
    }
    int ans = 0;
    for (int i = 1; i <=cnt; i++) {
        ans += dp[n][i][k];
    }
    cout << ans << endl;
    return 0;
}
```

八、【树上背包（树形**dp**）】

https://www.luogu.com.cn/problem/P2015

二叉苹果树

```
#include <bits/stdc++.h>
#define int long long
using namespace std;
const double pi = acos(-1.0);    // 高精度圆周率
const double eps = 1e-8;         // 偏差值，有时用1e-10
const int maxn = 2e3 + 7;
const int mod = 1e9 + 7;
```

```
const int inf = 1e9 + 7;
int t,q, n, m, x,k;
int val[maxn];
int mx;
struct Edge {
    int next,to,val;
}edge[maxn];
int head[maxn];
int cnt;
int dp[maxn][maxn];//i节点，保留j条边，留下的苹果的最大值
void init() {
    cnt = 0;
    memset(head, -1, sizeof(head));
    for (int i = 0; i <= maxn; i++) {
        for (int j = 0; j <= maxn; j++) {
            dp[i][j] = 0;
        }
    }

}
void add(int from,int to,int val) {
    edge[++cnt].to = to;
    edge[cnt].val = val;
    edge[cnt].next = head[from];
    head[from] = cnt;
}
int zs[maxn];
void dfs(int u, int fa) {
    for (int i =head[u]; i != -1; i = edge[i].next) {
        int to = edge[i].to;
        if (to == fa)continue;
        dfs(to, u);
        zs[u] += zs[to] + 1;
        for (int z = min(zs[u], k);z >=0 ; z--) {
            for (int j =0 ; j <=min(zs[to],z-1) ; j++) {
                dp[u][z] = max(dp[u][z], dp[u][z-j-1]+dp[to]
[j]+edge[i].val);
            }
        }
    }
}
signed main() {
    ios::sync_with_stdio(0), cin.tie(0), cout.tie(0);
    cin >> n >> k;
    init();
    int q = n - 1;
    int from, to,val;
```

```
    while (q--) {
        cin >> from >> to>>val;
        add(from, to,val);
        add(to, from,val);
    }
    dfs(1, 0);
    cout << dp[1][k] << endl;
    return 0;
}
```

九、【最长下降子序列】

注：dp中得到的数不一定是真正的下降子序列，只有数量是对的，因为每一次都是替换，如果要找真正的最长下降子序列，需要保存特判每一次更新末尾的时候的值

```
int main() {
    int n;
    cin >> n;
    vector<int> v(n), dp;
    for (int i = 0; i < n; i++) {
        cin >> v[i];
    }
    dp.push_back(v[0]);
    for (int i = 1; i < n; i++) {
        if (v[i] < dp.back()) dp.push_back(v[i]);
        else {
            int l = 0, r = dp.size()-1;
            while (l < r) {
                int m = l + (r - l) / 2;
                if (v[i] < dp[m])l = m + 1;
                else r = m;
            }
            dp[l] = v[i];//二分找到大于等于v[i]的并替换
            //只有替换的是最末尾的值，才代表当前dp代表的是真正的数值。
            //if(l==dp.size()-1){
                //保存
            //}
        }
    }
    //最长下降子序列的长度
    cout << dp.size() << endl;
}
```

【数论】

一、【基本的取模运算】

(a+b)%c=(a%c+b%c)%c

(a-b)%c=(a-b+c)%c

(a*b)%c=(a%c)*(b%c)%c

(a/b)%c=(a*b-1)%C

二、【GCD(a,b)、LCM(a,b)】

```
int GCD(int a,int b)
{
    if(b==0)return a;
    return GCD(b,a%b);
}
int LCM(int a,int b)
{
    int t=GCD(a,b);
    return a*b/t;
}
```

三、【扩展欧几里得算法】

适用范围：求**ax+by=k**，这一类的二元一次方程组的解时用到，可以求出这个方程的一组解**x**，**y**。

当要求ax+by=1时，只有当GCD（a,b）=1时才有解若！=1，则该方程无解。

易可知：

$x=x_0+k*b$

$y=y_0-k*a$

我们求出的一组解**x,y**为一组随机解而不是最小解，若想要得到最小解则只需将其对**b**和**a**取模。

$x_0=x\%b$　　$y_0=y\%a$

当我们运行到x=1,y=0时，因为a=GCD（a,b）,显然可得a*x+b*y=GCD(a,b)若想要得到一个其他的解，则假设已经解出了下一个解x_1,y_1。

则有：

$b*x_1+(a\%b)*y_1=GCD(a,b)$

因为：a%b = a - (a/b)*b

则:$b*x_1+(a - (a/b)*b)*y_1=GCD(a,b)$

与之前a*x+b*y=GCD(a,b)相比较

可得：$x_1=y$

$y_1=x-a/b*y$

即可求出相邻两步过程中解的联系，通过这一方式，每一步递归都可以求出一个正确的解，最后求出的一个解即为方程组的某一组特解。

模板：

```
int extgcd(int a,int b,int &x,int &y)
{
    if(b==0)
    {
        x=1;
        y=0;
        return a;
    }
    int gcd=exgcd(b,a%b,x,y);
    int temp=x;
    x=y;
    y=temp-a/b*y;
    return gcd;
}
```

例题：乘法逆元：

乘法逆元定义：**(a*x)%c=1%c**

简写为：　　a*x=1 (mod c)

salution 1：扩展欧几里得算法实现（条件弱，无限制）

可将次方程转化为： a*x+c*k=1

再明显一点：　　a*x+c*y=1

一般题目需要求出x的最小正整数值，则只需在扩展欧几里算法解出一组特解后对x对c取余，原因上文有讲。

但若c，x为负数，则应该将c取绝对值后让x%c（尽可能减去or加上c，使得x尽可能的小），若x<0，则还需将要将x+c以得到最小的正整数解。

当要求**a*x=p (mod c)**时，同理：

```
LL cal(LL a,LL b,LL p)
{
    LL x,y;
    LL gcd=extgcd(a,b,x,y);
    if(p%gcd!=0) return -1;  //gcd(a,b)=1时才有解
    x*=p/gcd;    //x是a*x+c*y=1的解，若要得到a*x+c*y=p的解则只需
                 //两边同时乘以p/gcd
    b/=gcd;
    if(b<0) b=-b;
    LL ans=x%b;
    if(ans<=0) ans+=b;
    return ans;
}
```

salution 2:费马小定理实现(条件强，要求c为质数)

-> a*x=1 (mod c)

-> $a*a^{c-2}=1$ (mod c)

则a^{c-2}即为所要求的的x

详细证明：

使用快速幂可快速得到解。

四、【快速幂、快速乘】

快速幂：

```
int ksm(int base, int power) {
    int result = 1;
    while (power > 0) {
        if (power & 1) {//此处等价于if(power%2==1)
            result = result * base % mod;
        }
        power >>= 1;//此处等价于power=power/2
        base = (base * base) % mod;
    }
    return result;
}
```

快速乘：

```
ll ksc(ll x,ll y,ll mod)
{
    return (x*y-(ll)((long double)x/mod*y)*mod+mod)%mod;
}
```

五、【欧拉筛】

```
int prime[maxn];//记录所有素数的数组 prime[0]存放范围内素数数量
bool visit[maxn];//记录所有数，并将其标记合数为1,素数为0
void Prime(){
    memset(visit,0,sizeof(visit));//初始化全为素数
    memset(prime, 0,sizeof(prime));
    for (int i = 2;i <= maxn; i++) {
        if (!visit[i]) prime[++prime[0]] = i;
        for (int j = 1; j <=prime[0] && i*prime[j] <= maxn; j++) {
            visit[i*prime[j]] = 1;//标记合数为1
            if (i % prime[j] == 0)break;
        }
    }
}
```

六、【分解质因数】

唯一分解定理：

对于任何一个大于1的正整数,都存在一个标准的分解式: N=p1^a1 * p2$^{a2*\cdots*pn}$an;（其中一系列an为指数，pn为质数）

此定理表明：任何一个大于 1 的正整数都可以表示为素数的积。

```
#include<iostream>
#include<math.h>
using namespace std;
int main()
{
 int num;
 cin >> num;
 cout << num << "=";
 while (num != 1)
 {
  int i;
  for (i = 2; i <= num; i++)
  {
   if (num%i == 0)//找到质因数，输出并更新该数寻找下一个质因数
   {
    cout << i;
    if (i < num)//当前质因数小于该数则不是最后一个质因数，输出乘号
     cout << "*";
    num = num / i;
    break;
   }
  }
 }
}
```

七、【数论分块】

对于计算带有下列特征的结果可采取数论分块的思想：

$$\sum_{i=1}^{n}\frac{n}{i}$$

例如n=100的情况下时，当i=51,52.....100时，n/i的结果均为1，因此，在n/i的值相同时，不必依次遍历获值，而只需设置一个left和right区间去乘上该段的贡献（n/i）。

```
int sum=0;
for(int l=1,r;l<=n;l=r+1)
{
    r=n/n/l;
    sum+=(n/l)*(r-l+1);
}
```

八、【中国剩余定理】

在《孙子算经》中有这样一个问题：“今有物不知其数，三三数之剩二（除以3余2），五五数之剩三（除以5余3），七七数之剩二（除以7余2），问物几何？

$$x \equiv 2 \pmod 3 \quad x \equiv 3 \pmod 5 \quad x \equiv 2 \pmod 7$$

具体解法分三步：

> *1*、找出三个数：从*3*和*5*的公倍数中找出被*7*除余*1*的最小数*15*，从*3*和*7*的公倍数中找出被*5*除余*1* 的最小数*21*，最后从*5*和*7*的公倍数中找出除*3*余*1*的最小数*70*。
>
> *2*、用*15*乘以*2*（*2*为最终结果除以*7*的余数），用*21*乘以*3*（*3*为最终结果除以*5*的余数），同理，用*70*乘以*2*（*2*为最终结果除以*3*的余数），然后把三个乘积相加*15*∗*2*+*21*∗*3*+*70*∗*2*得到和*233*。
>
> *3*、用*233*除以*3*，*5*，*7*三个数的最小公倍数*105*，得到余数*23*，即*233*%*105*=*23*
>
> 这个余数*23*就是符合条件的最小数。

核心公式：

$$(a + k * b)\%k = a\%k$$

九、【高维前缀和（1、2、3....维）】

一维：

$$DP[i] = DP[i-1] + num[i]$$

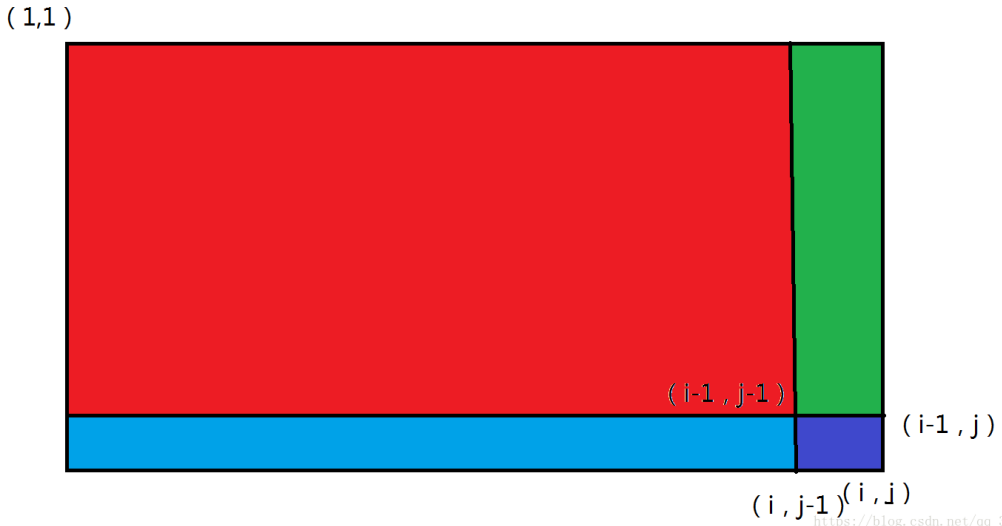
得到n到m区间的和：

$$sum = DP[m] - DP[n]$$

二维：

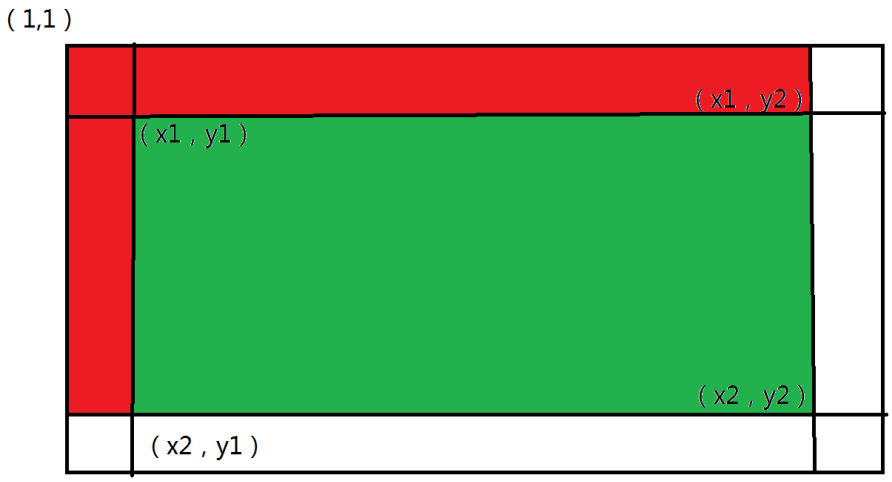
方法一：利用容斥求

注意：预处理边界为0。



$$DP[i][j] = DP[i][j] + DP[i][j] - DP[i-1][j-1] + num[i][j]$$

得到(x_1,y_1),(x_2y_2)坐标间矩阵的和：



$$sum = DP[x_2][y_2] - DP[x_2][y_1] - DP[x_1][y_2] + DP[x_1][y_1]$$

方法二：一维一维求。

```
//假设为n*m的矩阵
for(int i=1;i<=n;i++)
    for(int j=1;j<=m;j++)
        DP[i][j]+=DP[i-1][j];
for(int i=1;i<=n;i++)
    for(int j=1;j<=m;j++)
        DP[i][j]+=DP[i][j-1];
```

三维：

同二维矩阵，一维一维求

```
//假设为大小为x*y*z
for(int i = 1; i <= x; i++)
    for(int j = 1; j <= y; j++)
        for(int k = 1; k <= z; k++)
            a[i][j][k] += a[i - 1][j][k];
for(int i = 1; i <= x; i++)
    for(int j = 1; j <= y; j++)
        for(int k = 1; k <= z; k++)
            a[i][j][k] += a[i][j - 1][k];
for(int i = 1; i <= x; i++)
    for(int j = 1; j <= y; j++)
        for(int k = 1; k <= z; k++)
            a[i][j][k] += a[i][j][k - 1];
```

更高维同理.....

十、【大数乘法】

10.1【求A^BmodC 】

A^B%C = A^(B%phi(C)+phi(C))%C

$$A^x = A^{(x \bmod \mathrm{Phi}(C) + \mathrm{Phi}(C))} \bmod C \ (x \geq \mathrm{Phi}(C))$$

```
#include<iostream>
#include<stdio.h>
#include<cmath>
#include<algorithm>
```

```
#include<queue>
#include<map>
#include<set>
#include<stack>
#include<vector>
#define put putchar('\n')
#define re register
using namespace std;
typedef long long ll;
const int maxn =207;
const int mod = 998244353;
const int inf = 1e9 + 7;
inline int read() {
    char c = getchar(); int tot = 1; while ((c < '0' || c>'9') && c
!= '-') c = getchar(); if (c == '-') { tot = -1; c = getchar(); }
    int sum = 0; while (c >= '0' && c <= '9') { sum = sum * 10 + c
- '0'; c = getchar(); }return sum * tot;
}
inline void wr(int x) { if (x < 0) { putchar('-'); wr(-x); return;
}if (x >= 10)wr(x / 10); putchar(x % 10 + '0'); }
inline void wrn(int x) { wr(x); put; }
inline void wri(int x) { wr(x); putchar(' '); }
int arr[maxn];
int te[maxn];
ll   phi(ll n) {
    ll s = n;
    for (ll i = 2; i * i <= n; i++) {
        if (n % i == 0)  s = s / i * (i - 1);
        while (n % i == 0) n /= i;
    }
    if (n != 1)  s = s / n * (n - 1);
    return s;
}

ll  Pow(ll x, ll y, ll m) {
    ll s = 1;
    for (; y; y >>= 1) {
        if (y & 1) { s *= x; s %= m; }
        x *= x; x %= m;
    }
    return s;
}

ll   Pow(ll x, char* y, ll m) {
    ll phim = phi(m);
    ll s = 0;
    for (int i = 0; y[i] != '\0'; i++) {
```

```
        s = s * 10 + y[i] - '0';
        if (s >= m) break;
    }
    if (s >= m) {
        s = 0;
        for (int i = 0; y[i] != '\0'; i++) {
            s = s * 10 + y[i] - '0';
            if (s >= phim) s %= phim;
        }
        s += phim;
        return Pow(x, s, m);
    }
    else return Pow(x, s, m);
}

char y[1000008];

int main()
{
    ios::sync_with_stdio(false); cin.tie(0); cout.tie(0);
    int t;
    cin >> t;
    while (t--) {
        ll x, m;
        cin >> x >> y >> m;
        cout << Pow(x, y, m) << endl;

    }
    return 0;
}
```

10.2【乘法逆元的应用】

关键：将所有除法变成乘法，**a/b=a***（**b**的逆元）

```
#include<iostream>
#include<stdio.h>
#include<cmath>
#include<algorithm>
#include<queue>
```

```cpp
#include<map>
#include<set>
#include<stack>
#include<vector>
#define put putchar('\n')
#define re register
using namespace std;
typedef long long ll;
const int maxn =1e6;
const int mod = 1e9+7;
const int inf = 1e9 + 7;
inline int read() {
    char c = getchar(); int tot = 1; while ((c < '0' || c>'9') && c
!= '-') c = getchar(); if (c == '-') { tot = -1; c = getchar(); }
    int sum = 0; while (c >= '0' && c <= '9') { sum = sum * 10 + c
- '0'; c = getchar(); }return sum * tot;
}
inline void wr(int x) { if (x < 0) { putchar('-'); wr(-x); return;
}if (x >= 10)wr(x / 10); putchar(x % 10 + '0'); }
inline void wrn(int x) { wr(x); put; }
inline void wri(int x) { wr(x); putchar(' '); }
int arr[maxn];
ll ksm(ll base, ll power) {
    ll result = 1;
    while (power > 0) {
        if (power & 1) {//此处等价于if(power%2==1)
            result = result * base % mod;
        }
        power >>= 1;//此处等价于power=power/2
        base = (base * base) % mod;
    }
    return result;
}
int main()
{
    ios::sync_with_stdio(false); cin.tie(0); cout.tie(0);
    ll t;
    ll n, m, k, q;
    arr[0] = 1;
    for (ll i = 1; i <= 1e6 + 7; i++)arr[i] = (arr[i-1] * i) % mod;
    cin >> t;
    while (t--) {
        cin >> n >> m >> k >> q;
        if (k > n)cout << 0 << endl;
        else {
            //除某个数等于乘以该数的逆元
            ll ans = (arr[n]%mod *ksm(arr[n-k],mod-2)%mod)%mod;
```

```
            ll ni = (arr[m + n]%mod* ksm(arr[n + m - k],mod-
2)%mod)%mod;
            ni = ksm(ni, mod - 2) % mod;
            ll temp = (ni % mod * ans % mod) % mod;
            /*cout << ans << endl;
            cout << ni << endl;*/
            temp = ksm(temp, q);
            cout << temp << endl;
        }
    }
    return 0;
}
```

十一、【组合数学】

```
#include <bits/stdc++.h>
using namespace std;
#define int long long
const int maxn=3e6+10;
const int mod=1e9+7;
int inv[maxn],fac[maxn];
int C(int n,int m){return fac[n]*inv[n-m]%mod*inv[m]%mod;}
int A(int n,int m){return fac[n]*inv[n-m]%mod;}
int ksm(int x,int k){
    int res=1;
    while(k){
        if(k&1)res=res*x%mod;
        x=x*x%mod;
        k/=2;
    }
    return res;
}
int ny(int x){
    return ksm(x,mod-2);
}
void init(){
    inv[0]=fac[0]=1;
    inv[1]=1;
    for(int i=1;i<maxn;i++){
        fac[i]=fac[i-1]*i%mod;
    }
    //求每个数的逆元推导公式：i逆元≡-⌊p/i⌋*(p mod i)的逆元(mod p)
    inv[1]=1;
    for(int i=2;i<maxn;i++){
        inv[i]=(int)(mod-mod/i)*inv[mod%i]%mod;
```

```
    }
    //求阶乘的逆元
    inv[0]=1;
    for(int i=1;i<maxn;i++){
        inv[i]=inv[i-1]*inv[i]%mod;
    }
}
signed main()
{
    init();
    int n,k;
    cin>>n>>k;
    int ans=0;
    for(int i=0;i<=k;i++){
        ans=(ans+C(i+n-1,n-1)*i%mod)%mod;
    }
    cout<<ans<<"\n";
    return 0;
}
```

十二、【容斥】

例题：Shortest Path in GCD Graph

Problem Description

There is an edge-weighted complete graph K_n with n vertices, where vertices are labeled through $1, 2, \ldots, n$. For each $1 \le i < j \le n$, the weight of the edge (i, j) between i and j is $gcd(i, j)$, the greatest common divisor of i and j.

You need to answer q queries. In each query, given two vertices u, v, you need to answer the **length of the shortest path** as well as the **number of shortest paths** between u, v. Since the **number of shortest paths** may be too large, you only need to output it modulo 998244353.

Input

The first line contains two integers $n, q (2 \le n \le 10^7, 1 \le q \le 50000)$, denoting the number vertices in the graph and the number of queries, respectively.

Then q lines follow, where each line contains two integers $u, v (1 \le u, v \le n, u \ne v)$, denoting a query between u and v.

Output

For each query, output one line contains two integers, denote the length and number of shortest path between given nodes, respectively. Note that only the **number of shortest paths** should be taken modulo 998244353.

Sample Input

```
6 2
4 5
3 6
```

Sample Output

```
1 1
2 2
```

HUD-7240

注：该题卡常，必须用素数筛优化分解质因数并且不能使用二进制的容斥，只能用dfs版的方式求容斥（直接裂开）

```
#include <bits/stdc++.h>
#define int long long
using namespace std;
const int maxn = 1e7 + 7;
const int mod = 998244353;
const int inf = 1e18 + 9;
int t, n, m, x,k,q,a,b,cnt;
int read() {
    int x = 0, f = 1;
    char c = getchar();
    while (c < '0' || c>'9') { if (c == '-') f = -1; c = getchar();
}
    while (c >= '0' && c <= '9') x = x * 10 + c - '0', c =
getchar();
    return x * f;
}
int GCD(int a, int b)
{
    if (b == 0)return a;
    return GCD(b, a % b);
}
//素数筛优化分解质因数
int prime[maxn];//记录所有素数的数组 prime[0]存放范围内素数数量
bool vis[maxn];//记录所有数，并将其标记合数为1,素数为0
void Prime(int n) {
    memset(vis, 0, sizeof(vis));//初始化全为素数
    memset(prime, 0, sizeof(prime));
    for (int i = 2; i <= n; i++) {
        if (!vis[i]) prime[++prime[0]] = i;
        for (int j = 1; j <= prime[0] && i * prime[j] <= n; j++) {
            vis[i * prime[j]] = 1;//标记合数为1
            if (i % prime[j] == 0)break;
        }
    }
}
int ans;
vector<int>p;
set<int>se;//用来筛去相同的质因数
//分解质因数，素数筛优化
void Divide(int m)
{
    if (!vis[m]) {
```

```
        se.insert(m);
        return;
    }
    for (int i = 1; i <=prime[0] && m>1; i++){
        if (m % prime[i] == 0){
            se.insert(prime[i]);
            while (m % prime[i] == 0) m /= prime[i];
            if (m > 1 && !vis[m]){
                se.insert(m);return;
            }
        }
    }
}
//dfs版跑容斥
//1-n中与a，b互斥的元素个数
void dfs(int k, int l, int s, int a)
{
    if (k == p.size()) {
        if (l & 1) ans -= a / s;
        else ans += a / s;
        return;
    }
    dfs(k + 1, l, s, a);
    dfs(k + 1, l + 1, s * p[k], a);
    return;
}
void work(int n, int a,int b)
{
    ans = 0;
    p.clear();se.clear();
    Divide(a);Divide(b);
    for (auto x : se)p.push_back(x);
    dfs(0, 0, 1, n);
}
signed main()
{

    n = read(); q = read();
    Prime(n);
    for (int i = 0; i < q; i++) {
        a = read(); b = read();
        int temp = GCD(a, b);
        if (temp==1)cout << 1 << " " << 1 << "\n";
        else {
            cout << 2 << " ";
            work(n,a,b);
            cout << (ans + (temp == 2 ? 1 : 0)) % mod << "\n";
```

```
        }
    }
    return 0;
}
```

【图论】

一、【邻接表的建立】

1、首推：

【前向星式邻接表】

使用最为广泛的一种方法，内存与时间开销相对平衡，性能均衡，无明显的优缺点。

模板：

```
const int MAX = 1e6;     //数组大小初始化
struct Edges {
    int to, w, next; //（边的终点编号、权重、同起点的下一条边）
};
Edges edge[MAX];  //记录所有点的信息
ll head[MAX];     //例：head[1]=5;记录从1点出发的最后输入的一条边5
ll cnt;           //记录每组数据的编号
void init()       //初始化
{
    memset(head, -1, sizeof(head));
    cnt = 0;
}
void add(ll from, ll to, ll w) //添加边
{
    edge[cnt].to = to;
    edge[cnt].w = w;
    edge[cnt].next = head[from];
    head[from] = cnt++;
}
void visit(int from)
{
```

```
    for(int i=head[from];i!=-1;i=edge[i].next)
    {
        //遍历以from为起点的所有边
        //cout<<edge[i].to<<endl;
        //cout<<edge[i].w<<endl;
    }
}
```

2、vector：这是最容易理解的一种方法，使用比较广泛。优点：方便，代码简洁，容易理解。缺点：某些不怀好意的题可能会卡你时间，vector效率不高。

```
/*优点：方便，代码简洁
缺点：有时候有点慢*/
struct node
{
    int to, w;
};
vector<node> v[N]; //定义一个邻接表

void add(int from, int to, int w)
{
    node now;
    now.to = to;
    now.w = w;
    v[from].push_back(now); //建立一条from与to连接的边
    //如果是无向图，则再建立一条to-from的边
    //vec[to].push_back(from);
}

void visit(int i) //遍历与i相连的边
{
    int len = v[i].size();    //判断i点与多少个点相连
    for (int k = 0; k < len; k++) //tmp为与i相连接的点
    {
        node tmp = v[i][k];
    }
}
```

3、链表实现

```
/*优点：速度较快
缺点：内存开销大一些*/
struct Edge
{
    int to;
```

```
    int value;
    Edge *next;
};
Edge *edges[MAXN], pool[MAXM], *alloc;
Edge *new_Edge(int from, int to, int value) //建一条新边
{
    Edge *tmp = alloc++;
    tmp->to = to;
    tmp->value = value;
    tmp->next = edges[from];
    return tmp;
}
void add_edge(int from, int to, int value)
{
    edges[from] = new_Edge(from, to, value);
    //无向图建双向边，有向图不需要
    //edges[to] = new_Edge(to, from, value);
}
void init() //初始化
{
    alloc = pool;
    memset(edges, 0, sizeof edges);
}
```

二、【最短路问题】

适用范围：图上某起点到终点的最少花费（权重）。

1、SPFA算法

适用范围：单源最短路问题

优点：

1、可以判断给定的图是否存在负权边（负环）

2、平均时间复杂度低

缺点：

1、期望时间复杂度低，最优情况下为O(ke),但最差情况下为O（V*E）（V表示图中顶点,E表示图中的边）相当于对所有的顶点以及与之相连的所有边都遍历了一遍，可能会被卡时间，需要加上桶优化。

实现方法：

1、首先创建一个队列和dis数组（d[i]表示从起点到达i点的最短路径，初始化为无穷大），将初始起点的dis置为0后放入队列中。

2、取出起点，将与起点相连的所有边依次遍历，进行松弛操作（其实就是找到最短的路），若松弛成功但该点已在队列内，则跳过，否则将点放入队列中，并标记为在队列内。

3、循环往复直到队列为空

判断负环原理：

由于当不存在负环时，每个子节点最多被更新V（V个顶点）次，因此，当出现负环时只要判断更新次数是否大于V，若大于，则存在负环，返回相应的标记。

伪代码：
用dis数组记录源点到有向图上任意一点距离，其中源点到自身距离为0，到其他点距离为INF。将源点入队，并重复以下步骤：

1、队首x出队
2、遍历所有以队首为起点的有向边(x,i)，若dis[x]+w(x,i)<dis[i]，则更新dis[i]
3、如果点i不在队列中，则i入队
4、若队列为空，跳出循环；否则执行1

模板：

```
bool vis[]                                  //标记是否在队列内
int dis[]                                   //起点到各点的最短路
int c[]                                     //判断各个点更新次数，用来判断是否存在负环，当c[i]>n时存在
bool flag=0;                                //标记是否存在负环，0为不存在
void spfa(int start, int n)                  //start是源点,n为顶点数，用于判断负环
{
    memset(vis, false, sizeof(vis));
    memset(dis, inf, sizeof(dis));  //距离源点的距离初始化成inf(很大的数)
    queue<int>q;
    dis[start] = 0;                         //源点的距离默认成0
    vis[start] = true;                      //标记源点
    q.push(start);                          //放入队列
```

```
    while (!q.empty())
    {
        int u = q.front();                  //取出下一个节点
        q.pop();
        vis[u] = false;                     //标记为不在队列内
        for (int i = head[u]; i != -1; i = edge[i].next)
        {
            int v = edge[i].v;          //与u相连的v节点
            int w = edge[i].w;          //u到v节点的权重
            if (dis[v] > dis[u] + w)  //dis[v]为当前与u点相连的节点的最短路径（初始为INF）
            {
                dis[v] = dis[u] + w;  //如果子节点距离源点的距离可以更小，则更新子节点的距离源点的距离
                //c[v]++;
               // if(c[v]>n)flag=1 return;//存在负环则返回，flag标记为1
                if (!vis[v])            //如果改点不在队列内，则标记并加入队列
                {
                    vis[v] = true;
                    q.push(v);
                }
            }
        }
    }
}
```

2、Dijkstra算法

优化前：O（n^2）

缺点：无法处理带负权的图

核心：贪心

Dijkstra的大致思想就是:根据初始点，挨个的把离初始点最近的点一个一个找到并加入集合，集合中所有的点的d[i]都是该点到初始点最短路径长度，由于后加入的点是根据集合S中的点为基础拓展的，所以也能找到最短路径。

```
伪代码
清除所有点的标号;
    设d[0]=0，其他d[i]=INF;//INF是一个很大的值，用来替代正无穷
    循环n次 {
        在所有未标号结点中，选出d值最小的结点x;
        给结点x标记;
        对于从x出发的所有边(x,y)，更新d[y] = min{d[y], d[x]+w(x,y)}
```

模板：

优先队列优化

```
#include<iostream>
#include<cstdio>
#include<cstdlib>
#include<algorithm>
#include<cmath>
#include<string>
#include<stack>
#include<queue>
#include<cstring>
#include<map>
#include<iterator>
#include<list>
#include<set>
#include<functional>
#include<memory.h>

using namespace std;

const int maxn=105;//顶点最大数
const int M=1e4;//边数最大值
const int inf=0x3f3f3f3f;//无穷大
typedef struct Edge{
    int next;//起点相同的下一条边的序号。
    int to;//该边的终点。
    int w;//该边的权值。
}Edge;
Edge edge[M];//边数组。
int head[maxn];//联系边之间的关系数组，head[i]表示起点为i的第一条边的序号
bool visited[maxn];//若为true，则代表已经确定最短路径
typedef pair<int,int> pll;//first代表最短路径，second代表该最短路径的起点。
int dis[maxn];//存储最短路径。
void init(){
    memset(visited,false,sizeof(visited));
```

```
    memset(dis,inf,sizeof(dis));
    memset(head,-1,sizeof(head));//用-1表示第一条边不存在，即该顶点没边。
}
int n,m;//实际顶点数和边数。
int cnt;//递增序号，为边赋值。
void add(int u,int v,int w){
    //加边操作。
    edge[cnt].to=v;
    edge[cnt].w=w;
    edge[cnt].next=head[u];
    head[u]=cnt++;
}
void dijkstra(int S){
    priority_queue<pll,vector<pll>,greater<pll> > q;//优先队列。
    dis[S]=0;
    q.push(pll(0,S));//左边最短路径，右边顶点序号
    pll temp;
    cnt=0;
    while(!q.empty()){
        temp=q.top();
        q.pop();
        int u=temp.second;
        if(visited[u])continue;//已经确定了就跳过
        visited[u]=true;
        int t,len;
        for(int i=head[u];i!=-1;i=edge[i].next){
            t=edge[i].to;len=edge[i].w;//记录边终点和权值。
            if(!visited[t]&&dis[t]>dis[u]+len){
                dis[t]=dis[u]+len;
                q.push(pll(dis[t],t));
            }
        }
    }
}
int main(){
    while(cin>>n>>m){
        int u,v,w;
        init();
        for(int i=0;i<m;i++){
            cin>>u>>v>>w;//如果m太大了，一定要使用scanf输入，不然可能会超
时。
            add(u,v,w);
            add(v,u,w);
        }
        int S,E;//起点和终点。
        cin>>S>>E;
        dijkstra(S);
```

```
        cout<<dis[E]<<endl;
    }
    return 0;
}
```

优秀题目【**HDU-7187**】

Slipper

Time Limit: 10000/5000 MS (Java/Others) Memory Limit: 524288/262144 K (Java/Others)
Total Submission(s): 1269 Accepted Submission(s): 312

Problem Description

Gi is a naughty child. He often does some strange things. Therefore, his father decides to play a game with him.

Gi's father is a senior magician, he teleports Gi and Gi's Slipper into a labyrinth. To simplify this problem, we regard the labyrinth as a tree with n nodes, rooted at node 1. Gi is initially at node s, and his slipper is at node t. In the tree, going through any edge between two nodes costs w unit of power.

Gi is also a little magician! He can use his magic to teleport to any other node, if the depth difference between these two nodes equals to k. That is, if two nodes u, v satisfying that $|dep_u - dep_v| = k$, then Gi can teleport from u to v or from v to u. But each time when he uses magic he needs to consume p unit of power. Note that he can use his magic any times.

Gi want to take his slipper with minimum unit of power.

Input

Each test contains multiple test cases. The first line contains the number of test cases ($1 \le T \le 5$). Description of the test cases follows.

The first line contains an integer n --- The number of nodes in the tree. $2 \le n \le 10^6$.

The following $n-1$ lines contains 3 integers u, v, w that means there is an edge between nodes u and v. Going through this edge costs w unit of power.
$1 \le u, v \le n, 1 \le w \le 10^6$.

The next line will contain two separated integers k, p. $1 \le k \le \max_{u \subset V}(dep_u), 0 \le p \le 10^6$.

The last line contains two positive integers s, t, denoting the positions of Gi and slipper. $1 \le s \le n, 1 \le t \le n$. It is guaranteed the $s \ne t$.

Output

For each test case:

Print an integer in a line --- the minimum unit of power Gi needs.

Sample Input

```
1
6
6 1 2
3 5 2
2 4 6
5 2 2
5 6 20
3 8
6 5
```

Sample Output

```
12
```

Hint

Example1: Gi can go from node 6 to node 1 using 2 units of power. Then he teleports from node 1 to node 2 using 8 units of power. Finally, he goes from node 2 to node 5 using 2 units of power. $Total\ cost = 2 + 8 + 2 = 12$

标答：

1003.Slipper

设树的深度为d，在第i层$(1 \leq i \leq d)$和第$i+1$层中新增两个点l_i, r_i，l_i连向所有第$i+1$层的点，r_i连向所有第i层的点，对于原来树中所有的点，向l_{dep_u+k-1}连一条权为p单向边，向r_{dep_u-k}连一条权值为p的单向边，在修改后的图中跑dijkstra求s到t的最短路即可，复杂度O(nlogn)。

做法：

每层建立虚点（虚点下标从n+1到n+1+maxdeep），向同层的所有点连一条单向边，然后每一层的所有点向非同层的深度距离为K的虚点连一条单向边，最后所有的虚点之间距离（深度）为K的连一条双向边，最后通过dijkstra求出start-end的最短距离。

【特殊数据】：

1
3
1 2 1000000
1 3 1000000
1 1
2 3

```
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;
typedef long double ld;
typedef pair<ll, ll>P;
const ll maxn = 1e6 + 7;
const ll inf = 1e18 + 7;
const ll mod = 1e9 + 7;
const ld pi = 3.14159265358979323846;
ll t, n, k, p,st,en,maxdeep;
string str;
struct Edges {
    ll to, w, next;
};
Edges edge[maxn << 2];
ll head[maxn << 2];
ll cnt;
ll dis[maxn << 2];
ll deep[maxn << 2];
bool vis[maxn<<2];
void init() {
    for (ll i = 0; i < maxn<<2; i++) {
        head[i] = -1;
        dis[i] = inf;
```

```
        deep[i] = -1;
        vis[i] = 0;
    }
    cnt = 0;
    maxdeep = -1;
}
void add(ll from, ll to, ll w) {
    edge[cnt].to = to;
    edge[cnt].w = w;
    edge[cnt].next = head[from];
    head[from] = cnt++;
}
void dfs(ll u, ll fa) {
    deep[u] = deep[fa] + 1;
    maxdeep = max(maxdeep, deep[u]);
    for (ll i = head[u]; i != -1; i = edge[i].next) {
        if (edge[i].to == fa)continue;
        dfs(edge[i].to, u);
    }
}
void dijkstra() {
    priority_queue<P, vector<P>, greater<P>>qq;
    dis[st] = 0;
    qq.push(make_pair(dis[st], st));
    P temp;
    while (!qq.empty()) {
        temp = qq.top();
        qq.pop();
        ll u = temp.second;
        if (vis[u])continue;
        vis[u] = 1;
        for (ll i = head[u]; i != -1; i = edge[i].next) {
            ll to = edge[i].to;
            if (!vis[to] && dis[to] > dis[u] + edge[i].w) {
                dis[to] = dis[u] + edge[i].w;
                qq.push(P(dis[to],to));
            }
        }
    }
}
void solve() {
    ios::sync_with_stdio(false); cin.tie(0); cout.tie(0);
    init();
    cin >> n;
    ll from, to, w;
    for (ll i = 0; i < n - 1; i++) {
        cin >> from >> to >> w;
```

```
        add(from, to, w);
        add(to, from, w);
    }
    deep[0] = -1;
    dfs(1, 0);
    cin >> k >> p;
    for (ll i = 1; i <= n; i++) {
        add(n + deep[i] + 1, i, 0);
        if (deep[i] + k <= maxdeep) {
            add(i, n + 1 + deep[i] + k, p);
        }
        if (deep[i] - k >= 0) {
            add(i, n + 1 + deep[i] - k, p);
        }
    }

    for (ll i = n + 1; i+k <= n + maxdeep + 1; i++) {
        add(i, i + k, p);
        add(i + k, i, p);
    }
    cin >> st >> en;
    dijkstra();
    cout << dis[en] << "\n";
}
int main(){
    ios::sync_with_stdio(false); cin.tie(0); cout.tie(0);
    cin >> t;
    while (t--) {
        solve();
    }
    return 0;
}
```

三、【最小生成树】

1、【Kruskal算法】

适用范围：稀疏图

原理：“加边法”。将所有的边从大到小排序，每次选出权值最小的并且顶点不在一棵树的一条边加入到边集合中，重复此过程，直到所有的顶点都在一棵树中或选出了n-1条边即可。

具体过程：

1、把图中的所有边按代价从小到大排序；

2、 把图中的n个顶点看成独立的n棵树组成的森林；

3、 按权值从小到大选择边，所选的边连接的两个顶点ui,viui,vi,应属于两颗不同的树，则成为最小生成树的一条边，并将这两颗树合并作为一颗树。

4、重复(3),直到所有顶点都在一颗树内或者有n-1条边为止。

```
using namespace std;
typedef long long ll;
struct Edges {
    ll from,to, w;
};
Edges edge[2000005];
ll  cnt,n, m,ans,e1,e2;//n为顶点个数，m为边的个数
ll fa[5005];//并查集需要
//sort 结构体权值从小到大
bool cmp(Edges a, Edges b)
{
    return a.w < b.w;
}
//并查集
ll find_set(ll x)
{
    if (x != fa[x])return fa[x] = find_set(fa[x]);
    return x;
}
void Kruskal()
{
    sort(edge, edge + m, cmp);
    for (ll i = 0; i < m; i++)
    {
         e1 = find_set(edge[i].from);
         e2 = find_set(edge[i].to);
        if (e1 == e2)continue;
        fa[e2] = e1;
        ans += edge[i].w;
        cnt++;//记录边数
```

```
        if (cnt == n - 1) break;//边数为顶点数-1时退出
    }
}
```

2、【Prim算法】

适用范围：稠密图

原理：“加点法”。维护两个集合，一个是已经选取的点{a，b}，一个是还未选取的点{c,d,e,f}。每次选择权值最小的边的点加入到集合中，直到选取了所有的点加入集合。

具体过程：

1、图的所有顶点集合为VV；初始令集合u={s},v=V−uu={s},v=V−u;

2、在两个集合u,vu,v能够组成的边中，选择一条代价最小的边(u0,v0)，加入到最小生成树中，并把v0并入到集合u中。

3、重复上述步骤，直到最小生成树有n-1条边或者n个顶点为止。

```
#include<bits/stdc++.h>
using namespace std;
#define re register
#define il inline
il int read()
{
    re int x = 0, f = 1; char c = getchar();
    while (c < '0' || c>'9') { if (c == '-') f = -1; c = getchar();
}
    while (c >= '0' && c <= '9') x = (x << 3) + (x << 1) + (c ^
48), c = getchar();
    return x * f;
}//快读，不理解的同学用cin代替即可
#define inf 123456789
#define maxn 5005
#define maxm 200005
struct edge
{
    int to, w, next;
```

```
}e[maxm << 1];
//注意是无向图，开两倍数组
int head[maxn], dis[maxn], cnt, n, m, tot, now=1, ans;
//已经加入最小生成树的的点到没有加入的点的最短距离，比如说1和2号节点已经加入了最小生成树，那么dis[3]就等于min(1->3,2->3)
bool vis[maxn];
//链式前向星加边
il void add(int from, int to, int w)
{
    e[cnt].to = to;
    e[cnt].w = w;
    e[cnt].next = head[from];
    head[from] = cnt++;
}
//读入数据
il void init()
{
    n = read(), m = read();
    for (int i = 0; i <= n; i++)
    {
        head[i] = -1;
    }
    for (re int i = 0, from, to, w; i < m; i++)
    {
        from = read(), to = read(), w = read();
        add(from, to, w), add(to, from, w);
    }
}
il int prim()
{
    //先把dis数组附为极大值
    for (re int i = 0; i <= n; i++)
    {
        dis[i] = inf;
    }
    //这里要注意重边（多条相同起点和终点），所以要用到min
    for (re int i = head[1]; i!=-1; i = e[i].next)
    {
        dis[e[i].to] = min(dis[e[i].to], e[i].w);
    }
    while (tot != n-1)//最小生成树边数等于点数-1
    {
        re int minn = inf;//把minn置为极大值
        vis[now] = 1;//标记点已经走过
        //枚举每一个没有使用的点
        //找出最小值作为新边
        //注意这里不是枚举now点的所有连边，而是1~n
```

```
        for (re int i = 0; i <= n; i++)
        {
            if (!vis[i] && minn > dis[i])
            {
                minn = dis[i];
                now = i;
            }
        }
        ans += minn;
        //枚举now的所有连边，更新dis数组
        for (re int i = head[now]; i!=-1; i = e[i].next)
        {
            re int to = e[i].to;
            if (dis[to] > e[i].w && !vis[to])
            {
                dis[to] = e[i].w;
            }
        }
        tot++;
    }
    return ans;
}
```

四、【拓扑排序】

定义：在一个有向图中，对所有的节点进行排序，要求没有一个节点指向它前面的节点。

求出的解具有不唯一性质；

适用范围：两个点具有先后完成的顺序关系。

实际应用：1、可以用来查找一个有向图中最长路径长度

2、多个入度为0的点所构成的图中某一点在第几层。

3、从图中的某一点能否走到另一点（排序后从后往前可以直接确定，但从前往后需要另外加强条件）？

```
const int MAX = 1e6;     //数组大小初始化
struct Edges {
    int to, w, next; //（边的终点编号、权重、同起点的下一条边）
};
Edges edge[MAX];  //记录所有点的信息
ll head[MAX];     //例：head[1]=5;记录从1点出发的最后输入的一条边5
queue<int>q;
int in[maxn];//入度
```

```
vector<int>ans;//存放结果
void solve(){
    for(int i=0;i<n;i++)if(!in[i])q.push(i);
    while(!q.empty()){
          int from=q.front();q.pop();
          ans.push_back(from)
          for(int i=head[from];i!=-1;i=edge[i].next)
          {
                int point=edge[i].to;
              in[point]--;
              if(!in[point])
                  q.push(point);
          }
    }
}
```

例题：

```
#include<iostream>
#include<string>
#include<cstring>
#include<algorithm>
#include<stdio.h>
#include<vector>
#include<stack>
#include<bitset>
#include<cstdlib>
#include<cmath>
#include<set>
#include<list>
#include<deque>
#include<map>
#include<queue>
#include<istream>
#include<sstream>
using namespace std;
typedef long long ll;
const int MAXN = 1010;  //数组大小初始化
struct Edges {
    int to, next; //（边的终点编号、权重、同起点的下一条边）
};
Edges edge[3000010];  //记录所有点的信息
ll head[MAXN];     //例：head[1]=5;记录从1点出发的最后输入的一条边5
ll cnt;            //记录每组数据的编号
void init()        //初始化
```

```
{
    memset(head, -1, sizeof(head));
    cnt = 0;
}
void add(ll from, ll to) //添加边
{
    cnt++;
    edge[cnt].to = to;
    edge[cnt].next = head[from];
    head[from] = cnt;
}
queue<int>q;
int in[MAXN], is[MAXN], vis[MAXN][MAXN], step[MAXN], arr[MAXN];
vector<int>ans;//存放结果
int n, m, res, num;
void solve() {

    for (int i = 1; i <= n; i++) {
        if (!in[i]) {
            step[i] = 1;
            q.push(i);
        }
    }
    while (!q.empty()) {
        int from = q.front(); q.pop();
        ans.push_back(from);
            for (int i = head[from]; i != -1; i = edge[i].next)
            {
                int point = edge[i].to;
                step[point] = step[from] + 1;
                res = max(res, step[point]);
                in[point]--;
                if (!in[point])
                    q.push(point);
            }
    }
}


int main() {
    ios::sync_with_stdio(false); cin.tie(0); cout.tie(0);
    init();
    cin >> n >> m;
    while (m--) {
        memset(is, 0, sizeof(is));
        memset(arr, 0, sizeof(arr));
        cin >> num;
```

```
        for (int i = 1; i <= num; i++)
        {
            cin >> arr[i];
            is[arr[i]] = 1;
        }
        for (int i = arr[1]+1; i <= arr[num]; i++)
        {
            if (!is[i])
            {
                for (int j = 1; j <= num; j++)
                {
                    if (!vis[i][arr[j]])
                    {
                        vis[i][arr[j]] = 1;
                        add(i, arr[j]);
                        in[arr[j]]++;
                    }
                }
            }
        }
    }
    solve();
    cout << res << endl;
    return 0;

}
```

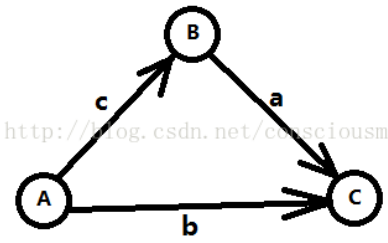
五、【差分约束系统】

B - A <= c　(1)

C - B <= a　(2)

C - A <= b　(3)

如果要求C-A的最大值，可以知道max（C-A）= min(b,a+c),而这正对应了下图中A到C的最短路。



若给定约束条件下无解，则存在负环，利用spfa判断是否存在负环。

注：求最小值是求最长路（改一下spfa的松弛条件即可），求最大值则为最短路

差分约束系统的解法如下：

1、 根据条件把题意通过变量组表达出来得到不等式组，注意要发掘出隐含的不等式，比如说前后两个变量之间隐含的不等式关系。

2、 进行建图：

首先根据题目的要求进行不等式组的标准化。

(1)、如果要求取最小值，那么求出最长路，那么将不等式全部化成xi - xj >= k的形式，这样建立j->i的边，权值为k的边，如果不等式组中有xi - xj > k，因为一般题目都是对整形变量的约束，化为xi - xj >= k+1即可，如果xi - xj = k呢，那么可以变为如下两个：xi - xj >= k，xi - xj <= k,进一步变为xj - xi >= -k，建立两条边即可。

(2)、如果求取的是最大值，那么求取最短路，将不等式全部化成xi - xj <= k的形式，这样建立j->i的边，权值为k的边，如果像上面的两种情况，那么同样地标准化就行了。

(3)、如果要判断差分约束系统是否存在解，一般都是判断环，选择求最短路或者最长路求解都行，只是不等式标准化时候不同，判环地话，用spfa即可，n个点中如果同一个点入队超过n次，那么即存在环。

值得注意的一点是：建立的图可能不联通，我们只需要加入一个超级源点，比如说求取最长路时图不联通的话，我们只需要加入一个点s，对其他的每个点建立一条权值为0的边图就联通了，然后从s点开始进行spfa判环。最短路类似。

3、 建好图之后直接spfa或bellman-ford求解，不能用dijstra算法，因为一般存在负边，注意初始化的问题。

六、【TarJan】

用途：

1、可以求出无向图（有向图）的割点与割边

2、可以求出所有的连通块

前置：

1、割点：删除该点以及其所连的边，无向图（有向图）的（强）连通分量增加。

2、割边：割点所连的所有边。

无向图：

模板：

```
//核心模板
//链式前向星加边
//由于是无向图，加边时要加两条边from->to,to->from
int head[maxn],dfn[maxn], low[maxn];
int ans,d;//存放割点答案数量
bool cut[maxn];
void tarjan(int from, int fa) {
    dfn[from] = low[from] = ++d;
    int child = 0;
    for (int i = head[from]; i != -1; i = edge[i].next){
        int to = edge[i].to;
        if (!dfn[to]){
            tarjan(to, fa);
            low[from] = min(low[from], low[to]);
            if (low[to] >= dfn[from] && from != fa) cut[from] = 1;
            if (from == fa)child++;
        }
        low[from] = min(low[from], dfn[to]);
    }
        if (from == fa && child >= 2)cut[from] = 1;
}
//由于tarjan图各个连通块之间可能不连通，主函数内要这样写
for (int i = 1; i <= n; i++)
        if (!dfn[i])
            tarjan(i, i);
```

有向图

模板：

```
int color[maxn],dfn[maxn], low[maxn],stack_[maxn];
int num[maxn];//同属于同一颜色连通块的节点个数
bool exist[maxn];
int d,id,top;
void tarjan(int x) {
    dfn[x] = low[x] = ++d;
    stack_[++top]=x;
    exist[x]=1;
    for (int i = head[x]; i != -1; i = edge[i].next){
        int to = edge[i].to;
        if (!dfn[to]){
```

```
            tarjan(to);
            low[x] = min(low[x], low[to]);
        }
        else if(exist[to]){
            low[x] = min(low[x], low[to]);
        }
    }
        if(low[x]==dfn[x])
        {
            id++;
            do{
               color[stack_[top]]=id;
                num[id]++;
                exist[stack_[top]]=0;
            }while(x!=stack_[top--]);
        }
}
```

缩点模板

```
#include<bits/stdc++.h>
#define int long long
using namespace std;
const int maxn=1e5+7;
const int inf=1e18;
int n,m,from,to;
struct node{
    int from,to,val;
    int next;
}edge[maxn],edge2[maxn];
int head[maxn],head2[maxn];
int cnt,cnt2;
int dfn[maxn],low[maxn];//dfs序以及子树上的最小序号
int belong[maxn];//缩点同一强连通块的所有点属于该强连通分量的根节点
int idx;//dfs序
int st[maxn],top,vis[maxn];//模拟栈,以及栈内元素数量,当前是否在栈内
int in[maxn];//入度，拓扑排序
int dp[maxn];//到新图i点的最长路
int kval[maxn];//一个强联通分量的总价值
void init(){
    memset(head,-1,sizeof(head));
    cnt=1;
    memset(head2,-1,sizeof(head2));
    cnt2=1;
}
void add(int from,int to,int val){
```

```
    edge[cnt].to=to;
    edge[cnt].val=val;
    edge[cnt].from=from;
    edge[cnt].next=head[from];
    head[from]=cnt++;
}
void add2(int from,int to,int val){
    edge2[cnt2].to=to;
    edge2[cnt2].val=val;
    edge2[cnt2].from=from;
    edge2[cnt2].next=head2[from];
    head2[from]=cnt2++;
}
void tarjan(int u){
    dfn[u]=low[u]=++idx;
    st[++top]=u,vis[u]=1;
    for(int i=head[u];i!=-1;i=edge[i].next){
        int v=edge[i].to;
        if(!dfn[v]){//若未被访问过
            tarjan(v);
            low[u]=min(low[u],low[v]);
        }else if(vis[v]){
            low[u]=min(low[u],dfn[v]);
        }
    }
    if(dfn[u]==low[u]){//找到了一个强连通分量，若为不能构成强连通分量的单个点，则自成单个强连通
        while(true){
            int to=st[top--];
            belong[to]=u;
            vis[to]=0;
            if(u==to)break;
            kval[u]+=kval[to];
        }
    }
}
void topu(){
    //遍历所有点，找到入度为0的根为本身的点，放入栈中。
    stack<int>ss;
    for(int i=1;i<=n;i++){
        if(!in[i]&&belong[i]==i)ss.push(i);
        dp[i]=kval[i];
    }
    while(!ss.empty()){
        int u=ss.top();
        ss.pop();
        for(int i=head2[u];i!=-1;i=edge2[i].next){
```

```
            int v=edge2[i].to;
            dp[v]=max(dp[v],dp[u]+kval[v]);
            in[v]--;
            if(!in[v])ss.push(v);
        }
    }
    int ans=0;
    for(int i=1;i<=n;i++){
        ans=max(ans,dp[i]);
    }
    cout<<ans<<endl;
}
signed main(){
    ios::sync_with_stdio(0),cin.tie(0),cout.tie(0);
    cin>>n>>m;
    init();
    for(int i=1;i<=n;i++)cin>>kval[i];
    for(int i=1;i<=m;i++){
        cin>>from>>to;
        add(from,to,1);
    }
    for(int i=1;i<=n;i++){
        if(!dfn[i])tarjan(i);
    }
    for(int i=1;i<=m;i++){
        int ffrom,tto;
        ffrom=belong[edge[i].from];
        tto=belong[edge[i].to];
        if(ffrom!=tto){
            add2(ffrom,tto,1);
            in[tto]++;
        }
    }
    topu();//利用拓扑进行dp，dp[v]=max(dp[v],dp[u]+val);
    return 0;
}
```

七、【网络流】

1、dini算法

核心模板：

```
const int maxn = 2050;
const int INF = 1e9 + 7;
struct Edges {
    int to, w, next;
};
Edges edge[maxn<<1];
int head[maxn], deep[maxn];
int cnt;
ll n, m;//n为汇点，源点默认为1，m是边的数量
void add(int from, int to,int w) {
    edge[cnt].to = to;
    edge[cnt].w = w;
    edge[cnt].next = head[from];
    head[from] = cnt++;
}
void init()
{
    memset(head, -1, sizeof(head));
    cnt = 0;
}
bool bfs()
{
    memset(deep, -1, sizeof(deep));
    deep[1] = 0;
    queue<int>q;
    q.push(1);
    while (!q.empty())
    {
        int from = q.front();
        q.pop();
        for (int i = head[from]; i != -1; i = edge[i].next)
        {
            int to = edge[i].to;
            if (deep[to] == -1 && edge[i].w)//子节点没被搜索过并且子节点
的剩余流量不为0。
            {
                deep[to] = deep[from] + 1;
                q.push(to);
            }
        }
    }
    return deep[n] != -1;//若所有通往汇点n的路中均有一条或多条路剩余流量为0
```

```
                                //则不可能到达汇点，断开，汇点不被更新，说明最大流搜
索结束；
}
int dfs(int from, int rest)
{

    if (from == n)return rest;
    int restnow = rest;
    for (int i = head[from]; i != -1; i = edge[i].next)
    {
        int to = edge[i].to;
        if (deep[to] == (deep[from] + 1) && edge[i].w > 0)
        {
            int allow = dfs(to, min(restnow, edge[i].w));
            edge[i].w -= allow, edge[i ^ 1].w += allow;
            restnow -= allow;
            if (restnow == 0)break;
        }
    }
    return rest - restnow;
}
int dini()
{
    int ans = 0;
    while (bfs())ans += dfs(1, INF);
    return ans;
}
int main() {
    ios::sync_with_stdio(false); cin.tie(0); cout.tie(0);
    init();
    int from, to, w;
    cin >> n >> m >> x;
    for (int i = 0; i < m; i++)
    {
        cin >> from >> to >> w;
        add(from, to, w), add(to, from, 0);
    }
    int ans = dini();
    cout<<ans<<endl;
    return 0;
}
```

八、【并查集】

```
//并查集_未优化
ll find_set(ll x)
{
    return x!=fa[x]?find_set(fa[x]):x;
}

//并查集_路径压缩优化
ll find_set(ll x)
{
    if (x != fa[x])return fa[x] = find_set(fa[x]);
    return x;
}

//并查集_启发式合并
//rank[maxn]代表深度，把深度小的合并到深度大的上
void merge(int u,int v){
    int t1=find(u);
    int t2=find(v);
    if(t1!=t2) {//启发式算法的核心代码，将节点深度小的并到节点深度大的上面。
        if(rank[t1]>rank[t2]){
            f[t2]=t1;
        }
        else if(rank[t1]<rank[t2]){
            f[t1]=t2;
        }
        else {//深度相等时将其中一个的深度增加
            f[t2]=t1;
            rank[t1]++;
        }
    }

}
```

【树】

一、【线段树】

适用范围：

1、遇到查询某个区间的和

2、遇到某个区间进行统一的某项操作（全部+a，全部*b）

作用：

单点修改、区间修改、单点查询、区间查询

复杂度：

O(logn)

核心模板：

用结构体的模板：

POJ:http://poj.org/problem?id=3468

```
#include<iostream>
#include<algorithm>
#include<cmath>
#include<stdio.h>
using namespace std;
typedef long long ll;
#define For(i,a,b) for(ll (i)=a;(i)<=(b);(i)++)
#define ls rt<<1
#define rs rt<<1|1
ll N, num;
const ll maxn = 1e5 + 7;
struct node {
    ll l, r, sum;
    ll lazy;
    ll len() { return r - l + 1; }
}tr[maxn << 2];
ll arr[maxn];
void push_up(ll rt) {
    tr[rt].sum = tr[ls].sum + tr[rs].sum;
}
void push_down(ll rt)
{
    if (tr[rt].lazy)
    {
        tr[ls].sum += tr[rt].lazy * tr[ls].len();
```

```
        tr[rs].sum += tr[rt].lazy * tr[rs].len();
        tr[ls].lazy += tr[rt].lazy;
        tr[rs].lazy += tr[rt].lazy;
        tr[rt].lazy = 0;
    }
}


void build(ll l, ll r, ll rt) {
    tr[rt].l = l, tr[rt].r = r;
    if (l == r) {
        scanf("%lld", &tr[rt].sum);//这里输入，也可以放到下面，改成
tr[rt].sum=arr[l];
        tr[rt].lazy = 0;
        return;
    }
    ll mid = l + r >> 1;
    build(l, mid, ls);
    build(mid + 1, r, rs);
    push_up(rt);
}

void update(ll l, ll r, ll rt, ll add) {//区间修改其实可以当单点修改来用
    if (l <= tr[rt].l && tr[rt].r <= r)
    {
        tr[rt].lazy += add;
        tr[rt].sum += add * tr[rt].len();
        //cout << rt << " " << tr[rt].l << " " << tr[rt].r << ":"
<< tr[rt].len() << endl;
        return;
    }
    ll mid = tr[rt].l + tr[rt].r >> 1;
    push_down(rt);
    if (l <= mid)update(l, r, ls, add);
    if (r > mid)update(l, r, rs, add);
    push_up(rt);
}

ll query(ll l, ll r, ll rt) { //区间查询

    if (l <= tr[rt].l && tr[rt].r <= r) return tr[rt].sum;
    push_down(rt);
    ll mid = tr[rt].l + tr[rt].r >> 1;
    ll ans = 0;
    if (l <= mid)ans += query(l, r, ls);
    if (r > mid)ans += query(l, r, rs);
    push_up(rt);
```

```
    return ans;
}

int main() {
    ios::sync_with_stdio(false); cin.tie(0); cout.tie(0);
    scanf("%lld%lld", &N, &num);
        build(1, N, 1);
        ll a, b, c;
        char op;
        while (num--) {
            scanf(" %c", &op);
            if (op== 'Q') {
                scanf("%lld%lld", &a, &b);
                printf("%lld\n", query(a, b, 1));
            }
            else {
                scanf("%lld%lld%lld", &a, &b, &c);
                update(a, b, 1, c);
            }
        }
    return 0;
}
```

```
#define maxn 50005  //元素总个数
#define ls rt<<1        //左子叶
#define rs rt<<1|1      //右子叶
int Sum[maxn << 2], Add[maxn << 2]; //Sum建树求和，Add懒惰标记
int A[maxn], n;                     //A[]用来存放数据
void pushup(int rt)                 //子节点更新，递归回溯时父节点也需更新
{
    Sum[rt] = Sum[ls] + Sum[rs];
}
void pushdown(int rt, int len)      //下放标记(加法)
{
    if (Add[rt]) {
        Sum[ls] += Add[rt] * (len - len / 2);           //修改子节点
的Sum使之与对应的Add相对应
        Sum[rs] += Add[rt] * (len / 2);
        Add[ls] += Add[rt];                             //下推标记
        Add[rs] += Add[rt];
        Add[rt] = 0;                                    //清除本节点标记
    }
}
```

```
//建树
void build(int l, int r, int rt)
{
    Add[rt] = 0;
    if (l == r)
    {
        Sum[rt] = A[l];
        //cin>>Sum[rt];其实也可以直接赋值，
        return;
    }
    int mid = (l + r) >> 1;
    build(l, mid, ls);
    build(mid + 1, r, rs);
    pushup(rt);
}
//修改单个节点
void Update(int l, int r, int rt, int pos, int add) {        //l,r表
示当前节点区间，rt表示当前节点编号,pos代表在原数组中的位置，add代表要加上多少
    if (l == r&&l==pos) {
        Sum[rt] += add;
        return;
    }
    int mid = (l + r) >> 1;
    pushdown(rt, r - l + 1);
    if (pos <= mid) {                                        //剪枝
         Update(l, mid, ls, pos, add);
    }
    else {
         Update(mid + 1, r, rs, pos, add);
    }
    pushup(rt);
}
//单点查询
int Query(int l, int r, int rt, int pos) {       //l,r表示当前节点区
间，rt表示当前节点编号,pos代表在原数组中的位置，add代表要加上多少
    if (l == r && l == pos) {
        return Sum[rt];
    }
    int mid = (l + r) >> 1;
    pushdown(rt, r - l + 1);
    if (pos <= mid) {                                        //剪枝
         Query(l, mid, ls, pos);
    }
    else {
         Query(mid + 1, r, rs, pos);
    }
    pushup(rt);
```

```
}
//区间修改
void Update(int l, int r, int rt, int L, int R, int add) {//L,R表示操作区间，l,r表示当前节点区间，rt表示当前节点编号
    if (L <= l && r <= R) {                                    //如果本区间完全在操作区间[L,R]以内
        Sum[rt] += add * (r - l + 1);                          //更新数字和，向上保持正确
        Add[rt] += add;                                        //增加Add标记，表示本区间的Sum正确，子区间的Sum仍需要根据Add的值来调整
        return;
    }
    int mid = (l + r) >> 1;
    pushdown(rt, r - l + 1);                                   //下推标记
    if (L <= mid) {                                            //剪枝
        Update(l, mid, ls, L, R, add);
    }
    if (R > mid) {
        Update(mid + 1, r, rs, L, R, add);
    }
    pushup(rt);                                                //更新本节点信息
}
//区间查询
int Query(int l, int r, int rt, int L, int R) {                //L,R表示操作区间，l,r表示当前节点区间，rt表示当前节点编号
    if (L <= l && r <= R) {
        return Sum[rt];                                        //在区间内，直接返回
    }
    int mid = (l + r) >> 1;
    pushdown(rt, r - l + 1);                                   //下推标记，否则Sum可能不正确
    int ans = 0;
    if (L <= mid) {
        ans += Query(l, mid, ls, L, R);
    }
    if (R > mid) {
        ans += Query(mid + 1, r, rs, L, R);
    }
    pushup(rt);
    return ans;
}
```

例题&模板

N个气球排成一排，从左到右依次编号为1,2,3....N.每次给定2个整数a b(a <= b),lele便为骑上他的“小飞鸽"牌电动车从气球a开始到气球b依次给每个气球涂一次颜色。但是N次以后lele已经忘记了第I个气球已经涂过几次颜色了，你能帮他算出每个气球被涂过几次颜色吗？

```
#include<iostream>
#include<string>
#include<cstring>
#include<algorithm>
#include<stdio.h>
#include<vector>
#include<stack>
#include<bitset>
#include<cstdlib>
#include<cmath>
#include<set>
#include<list>
#include<deque>
#include<map>
#include<queue>
using namespace std;
#define maxn 100007  //元素总个数
#define ls rt<<1
#define rs rt<<1|1
int Sum[maxn << 2], Add[maxn << 2]; //Sum求和，Add为懒惰标记
int A[maxn], n;                     //存原数组数据下标[1,n]
void pushup(int rt)                 //子节点更新，递归回溯时父节点也需更新
{
    Sum[rt] = Sum[ls] + Sum[rs];
}
void pushdown(int rt, int len)      //下放标记(加法)
{
    if (Add[rt]) {
        Sum[ls] += Add[rt] * (len - len / 2);           //修改子节点的Sum使之与对应的Add相对应
        Sum[rs] += Add[rt] * (len/2);
        Add[ls] += Add[rt];                             //下推标记
        Add[rs] += Add[rt];
        Add[rt] = 0;                                    //清除本节点标记
    }
}
//建树
void build(int l, int r, int rt)
{
    Add[rt] = 0;
    if (l == r)
    {
```

```
        Sum[rt] = 0;
        return;
    }
    int mid = (l + r) >> 1;
    build(l, mid, ls);
    build(mid + 1, r, rs);
    pushup(rt);
}
//修改单个节点
void Update(int l, int r, int rt, int pos, int add) {       //l,r表示当前节点区间，rt表示当前节点编号,pos代表在原数组中的位置，add代表要加上多少
    if (l == r&&l==pos) {
        Sum[rt] += add;
        return;
    }
    int mid = (l + r) >> 1;
    pushdown(rt, r - l + 1);
    if (pos <= mid) {                                       //剪枝
        return Update(l, mid, ls, pos, add);
    }
    else {
        return Update(mid + 1, r, rs, pos, add);
    }
    pushup(rt);
}
//单点查询
int Query(int l, int r, int rt, int pos) {      //l,r表示当前节点区间，rt表示当前节点编号,pos代表在原数组中的位置，add代表要加上多少
    if (l == r&&l==pos) {
        return Sum[rt];
    }
    int mid = (l + r) >> 1;
    pushdown(rt, r - l + 1);
    if (pos <= mid) {                                       //剪枝
        return Query(l, mid, ls, pos);
    }
    else {
        return Query(mid + 1, r, rs, pos);
    }
}
//区间修改
void Update(int l, int r, int rt, int L, int R, int add) {//L,R表示操作区间，l,r表示当前节点区间，rt表示当前节点编号
    if (L <= l && r <= R) {                                 //如果本区间完全在操作区间[L,R]以内
        Sum[rt] += add * (r - l + 1);                       //更新数字和，向上保持正确
```

```
        Add[rt] += add;                                    //增加Add标记，表示本区间的Sum正确，子区间的Sum仍需要根据Add的值来调整
        return;
    }
    int mid = (l + r) >> 1;
    pushdown(rt, r - l + 1);                               //下推标记
    if (L <= mid) {                                        //剪枝
        Update(l, mid, ls, L, R, add);
    }
    if (R > mid) {
        Update(mid + 1, r, rs, L, R, add);
    }
    pushup(rt);                                            //更新本节点信息
}
//区间查询
int Query(int l, int r, int rt, int L, int R) {            //L,R表示操作区间，l,r表示当前节点区间，rt表示当前节点编号
    if (L <= l && r <= R) {
        return Sum[rt];                                    //在区间内，直接返回
    }
    int mid = (l + r) >> 1;
    pushdown(rt, r - l + 1);                               //下推标记，否则Sum可能不正确
    int ans = 0;
    if (L <= mid) {
        ans += Query(l, mid, ls, L, R);
    }
    if (R > mid) {
        ans += Query(mid + 1, r, rs, L, R);
    }
    pushup(rt);
    return ans;
}

int main() {
    int n;
    while (cin >> n)
    {
        if (n == 0)break;
        build(1, n, 1);
        for (int i = 1; i <= n; i++)
        {
            int l, r;
            cin >> l >> r;
            Update(1, n, 1, l, r, 1);
```

```
        }
        for (int i = 1; i <= n; i++)
        {
            if (i != n) {
                cout << Query(1, n, 1, i) << " ";
            }
            else {
                cout << Query(1, n, 1, i);
            }
        }
        cout << endl;
    }
    return 0;

}
```

1、【势能线段树】

例子&模板

1276.花神游历各国

```
#include<iostream>
#include<algorithm>
#include<cmath>
#define ls k<<1
#define rs k<<1|1
using namespace std;
typedef long long ll;
const ll maxn = 1e5 + 7;
ll n, m, a[maxn], f;
struct tree {
    ll r, l, sum, mx;
};
tree tr[maxn<<2];//开四倍空间
void pushup(ll k) {
    tr[k].sum = tr[ls].sum + tr[rs].sum;
    tr[k].mx = max(tr[ls].mx, tr[rs].mx);
}
void build(ll k,ll l,ll r) {
    tr[k].l = l, tr[k].r = r;
    if (l == r) {
        tr[k].sum =tr[k].mx=a[l];
        return;
    }
    ll mid = (l + r) >> 1;
```

```
    build(ls, l, mid);
    build(rs, mid + 1, r);
    pushup(k);
}
void update(ll k,ll l, ll r) {
    if (tr[k].l == tr[k].r) {
        tr[k].mx = sqrt(tr[k].mx);
        tr[k].sum = tr[k].mx;
        return;
    }
    ll mid = (tr[k].l + tr[k].r) >> 1;
    if (l <=mid && tr[ls].mx > 1)update(ls, l, r);
    if (r > mid && tr[rs].mx > 1)update(rs, l, r);
    pushup(k);
}
ll query(ll k, ll l, ll r) {
    if (tr[k].l >= l && tr[k].r <= r) return tr[k].sum;
    ll mid = (tr[k].l + tr[k].r) >> 1;
    ll ans = 0;
    if (l <= mid)ans+= query(ls, l, r);
    if (r > mid)ans += query(rs, l, r);
    return ans;
}

int main() {
    ios::sync_with_stdio(false); cin.tie(0); cout.tie(0);
    cin >> n;
    for (ll i = 1; i <= n; i++)cin >> a[i];
    build(1, 1, n);
    cin >> m;
    while (m--) {
        ll l, r;
        cin >> f >> l >> r;
        if (f == 1)cout << query(1,l, r) << endl;
        else if (f == 2)update(1,l, r);
    }
    return 0;
}
```

二、【树链剖分】

模板：

```
//u是当前节点
//fa是当前节点的父节点
void dfs1(int u, int fa) {
    size[u] = 1;//表示刚搜到u的时候以u为根的子树里只有u一个节点
    for (int i = head[u]; ~i; i = e[i].nx) {
        int v = e[i].v;//连向的节点
        if (v != fa) {//因为连的是无相边，而且是树，不能往上搜，所以我们要判断u是不是从fa搜过来，也就是判断v是不是u的子节点，也可以写作!dep[v](没有被搜到过)
            dep[v] = dep[u] + 1;//v的深度是当前节点的深度+1
            f[v] = u;//记录一下父亲
            dfs1(v, u);//继续往下搜，一直搜到叶节点为止
            size[u] += size[v];//往上回溯，更新以u为根的子树的size
            if (size[v] > size[son[u]]) son[u] = v;//重儿子是节点个数更多的子树，如果以u的子树中，以v为根的子树节点多，那就更新一下u的重儿子为v
        }
    }
}
//u是当前节点
//t是所在链的顶端
void dfs2(int u, int t) {
    id[u] = ++cnt;//给这个点一个新的编号
    a[cnt] = w[u];//记录这个编号下点的值
    top[u] = t;//记录u所在链的顶端为t
    if (son[u]) dfs2(son[u], t);//先走重儿子，如果没有重儿子说明没有儿子，下面的for循环也不会进去，相当于直接return。
    for (int i = head[u]; ~i; i = e[i].nx) {
        int v = e[i].v;//搜轻儿子
        if (v != f[u] && v != son[u])//判断是否是轻儿子 ，重儿子之前走过了，跳过。
                    dfs2(v, v);//以轻儿子为顶的链
    }
}
```

三、【树状数组】

```
int n;
int a[1005],c[1005]; //对应原数组和树状数组

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

void updata(int i,int k){    //在i位置加上k
```

```
    while(i <= n){
        c[i] += k;
        i += lowbit(i);
    }
}

int getsum(int i){        //求A[1 - i]的和
    int res = 0;
    while(i > 0){
        res += c[i];
        i -= lowbit(i);
    }
    return res;
}
```

1、【二维树状数组】

```
void Modify(int i, int j, int delta){
       A[i][j]+=delta;
       for(int x = i; x< A.length; x += lowbit(x)){
        for(int y = j; y <A[i].length; y += lowbit(y)){
          C[x][y] += delta;
        }
    }
}
//i,j矩阵内的值
    int Sum(int i, int j){
      int result = 0;
      for(int x = i; x > 0; x -= lowbit(x)) {
        for(int y = j; y > 0; y -= lowbit(y)) {
            result += C[x][y];
        }
      }
    return result;
   }
```

比如：

```
    Sun(1,1)=C[1][1];  Sun(1,2)=C[1][2]; Sun(1,3)=C[1][3]+C[1]
[2];...
    Sun(2,1)=C[2][1];  Sun(2,2)=C[2][2]; Sun(2,3)=C[2][3]+C[2]
[2];...
    Sun(3,1)=C[3][1]+C[2][1]; Sun(3,2)=C[3][2]+C[2][2]
```

四、【LCA】

```
#include <iostream>
#include <cstdio>
#include <cstring>
#include <algorithm>
using namespace std;
struct zzz {
    int t, nex;
}e[500010 << 1]; int head[500010], tot;
void add(int x, int y) {
    e[++tot].t = y;
    e[tot].nex = head[x];
    head[x] = tot;
}
int depth[500001], fa[500001][22], lg[500001];
void dfs(int now, int fath) {  //now表示当前节点，fath表示它的父亲节点
    fa[now][0] = fath; depth[now] = depth[fath] + 1;
    for(int i = 1; i <= lg[depth[now]]; ++i)
        fa[now][i] = fa[fa[now][i-1]][i-1]; //这个转移可以说是算法的核心之一
                                            //意思是now的2^i祖先等于now的2^(i-1)祖先的2^(i-1)祖先
                                                //2^i = 2^(i-1) + 2^(i-1)
    for(int i = head[now]; i; i = e[i].nex)
        if(e[i].t != fath) dfs(e[i].t, now);
}
int LCA(int x, int y) {
    if(depth[x] < depth[y]) //用数学语言来说就是：不妨设x的深度 >= y的深度
        swap(x, y);
    while(depth[x] > depth[y])
        x = fa[x][lg[depth[x]-depth[y]] - 1]; //先跳到同一深度
    if(x == y)  //如果x是y的祖先，那他们的LCA肯定就是x了
        return x;
    for(int k = lg[depth[x]] - 1; k >= 0; --k) //不断向上跳（lg就是之前说的常数优化）
        if(fa[x][k] != fa[y][k])  //因为我们要跳到它们LCA的下面一层，所以它们肯定不相等，如果不相等就跳过去。
            x = fa[x][k], y = fa[y][k];
    return fa[x][0];  //返回父节点
}
int main() {
    int n, m, s; scanf("%d%d%d", &n, &m, &s);
    for(int i = 1; i <= n-1; ++i) {
        int x, y; scanf("%d%d", &x, &y);
        add(x, y); add(y, x);
    }
```

```
for(int i = 1; i <= n; ++i) //预先算出log_2(i)+1的值，用的时候直接调用就可以了
      lg[i] = lg[i-1] + (1 << lg[i-1] == i);  //看不懂的可以手推一下

    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;
}
```

五、【主席树求区间前k大和】

```
#include <bits/stdc++.h>
#define int long long
#define mid (left+right)/2
using namespace std;
const int maxn = 1e5 + 10;
int n, m, q, tot = 0;
int a[maxn], b[maxn], sum[maxn];
int T[maxn], tree[maxn * 20], L[maxn * 20], R[maxn * 20];
int sumt[maxn * 20];
// T[i]存的是第i棵树的root， tree[i]存的是正常线段树的值， l[i]存的是i号节点的左儿子

int built_tree(int left, int right)
{
    int node = tot++;
    if (left < right) {
        L[node] = built_tree(left, mid);
        R[node] = built_tree(mid + 1, right);
    }
    return node;
}

int update(int pre, int left, int right, int x)
{
    int node = tot++;
    L[node] = L[pre];  // 必须要加
    R[node] = R[pre];
    tree[node] = tree[pre] + 1;
    sumt[node] = sumt[pre] + b[x];
    if (left < right) {
        if (x <= mid) L[node] = update(L[pre], left, mid, x);
        else R[node] = update(R[pre], mid + 1, right, x);
```

```
    }
    return node;
}

// 求l~r，第1大，2大，3大...k大的和
int query(int node1, int node2, int left, int right, int k)
{
    if (left == right) return b[left] * k;
    int rsum = tree[R[node2]] - tree[R[node1]];
    // 找第k大及其更大的和
    if (rsum >= k) {
        return query(R[node1], R[node2], mid + 1, right, k);
    }
    else {
        int sumr = sumt[R[node2]] - sumt[R[node1]];
        return sumr + query(L[node1], L[node2], left, mid, k -
rsum);
    }
}

signed main()
{
        tot = 0; // 主席树动态开点，从0开始
        memset(T, 0, sizeof(T)); memset(tree, 0, sizeof(tree));
        memset(L, 0, sizeof(L)); memset(R, 0, sizeof(R));
        int k, x;
        cin >> n >> q >> k >> x;
        for (int i = 1; i <= n; i++) {
            cin>>a[i]; b[i] = a[i];
        }
        sort(b + 1, b + 1 + n);
        m = unique(b + 1, b + 1 + n) - b - 1; // 离散化
        T[0] = built_tree(1, m);
        for (int i = 1; i <= n; i++) {
            a[i] = lower_bound(b + 1, b + 1 + m, a[i]) - b;
            T[i] = update(T[i - 1], 1, m, a[i]);
        }
        while (q--) {
            int l, r;
            cin >> l >> r;
            int ans = query(T[l - 1], T[r], 1, m, k);
            if (ans >= x)cout << "Y\n";
            else cout << "N\n";
        }
    return 0;
}
```

六、【树上差分】

https://ac.nowcoder.com/acm/contest/33191/B

```
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;
typedef long double ld;
typedef pair<ll, ll>P;
const ll maxn = 2e6 + 99;
const ll inf = 1e18 + 7;
const ll mod = 1e9 + 7;
const ld pi = 3.14159265358979323846;
ll t, n, k, m;
struct Edges {
    ll to, w, next; //（边的终点编号、权重、同起点的下一条边）
};
Edges edge[maxn*2];  //记录所有点的信息
ll head[maxn];      //例：head[1]=5;记录从1点出发的最后输入的一条边5
ll cnt;            //记录每组数据的编号
ll faver[maxn];
void init()        //初始化
{
    memset(head, -1, sizeof(head));
    cnt = 0;
}
void add(ll from, ll to, ll w) //添加边
{
    edge[cnt].to = to;
    edge[cnt].w = w;
    edge[cnt].next = head[from];
    head[from] = cnt++;
}
ll cf[maxn];
ll st[maxn];
ll deep;
void dfs(ll u, ll fa) {
    st[++deep] = u;
    cf[u] += 1;
    int p = faver[u]+1;
    if (p >= deep)p = deep;
    cf[st[deep - p]] -= 1;
    for (ll i = head[u]; i != -1; i = edge[i].next) {
        if (edge[i].to == fa)continue;
        dfs(edge[i].to, u);
        cf[u] += cf[edge[i].to];
    }
```

```
    deep--;
}
int main() {
    ios::sync_with_stdio(false); cin.tie(0); cout.tie(0);
    cin >> n;
    init();
    ll from, to;
    for (ll i = 0; i < n - 1; i++) {
        cin >> from >> to;
        add(from, to, 1);
        add(to, from, 1);
    }
    for (ll i = 1; i <= n; i++) cin >> faver[i];
    dfs(1, 0);
    for (ll i = 1; i <= n; i++) {
        cout << cf[i] << " \n"[i == n];
    }
    return 0;
}
```

【博弈】

一、【寻找SG值模板】

模板：

```
//前向星
//const int MAXN = 1e6; //数组大小初始化
//struct Edges {
//  int to,  next; //（边的终点编号、同起点的下一条边）
//};
//Edges edge[MAXN];  //记录所有点的信息
//ll head[MAXN];     //例：head[1]=5;记录从1点出发的最后输入的一条边5
//ll cnt;             //记录每组数据的编号
//void init()         //初始化
//{
//  memset(head, -1, sizeof(head));
//  cnt = 0;
//}
```

```
//void add(ll from, ll to) //添加边
//{
//  edge[cnt].to = to;
//  edge[cnt].next = head[from];
//  head[from] = cnt++;
//}
ll SG[MAXN];//用来存放每个点的SG值,初始化为-1
ll find_SG(int now)//now初始为图上的起点,递归中则为当前所在的节点
{
    if(SG[now]!=-1)return SG[now];  //已经找到改点SG值，则无需重复寻找，直接插入到set s中
    ll i;
    multiset<ll>s;                  //存放SG值的列表，可以插入完全相同的两条记录,会提高数据插入的速度
    for(i=head[now];i!=-1;i=edge[i].next)   //遍历以now节点开始的所有边（前向星式邻接表）
    {
        s.insert(find_SG(edge[i].to));
    }
    for(i=0;;i++)//遍历获得最小非负整数SG值
    {
        if(s.find(i)==s.end())break;//没找到i，说明i就是最小的，获得!
    }
    SG[now]=i;
    return SG[now];
}
```

【平面几何】

基础知识

点积:

1. $(x_0, y_0) \cdot (x_1, y_1) = x_0x_1 + y_0y_1$
2. $\overrightarrow{OA} \cdot \overrightarrow{OB} = |OA| * |OB| * COS < AOB$

夹角小于90°时，点积为正；夹角大于90°时，点积为负；互相垂直时，点积为0。

叉积:

$$(x_0,y_0)\times(x_1,y_1)=x_0y_1-x_1y_0$$

$$\overrightarrow{OA}\chi\overrightarrow{OB}=|OA|*|OB|*Sin<OAB=2\triangle OAB$$

OA到OB为顺时针，叉积为正；OA到OB为逆时针，叉积为负；叉积平行时，点积为0。

海伦公式：

$$\text{半周长}\quad p=1/2(a+b+c)$$
$$S_(\triangle ABC)=\sqrt{(p(p-a)(p-b)(p-c))}$$
$$\text{叉积：}\quad S_(\triangle ABC)=1/2|(AB)^{\rightarrow}\cdot(AC)^{\rightarrow}|$$

多边形面积：**

$$S=1/2|(OP_1)^{\rightarrow}\times(OP_2)^{\rightarrow}+(OP_2)^{\rightarrow}\times(OP_3)^{\rightarrow}+\ldots+(OP_(n-1))^{\rightarrow}\times(OP_n)^{\rightarrow}+(OP_n)^{\rightarrow}\times(OF$$

K - Triangle

Gym - 103466K

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

using namespace std;

const double pi = acos(-1.0);    // 高精度圆周率
const double eps = 1e-8;         // 偏差值，有时用1e-10

int sgn(double x) {               // 判断x是否等于0
    if (fabs(x) < eps)
        return 0;
    else
        return x < 0 ? -1 : 1;
}

int dcmp(double x, double y) {   // 比较浮点数
    if (fabs(x - y) < eps)
        return 0;
    else
        return x < y ? -1 : 1;
}

struct Point {
    double x, y;
```

```
    Point() {}
    Point(double x, double y) : x(x), y(y) {}
    Point operator+(Point B) {
        return Point(x + B.x, y + B.y);
    }
    Point operator-(Point B) {
        return Point(x - B.x, y - B.y);
    }
    Point operator*(double k) {
        return Point(x * k, y * k);
    }
    Point operator/(double k) {
        return Point(x / k, y / k);
    }
    bool operator==(Point B) {
        return sgn(x - B.x) == 0 && sgn(y - B.y) == 0;
    }
};

typedef Point Vector;

double Distance(Point A, Point B) {
    return hypot(A.x - B.x, A.y - B.y);
}

// Dot product
double Dot(Vector A, Vector B) {
    return A.x * B.x + A.y * B.y;
}

double Len(Vector A) {
    return sqrt(Dot(A, A));
}

double Len2(Vector A) {
    return Dot(A, A);
}

double Angle(Vector A, Vector B) {
    return acos(Dot(A, B) / Len(A) / Len(B));
}

// Cross product
// A X B = |A| |B| sin0
double Cross(Vector A, Vector B) {
    return A.x * B.y - A.y * B.x;
}
```

```
double Area2(Point A, Point B, Point C) {
    return Cross(B - A, C - A);
}

// 向量旋转---逆时针
Vector Roatate(Vector A, double rad) {
    return Vector(A.x * cos(rad) - A.y * sin(rad), A.x * sin(rad) +
A.y * cos(rad));
}

Vector Normal(Vector A) {
    return Vector(-A.y / Len(A), A.x / Len(A));
}

bool Parallel(Vector A, Vector B) {
    return sgn(Cross(A, B)) == 0;
}

struct Line {
    Point p1, p2;
    Line() {}
    Line(Point p1, Point p2) :p1(p1), p2(p2) {}
    Line(Point p, double angle) {
        p1 = p;
        if (sgn(angle - pi / 2) == 0) {
            p2 = (p1 + Point(0, 1));
        }
        else {
            p2 = (p1 + Point(1, tan(angle)));
        }
    }
    Line(double a, double b, double c) {
        if (sgn(a) == 0) {
            p1 = Point(0, -c / b);
            p2 = Point(1, -c / b);
        }
        else if (sgn(b) == 0) {
            p1 = Point(-c / a, 0);
            p2 = Point(-c / a, 1);
        }
        else {
            p1 = Point(0, -c / b);
            p2 = Point(1, (-c - a) / b);
        }
    }
};
```

```
typedef Line Segment;

bool Point_on_seg(Point p, Line v) {
    return sgn(Cross(p - v.p1, v.p2 - v.p1)) == 0 && sgn(Dot(p -
v.p1, p - v.p2)) <= 0;
}

int ttt;
double x1, y_1, x2, y2, x3, y3, px, py;
double a, b;

void chck() {
    bool on12 = Point_on_seg({ px, py }, Line{ {x1, y_1}, {x2, y2}
});
    bool on23 = Point_on_seg({ px, py }, Line{ {x2, y2}, {x3, y3}
});
    bool on31 = Point_on_seg({ px, py }, Line{ {x3, y3}, {x1, y_1}
});
    if (on12 + on23 + on31==0) {
        printf("-1\n");
        return;
    }
    if (on23) {
        std::swap(x1, x3);
        std::swap(y_1, y3);
    }
    else if (on31) {
        std::swap(x2, x3);
        std::swap(y2, y3);
    }
    double dpx1 = Distance({ px, py }, { x1, y_1 });
    double dpx2 = Distance({ px, py }, { x2, y2 });
    a = dpx1, b = dpx2;
    if (dcmp(a,b)==0) {
        // 对面顶点
        printf("%.12lf %.12lf\n", x3, y3);
        // cout << x3 << " " << y3 << "\n";
        return;
    }
    else if (dcmp(a,b)==-1)
        std::swap(a, b);
    if (dpx1 > dpx2) {
        std::swap(x1, x2);
        std::swap(y_1, y2);
    }
    double ansx = x2 + (x3 - x2) * (a + b) / a *0.5;
```

```
    double ansy = y2 + (y3 - y2) * (a + b) / a *0.5;
    printf("%.12lf %.12lf\n", ansx, ansy);
    return;
}

int main() {
    scanf("%d", &ttt);
    while (ttt--) {
        scanf("%lf%lf%lf%lf%lf%lf%lf%lf", &x1, &y_1, &x2, &y2, &x3,
&y3, &px, &py);
        chck();
    }
    return 0;
}
```

【精选好题】

【一、排队时间轴问题（**set+priority_queue**）】

2022“杭电杯”中国大学生算法设计超级联赛（5）1012

Buy Figurines

Time Limit (Java / Others)
12000 / 6000 MS

Memory Limit (Java / Others)
524288 / 262144 K

Ratio (Accepted / Submitted)
21.09% (669/3172)

Problem Description

During the "Hues of the Violet Garden" event, As the professional Lady Guuji hired, Sayu is assigned to buy one of the figurines, that is "Status of Her Excellency, the Almighty Narukami Ogosho, God of Thunder".

There are n people numbered from 1 to n who intent to buy a figurine and the store has m windows with m queues identified from 1 to m. The i-th person has an arrival time a_i and a spent time s_i to buy a figurine(It guaranteed that everyone's arrival time a_i is different). When a person arrives at the store, he will choose the queue with the least number of people to queue. If there are multiple queues with the least number of people, he will choose the queue with the smallest identifier. It should be noted that if someone leaves the queue at the same time, the person will choose the queue after everyone leaves the team.

Sayu has been here since last night so she could buy a figurine. But after waiting and waiting, her eyes started to feel real droopy and... overslept. If Sayu doesn't buy one of these figurines, the Tenryou Commission tengu will lock her up for life! The store will close after these n people buy figurines, that means she must wake up before the last one leaves. Now Lady Guuji wants to know the latest time Sayu wakes up.

For example, there are two people in the same line, $a_1 = 1, s_1 = 2, a_2 = 2, s_2 = 2$. When the first person arrives, there is no one in the line, so the start time and end time of purchasing the figurine are 1 and 3. When the second person arrives, the first person is still in line, so the start time and end time of purchasing the figurine are 3 and 5. And if the end time of the last person is x, the answer is x.

Input

The first line contains one integer T ($1 \le T \le 10$) .

The first line of each test case contains two positive integers n and m ($1 \le n \le 2 \times 10^5, 1 \le m \le 2 \times 10^5$) --- the number of people and the number of queues.

Then, n lines follow, each consisting of two integers a_i and s_i ($1 \le a_i, s_i \le 10^9$) --- the arrival time and spent time of i-th person.

It guaranteed that the sum of n does not exceed 2×10^6, and the sum of m does not exceed 2×10^6.

Output

For each test case:

print a line containing a single integer --- the latest time Sayu wakes up, that means the end time of the last person.

Sample Input

```
1
5 3
2 4
1 3
5 1
3 4
4 2
```

Sample Output

```
7
```

题解：

1012.Buy Figurines

考虑时刻 a_i 第 i 个人到达商店时，如何快速查询最短并且编号最小的队伍，可以用 set 维护队伍的信息，又考虑到每个人的开始时刻和结束时刻是不同的，可以用一个优先队列来维护时间轴的信息。先按照时间轴的顺序，模拟商店入队和出队的情况，进而求出每个人的购买雷电将军人偶的开始时间和结束时间，最后一个人的结束时间即为本题所求答案。

重点：优先队列维护时间轴信息（logn），set用来维护队伍信息（队伍的人数、下标），通过set可以在logn时间内查询到时间轴上离开的人所处的队伍，从而能够让其队伍人数减一。整体时间复杂度nlogn。

注意事项：

1、为了区分时间轴上人是离开还是进入，需要type变量来区别。

2、为了能够通过set快速查询所处队伍，需要一个队伍数组quenum来记录每个队伍的人数，每个人也需要一个idx变量来记录其所排的队伍下标，配合使用，快速得到该人所处队伍人数，结合队伍下标，即可通过set在logn的时间内查询。

3、需要一个lsat数组来记录每个队伍最后的时间，进行模拟。

AC代码：

```cpp
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;
typedef long double ld;
typedef pair<ll, ll>P;
const ll maxn = 1e6 + 7;
const ll inf = 1e18 + 7;
const ll mod = 1e9 + 7;
const ld pi = 3.14159265358979323846;
ll t, n, k,m,x;
struct node {
    ll hidx;
    ll queidx;
    ll ti;
    bool type;
    friend bool operator<(node a, node b) {
        if (a.ti == b.ti)return a.type > b.type;
        return a.ti > b.ti;
    }
};
struct que {
    ll num, idx;
    friend bool operator < (que a, que b) {
        if (a.num == b.num)return a.idx < b.idx;
        else return a.num < b.num;
    }
};
ll hh[maxn];
ll quenum[maxn];
ll last[maxn];
priority_queue<node>tline;
set<que>ss;
int main(){
    ios::sync_with_stdio(false); cin.tie(0); cout.tie(0);
    cin >> t;
    while (t--) {
        ss.clear();
        cin >> n >> m;
        ll l, h;
        node tt;
        for (ll i = 1; i <= n; i++) {
            cin >> l >> h;
            hh[i] = h;
            tt.ti = l; tt.type = 1; tt.hidx = i;
            tline.push(tt);
        }
        que qq;
```

```
        for (ll i = 1; i <=m; i++) {
            qq.idx = i; qq.num = 0;
            ss.insert(qq);
            quenum[i] = 0;
            last[i] = 0;
        }
        while (!tline.empty()) {
            node person = tline.top(); tline.pop();
            if (person.type == 1) {
                person.type = 0;
                auto qq = *ss.begin();
                if (qq.num == 0)last[qq.idx] =
person.ti+hh[person.hidx];
                else last[qq.idx] += hh[person.hidx];
                person.ti = last[qq.idx];
                ss.erase(qq);
                qq.num++;
                ss.insert(qq);
                quenum[qq.idx] = qq.num;
                person.queidx = qq.idx;
                tline.push(person);
            }
            else {
                que temp;
                temp.idx = person.queidx;
                temp.num = quenum[person.queidx];
                auto qq = *ss.find(temp);
                qq.num--;
                ss.insert(qq);
                quenum[person.queidx]--;
            }
        }
        ll ans = 0;
        for (ll i = 1; i <= n; i++) {
            ans = max(ans, last[i]);
        }
        cout << ans << "\n";
    }
    return 0;
}
```

【二、构造新图跑最短路（dijsktra）】

【HDU-7187】

Slipper

Time Limit: 10000/5000 MS (Java/Others) Memory Limit: 524288/262144 K (Java/Others)
Total Submission(s): 1269 Accepted Submission(s): 312

Problem Description

Gi is a naughty child. He often does some strange things. Therefore, his father decides to play a game with him.

Gi's father is a senior magician, he teleports Gi and Gi's Slipper into a labyrinth. To simplify this problem, we regard the labyrinth as a tree with n nodes, rooted at node 1. Gi is initially at node s, and his slipper is at node t. In the tree, going through any edge between two nodes costs w unit of power.

Gi is also a little magician! He can use his magic to teleport to any other node, if the depth difference between these two nodes equals to k. That is, if two nodes u, v satisfying that $|dep_u - dep_v| = k$, then Gi can teleport from u to v or from v to u. But each time when he uses magic he needs to consume p unit of power. Note that he can use his magic any times.

Gi want to take his slipper with minimum unit of power.

Input

Each test contains multiple test cases. The first line contains the number of test cases ($1 \le T \le 5$). Description of the test cases follows.

The first line contains an integer n --- The number of nodes in the tree. $2 \le n \le 10^6$.

The following $n-1$ lines contains 3 integers u, v, w that means there is an edge between nodes u and v. Going through this edge costs w unit of power.
$1 \le u, v \le n, 1 \le w \le 10^6$.

The next line will contain two separated integers k, p. $1 \le k \le \max_{u \subset V}(dep_u), 0 \le p \le 10^6$.

The last line contains two positive integers s, t, denoting the positions of Gi and slipper. $1 \le s \le n, 1 \le t \le n$. It is guaranteed the $s \ne t$.

Output

For each test case:

Print an integer in a line --- the minimum unit of power Gi needs.

Sample Input

```
1
6
6 1 2
3 5 2
2 4 6
5 2 2
5 6 20
3 8
6 5
```

Sample Output

```
12
```

Hint

Example1: Gi can go from node 6 to node 1 using 2 units of power. Then he teleports from node 1 to node 2 using 8 units of power. Finally, he goes from node 2 to node 5 using 2 units of power. $Total\ cost = 2 + 8 + 2 = 12$

标答：

1003.Slipper

设树的深度为d，在第i层($1 \le i \le d$)和第$i+1$层中新增两个点l_i, r_i，l_i连向所有第$i+1$层的点，r_i连向所有第i层的点，对于原来树中所有的点，向l_{dep_u+k-1}连一条权为p单向边，向r_{dep_u-k}连一条权值为p的单向边，在修改后的图中跑dijkstra求s到t的最短路即可，复杂度O(nlogn)。

做法：

每层建立虚点（虚点下标从n+1到n+1+maxdeep），向同层的所有点连一条单向边，然后每一层的所有点向非同层的深度距离为K的虚点连一条单向边，最后所有的虚点之间距离（深度）为K的连一条双向边，最后通过dijkstra求出start-end的最短距离。

【特殊数据】：

1
3
1 2 1000000
1 3 1000000
1 1
2 3

```
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;
typedef long double ld;
typedef pair<ll, ll>P;
const ll maxn = 1e6 + 7;
const ll inf = 1e18 + 7;
const ll mod = 1e9 + 7;
const ld pi = 3.14159265358979323846;
ll t, n, k, p,st,en,maxdeep;
string str;
struct Edges {
    ll to, w, next;
};
Edges edge[maxn << 2];
ll head[maxn << 2];
ll cnt;
ll dis[maxn << 2];
ll deep[maxn << 2];
bool vis[maxn<<2];
void init() {
    for (ll i = 0; i < maxn<<2; i++) {
        head[i] = -1;
        dis[i] = inf;
        deep[i] = -1;
        vis[i] = 0;
    }
    cnt = 0;
    maxdeep = -1;
}
void add(ll from, ll to, ll w) {
    edge[cnt].to = to;
    edge[cnt].w = w;
    edge[cnt].next = head[from];
```

```
    head[from] = cnt++;
}
void dfs(ll u, ll fa) {
    deep[u] = deep[fa] + 1;
    maxdeep = max(maxdeep, deep[u]);
    for (ll i = head[u]; i != -1; i = edge[i].next) {
        if (edge[i].to == fa)continue;
        dfs(edge[i].to, u);
    }
}
void dijkstra() {
    priority_queue<P, vector<P>, greater<P>>qq;
    dis[st] = 0;
    qq.push(make_pair(dis[st], st));
    P temp;
    while (!qq.empty()) {
        temp = qq.top();
        qq.pop();
        ll u = temp.second;
        if (vis[u])continue;
        vis[u] = 1;
        for (ll i = head[u]; i != -1; i = edge[i].next) {
            ll to = edge[i].to;
            if (!vis[to] && dis[to] > dis[u] + edge[i].w) {
                dis[to] = dis[u] + edge[i].w;
                qq.push(P(dis[to],to));
            }
        }
    }
}
void solve() {
    ios::sync_with_stdio(false); cin.tie(0); cout.tie(0);
    init();
    cin >> n;
    ll from, to, w;
    for (ll i = 0; i < n - 1; i++) {
        cin >> from >> to >> w;
        add(from, to, w);
        add(to, from, w);
    }
    deep[0] = -1;
    dfs(1, 0);
    cin >> k >> p;
    for (ll i = 1; i <= n; i++) {
        add(n + deep[i] + 1, i, 0);
        if (deep[i] + k <= maxdeep) {
            add(i, n + 1 + deep[i] + k, p);
```

```
        }
        if (deep[i] - k >= 0) {
            add(i, n + 1 + deep[i] - k, p);
        }
    }

    for (ll i = n + 1; i+k <= n + maxdeep + 1; i++) {
        add(i, i + k, p);
        add(i + k, i, p);
    }
    cin >> st >> en;
    dijkstra();
    cout << dis[en] << "\n";
}
int main(){
    ios::sync_with_stdio(false); cin.tie(0); cout.tie(0);
    cin >> t;
    while (t--) {
        solve();
    }
    return 0;
}
```

【杂项】

一、【高精度】

加法：

传入参数约定：传入参数均为string类型，返回值为string类型

算法思想：倒置相加再还原。

算法复杂度：o(n)

```
string add(string a,string b)//只限两个非负整数相加
{
    const int L=1e5;
    string ans;
```

```
    int na[L]={0},nb[L]={0};
    int la=a.size(),lb=b.size();
    for(int i=0;i<la;i++) na[la-1-i]=a[i]-'0';
    for(int i=0;i<lb;i++) nb[lb-1-i]=b[i]-'0';
    int lmax=la>lb?la:lb;
    for(int i=0;i<lmax;i++)
na[i]+=nb[i],na[i+1]+=na[i]/10,na[i]%=10;
    if(na[lmax]) lmax++;
    for(int i=lmax-1;i>=0;i--) ans+=na[i]+'0';
    return ans;
}
```

减法：

传入参数约定：传入参数均为string类型，返回值为string类型

算法思想：倒置相减再还原。

算法复杂度：o(n)

```
string sub(string a,string b)//只限大的非负整数减小的非负整数
{
    const int L=1e5;
    string ans;
    int na[L]={0},nb[L]={0};
    int la=a.size(),lb=b.size();
    for(int i=0;i<la;i++) na[la-1-i]=a[i]-'0';
    for(int i=0;i<lb;i++) nb[lb-1-i]=b[i]-'0';
    int lmax=la>lb?la:lb;
    for(int i=0;i<lmax;i++)
    {
        na[i]-=nb[i];
        if(na[i]<0) na[i]+=10,na[i+1]--;
    }
    while(!na[--lmax]&&lmax>0)  ;lmax++;
    for(int i=lmax-1;i>=0;i--) ans+=na[i]+'0';
    return ans;
}
```

乘法：

传入参数约定：传入参数均为string类型，返回值为string类型

算法思想：倒置相乘，然后统一处理进位，再还原。

算法复杂度：o(n^2)

```
string mul(string a,string b)//高精度乘法a,b,均为非负整数
{
    const int L=1e5;
    string s;
    int na[L],nb[L],nc[L],La=a.size(),Lb=b.size();//na存储被乘数，nb存储乘数，nc存储积
    fill(na,na+L,0);fill(nb,nb+L,0);fill(nc,nc+L,0);//将na,nb,nc都置为0
    for(int i=La-1;i>=0;i--) na[La-i]=a[i]-'0';//将字符串表示的大整形数转成i整形数组表示的大整形数
    for(int i=Lb-1;i>=0;i--) nb[Lb-i]=b[i]-'0';
    for(int i=1;i<=La;i++)
        for(int j=1;j<=Lb;j++)
        nc[i+j-1]+=na[i]*nb[j];//a的第i位乘以b的第j位为积的第i+j-1位（先不考虑进位）
    for(int i=1;i<=La+Lb;i++)
        nc[i+1]+=nc[i]/10,nc[i]%=10;//统一处理进位
    if(nc[La+Lb]) s+=nc[La+Lb]+'0';//判断第i+j位上的数字是不是0
    for(int i=La+Lb-1;i>=1;i--)
        s+=nc[i]+'0';//将整形数组转成字符串
    return s;
}
```

除法：(待看)

传入参数约定：传入第一第二个参数均为string类型，第三个为int型，返回值为string类型

算法思想：倒置，试商，高精度减法。

算法复杂度：o(n^2)

```
int sub(int *a,int *b,int La,int Lb)
{
    if(La<Lb) return -1;//如果a小于b，则返回-1
    if(La==Lb)
    {
        for(int i=La-1;i>=0;i--)
            if(a[i]>b[i]) break;
            else if(a[i]<b[i]) return -1;//如果a小于b，则返回-1

    }
    for(int i=0;i<La;i++)//高精度减法
    {
        a[i]-=b[i];
```

```
        if(a[i]<0) a[i]+=10,a[i+1]--;
    }
    for(int i=La-1;i>=0;i--)
        if(a[i]) return i+1;//返回差的位数
    return 0;//返回差的位数

}
string div(string n1,string n2,int nn)
//n1,n2是字符串表示的被除数，除数,nn是选择返回商还是余数
{
    const int L=1e5;
    string s,v;//s存商,v存余数
     int a[L],b[L],r[L],La=n1.size(),Lb=n2.size(),i,tp=La;
     //a，b是整形数组表示被除数，除数，tp保存被除数的长度
     fill(a,a+L,0);fill(b,b+L,0);fill(r,r+L,0);//数组元素都置为0
     for(i=La-1;i>=0;i--) a[La-1-i]=n1[i]-'0';
     for(i=Lb-1;i>=0;i--) b[Lb-1-i]=n2[i]-'0';
     if(La<Lb || (La==Lb && n1<n2)) {
            //cout<<0<<endl;
     return n1;}//如果a<b,则商为0，余数为被除数
     int t=La-Lb;//除被数和除数的位数之差
     for(int i=La-1;i>=0;i--)//将除数扩大10^t倍
        if(i>=t) b[i]=b[i-t];
        else b[i]=0;
     Lb=La;
     for(int j=0;j<=t;j++)
     {
         int temp;
         while((temp=sub(a,b+j,La,Lb-j))>=0)//如果被除数比除数大继续减
         {
             La=temp;
             r[t-j]++;
         }
     }
     for(i=0;i<L-10;i++) r[i+1]+=r[i]/10,r[i]%=10;//统一处理进位
     while(!r[i]) i--;//将整形数组表示的商转化成字符串表示的
     while(i>=0) s+=r[i--]+'0';
     //cout<<s<<endl;
     i=tp;
     while(!a[i]) i--;//将整形数组表示的余数转化成字符串表示的</span>
     while(i>=0) v+=a[i--]+'0';
     if(v.empty()) v="0";
     //cout<<v<<endl;
     if(nn==1) return s;//返回商
     if(nn==2) return v;//返回余数
}
```

【莫队】复杂度(O(N√N))

算法过程：

1. 对于多段区间的询问,先将询问离线存储下来,然后再从左到右扫一遍,在过程中维护一段区间,就可以得到每个寻问的答案.
2. 但暴力扫肯定不行，所以在扫的过程中,需要对 l 进行排序,以求能够在移动次数最少的情况下,得到所有希望求出的区间.
3. 首先对每个区间进行分块操作,再将左端点在一起的区间询问放在一起进行处理,对于每个块处理一遍,那么就可以得到所有询问的答案.

以左指针作为第一关键序，右指针为第二关键序

将整个区间划分为【根号n】个块，使得左指针一共有根号n种可能，对于同一左指针位于同一块内的区间来说，右指针最差情况为O(n),因此整体复杂度为O(n*n^1/2)

block为sqrt(n)

但block=n/sqrt(m*2/3) 会比上一个更快一点，差不多是原来的0.9倍

```
int block=sqrt(n);
const int maxn=1e5+7;
struct node{
  int l,r,id;
}q[maxn];
//正常排序
bool cmp(node a,node b){
    return (a.l/block)==(b.l/block)?a.r<b.r:a.l<b.l;
}
//奇偶性排序
bool cmp(node a,node b){
    return (a.l/block)^(b.l/block)?a.l<b.l:(((a.l/block)&1)?
a.r<b.r:a.r>b.r);
}
void add(int x){
    if(!cnt[a[x]])ans++;
    cnt[a[x]++;
}
void del(int x){
    cnt[a[x]]--;
    if(!cnt[a[x]])ans--;
}
sort(q+1,q+1+m,cmp);
```

例1

【洛谷P1972 [SDOI2009]HH的项链】

```
#include<iostream>
#include<stdio.h>
#include<cmath>
#include<algorithm>
#include<vector>
#include<queue>
#include<map>
#include<set>
#include<stack>
#define put putchar('\n')
#define re register
using namespace std;
typedef long long ll;
typedef long double ld;
const int maxn = 1e6 + 10;
int arr[maxn], cnt[maxn], res[maxn];
int ans;
inline int read() {
    char c = getchar(); int tot = 1; while ((c < '0' || c>'9') && c
!= '-') c = getchar(); if (c == '-') { tot = -1; c = getchar(); }
    int sum = 0; while (c >= '0' && c <= '9') { sum = sum * 10 + c
- '0'; c = getchar(); }return sum * tot;
}
inline void wr(int x) { if (x < 0) { putchar('-'); wr(-x); return;
}if (x >= 10)wr(x / 10); putchar(x % 10 + '0'); }
inline void wrn(int x) { wr(x); put; }
inline void wri(int x) { wr(x); putchar(' '); }
struct node {
    int l, r, id;
}q[maxn];
int block;
bool cmp(node a, node b) {
    return (a.l / block) ^ (b.l / block) ? a.l < b.l : (((a.l /
block) & 1) ? a.r<b.r : a.r>b.r);
}
void add(int x) {
    ans += (++cnt[arr[x]] == 1);
}
void del(int x) {
    ans -= (--cnt[arr[x]] == 0);
}

int main() {
```

```
    ios::sync_with_stdio(false); cin.tie(0);
    int n, t;
    n = read();
    for (int i = 1; i <= n; i++)arr[i] = read();
    t = read();
    block =sqrt(n);
    for (int i = 1; i <= t; i++) {
        q[i].l = read();
        q[i].r = read();
        q[i].id = i;
    }
    sort(q + 1, q + 1 + t, cmp);
    int l = 1, r = 0;           //该题r必为0，若为1，第一个挤不进去，l 0和1皆可，但1更好
    for (int i = 1; i <= t; ++i) {
        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--);
        res[q[i].id] = ans;
    }
    for (int i = 1; i <= t; ++i)wrn(res[i]);
    return 0;
}
```

例2

P2709 小B的询问

```
#include<iostream>
#include<stdio.h>
#include<cmath>
#include<algorithm>
#include<vector>
#include<queue>
#include<map>
#include<set>
#include<stack>
#define put putchar('\n')
#define re register
using namespace std;
typedef long long ll;
typedef long double ld;
const int maxn = 1e6 + 10;
int arr[maxn], cnt[maxn], res[maxn];
int ans;
```

```cpp
inline int read() {
    char c = getchar(); int tot = 1; while ((c < '0' || c>'9') && c
!= '-') c = getchar(); if (c == '-') { tot = -1; c = getchar(); }
    int sum = 0; while (c >= '0' && c <= '9') { sum = sum * 10 + c
- '0'; c = getchar(); }return sum * tot;
}
inline void wr(int x) { if (x < 0) { putchar('-'); wr(-x); return;
}if (x >= 10)wr(x / 10); putchar(x % 10 + '0'); }
inline void wrn(int x) { wr(x); put; }
inline void wri(int x) { wr(x); putchar(' '); }
struct node {
    int l, r, id;
}q[maxn];
int block;
bool cmp(node a, node b) {
    return (a.l / block) ^ (b.l / block) ? a.l < b.l : (((a.l /
block) & 1) ? a.r<b.r : a.r>b.r);
}

int main() {
    ios::sync_with_stdio(false); cin.tie(0);
    int n, t, k;
    n = read(); t = read(); k = read();
    for (int i = 1; i <= n; i++)arr[i] = read();
    block =sqrt(n);
    for (int i = 1; i <= t; i++) {
        q[i].l = read();
        q[i].r = read();
        q[i].id = i;
    }
    sort(q + 1, q + 1 + t, cmp);
    int l = 1, r = 0;     //l初始要为1,若为0，当第一个为1-4区间时，l必定小于
q[i].l，会导致sum=1，多加了一个1.同理r初始只能为0，若为1，则可能第一个加不进
去。
    int sum = 0;
    for (int i = 1; i <= t; i++) {
        while (l < q[i].l)cnt[arr[l]]--,sum-=(2*cnt[arr[l]]+1),l++;
        while (l > q[i].l)l--,cnt[arr[l]]++,sum+=(2*cnt[arr[l]]-1);
        while (r < q[i].r)r++, cnt[arr[r]]++ ,sum += (2 *
cnt[arr[r]] -1);
        while (r > q[i].r)cnt[arr[r]]--, sum -= (2 * cnt[arr[r]] +
1), r--;
        res[q[i].id] = sum;
    }
    for (int i = 1; i <= t; i++)cout << res[i] << endl;
    return 0;
}
```

```cpp
#include<iostream>
#include<fstream>
#include<algorithm>
#include<string>
using namespace std;
const int maxn = 2e3 + 7;            //开两倍空间
typedef long long ll;
int fa[maxn];
void init() {
    for (int i = 0; i < maxn; i++) fa[i] = i;
}
//并查集
int find_set(int x)
{
    if (x != fa[x])return fa[x] = find_set(fa[x]);
    return x;
}
void merge(int x, int y) {
    x = find_set(x);
    y = find_set(y);
    if (x == y) return;
    fa[x] = y;
}

int main() {
    ifstream fin;
    ofstream fout;
    fin.open("virus.in",ios::in);
    fout.open("virus.out", ios::out);
    int n, m,p,q,num=5;
    char op;
    cout << "【一共有5组测试样例】" << "\n\n";

    while (num--) {
        cout << "【测试样例" << 5 - num  << "】:" << endl;
        init();
        fin >> n;
        fin >> m;
        cout << n << endl;
        cout<< m << endl;
        while (m--) {
            fin >> op >> p >> q;
            cout << op << " " <<p<< " "<<q<< endl;
            if (op == 'S') {
//合并同类病毒
                merge(p, q);
            }
```

```
            else if (op == 'H') {
//敌对病毒并查集反集
                merge(p + n, q);
                merge(q + n, p);
            }
        }
        int cnt = 0;
        for (int i = 1; i <= n; i++) {
            //cout << i << " " << fa[i] << endl;
            if (fa[i] == i)cnt++;
        }
        fout <<"【测试样例"<<5-num<<"】:  " << cnt << endl;
        cout << "【测试样例"<<5-num<<"答案】 : " << cnt << "\n\n";
    }
    fin.close();
    fout.close();
    cout << "已完成!" << endl;
    return 0;
}
```

```
#include<bits/stdc++.h>
#include<iostream>
using namespace std;
const int maxn = 1e6 + 7;
int n;
int list1[maxn];
int list2[maxn];
void solve1() {
    int a, b;
    a = b = 1;
    int cnt = 0;
    while (cnt != n) {
        if (list1[a] < list2[b]) {
            cnt++;
            if (cnt == n) {
                cout << list1[a] << endl; break;
```

```
            }
            a++;
        }
        else {
            cnt++;
            if (cnt == n) {
                cout << list2[b] << endl; break;
            }
            b++;
        }
    }
    cout << "初始的n:" << n << endl;
    cout << "执行总次数: " << cnt << endl;
}
void solve2() {
    int k =n/2;
    int kres=n;
    int ldel,rdel,lnow,rnow;
    lnow=rnow =k;
    ldel = rdel = 0;
    int cnt = 0;
    while (true) {
        cnt++;
        if (kres == 1) {
            cout << min(list1[++lnow], list2[++rnow]) << endl;
            break;
        }
        if (list1[lnow] < list2[rnow]) ldel += k;
        else rdel += k;
        kres=kres-k;
        k = kres / 2;
        lnow = ldel + k;
        rnow = rdel + k;
        //cout<<list1[lnow] << " " << list2[rnow] << endl;
        //cout << lnow << " " << rnow << endl;

    }
    cout << "初始的n:" << n << endl;
    cout << "执行总次数: " << cnt << endl;
}
int main() {
    cin >> n;
    for (int i = 1; i <= n; i++) cin >> list1[i];
    for (int i = 1; i <= n; i++) cin >> list2[i];
    cout << "第一种方法（O(N)）: " << endl;
    solve1();//时间复杂度O(n)
    cout << "第二种方法（O(logn)）: " << endl;
```

```
    solve2();//时间复杂度为O(logn)
    return 0;
}
```