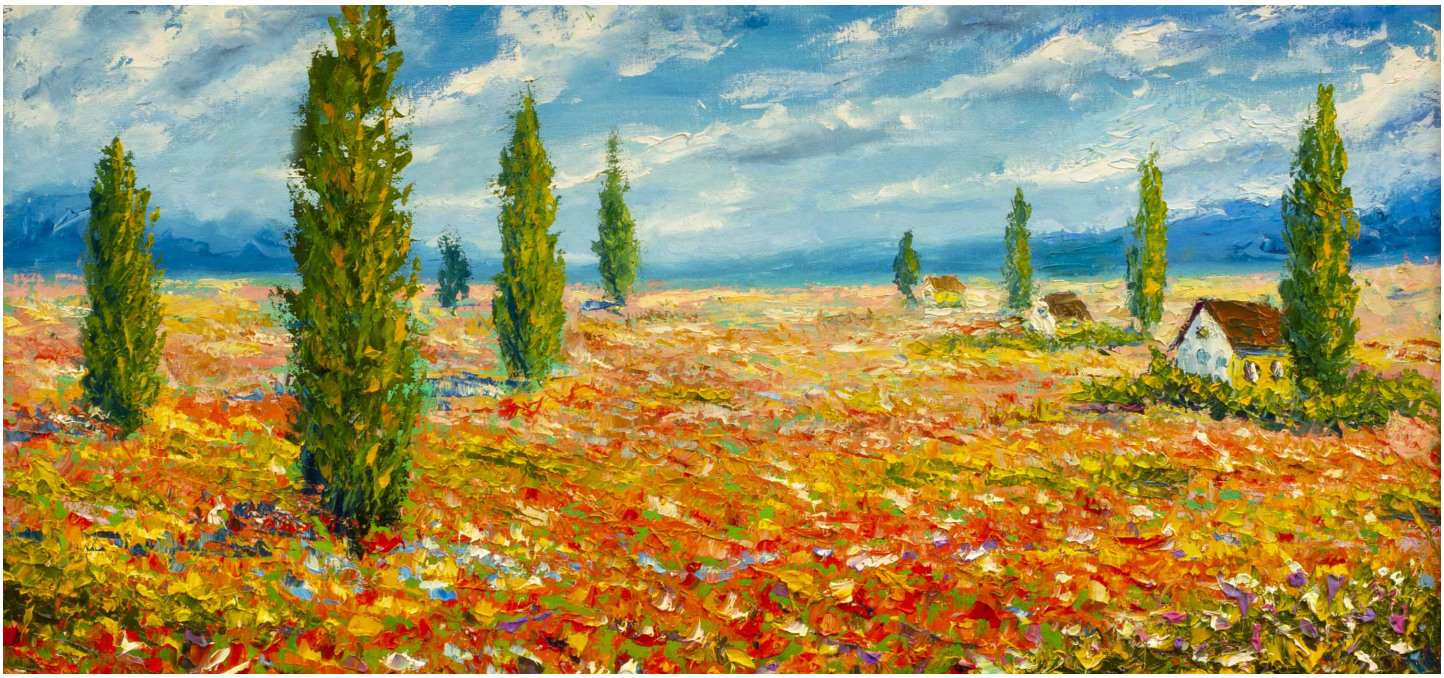


我的code模板

动态规划

最长上升子序列（单调队列优化）（nlogn）

```cpp
#include<bits/stdc++.h>
using namespace std;
typedef long long int ll;
typedef unsigned long long int Ull;
const ll N = 1e5+10;
int arr[N];
int main(){
    //ios_base::sync_with_stdio(0),cin.tie(0),cout.tie(0);
    //用单调队列的思想，V[i]的位置存长度为i的递增序列最后一个数的最小值
    //可知，V 是递增的。
    //遍历原数组，若arr[i]大于队尾元素则入队列（说明当前维护的最长序列长度可以加一），
    //否则就用arr[i]替换V中第一个大于等于arr[i]的值（若该值是V[j],说明V[j-1]严格小于arr[i]，V[j]大于等于arr[i],
    //则j长度的递增序列的最后一个数更新为arr[i]）
    //lower_bound用二分的方法从数组的begin位置到end-1位置二分查找第一个大于或等于num的数字，找到返回该数字的地址，不存在则返回end
    int n;
    cin>>n;
    for(int i = 1; i <= n; i++) cin>>arr[i];
    vector<int> V;
```

```
    for(int i = 1; i <= n; i++){
        if(V.size() == 0) V.push_back(arr[i]);
        else{
            if(arr[i] > V.back()) V.push_back(arr[i]);
            else *lower_bound(V.begin(),V.end(),arr[i]) = arr[i];
        }
    }
    cout<<V.size()<<endl;
}
```

最长公共子序列和最长上升字串（LCS）

用二维数组c[i][j]记录串与的LCS长度，则可得到状态转移方程

$$c[i,j]=\begin{cases}0 & \text{if } i=0 \text{ or } j=0,\\ c[i-1,j-1]+1 & \text{if } i,j>0 \text{ and } x_i=y_j,\\ \max(c[i,j-1],c[i-1,j]) & \text{if } i,j>0 \text{ and } x_i\neq y_j.\end{cases}$$

稍作修改，即可返回最长公共子序列

i和j分别从m，n开始，递减循环直到i = 0，j = 0。其中，m和n分别为两个串的长度。

- 如果str1[i] == str2[j]，则将str[i]字符插入到子序列内，i--，j--；
- 如果str1[i] != str[j]，则比较L[i,j-1]与L[i-1,j]，L[i,j-1]大，则j--，否则i--；（如果相等，则任选一个）

```
// 动态规划求解并输出所有LCS
#include <iostream>
#include <string>
#include <vector>
#include <set>
#include <algorithm>
using namespace std;

string X = "ABCBDAB";
string Y = "BDCABA";
vector<vector<int>> table; // 动态规划表
set<string> setOfLCS;      // set保存所有的LCS

int max(int a, int b)
{
    return (a > b) ? a : b;
}

/**
 * 构造表，并返回X和Y的LCS的长度
 */
int lcs(int m, int n)
```

```
{
    // 表的大小为(m+1)*(n+1)
    table = vector<vector<int>>(m + 1, vector<int>(n + 1));

    for (int i = 0; i < m + 1; ++i)
    {
        for (int j = 0; j < n + 1; ++j)
        {
            // 第一行和第一列置0
            if (i == 0 || j == 0)
                table[i][j] = 0;
            else if (X[i - 1] == Y[j - 1])
                table[i][j] = table[i - 1][j - 1] + 1;
            else
                table[i][j] = max(table[i - 1][j], table[i][j - 1]);
        }
    }

    return table[m][n];
}

/**
 * 求出所有的最长公共子序列，并放入set中
 */
void traceBack(int i, int j, string lcs_str, int lcs_len)
{
    while (i > 0 && j > 0)
    {
        if (X[i - 1] == Y[j - 1])
        {
            lcs_str.push_back(X[i - 1]);
            --i;
            --j;
        }
        else
        {
            if (table[i - 1][j] > table[i][j - 1])
                --i;
            else if (table[i - 1][j] < table[i][j - 1])
                --j;
            else // 相等的情况
            {
                traceBack(i - 1, j, lcs_str, lcs_len);
                traceBack(i, j - 1, lcs_str, lcs_len);
                return;
            }
        }
```

```
    }

    string str(lcs_str.rbegin(), lcs_str.rend()); // lcs_str逆序
    if ((int)str.size() == lcs_len)               // 判断str的长度是否等于lcs_len
        setOfLCS.insert(str);
}

void print()
{
    set<string>::iterator beg = setOfLCS.begin();
    for (; beg != setOfLCS.end(); ++beg)
        cout << *beg << endl;
}

int main()
{
    int m = X.length();
    int n = Y.length();
    int length = lcs(m, n);
    cout << "The length of LCS is " << length << endl;

    string str;
    traceBack(m, n, str, length);
    print();

    getchar();
    return 0;
}
```

$$c[i][j]\begin{cases} 0 & if\ i=0\ or\ j=0 \\ c[i-1][j-1]+1 & if\ i,j>0\ and\ xi=yi \\ 0 & if\ i,j>0\ and\ xi\neq yi \end{cases}$$

最长公共子串的长度为

稍微修改一下，即可返回最长公共子串。endPosition记录公共子串的结尾字符位置。subStrlength记录公共子串的长度。

```
public static int lcs(String str1, String str2) {
    int len1 = str1.length();
    int len2 = str2.length();
    int result = 0;     //记录最长公共子串长度
```

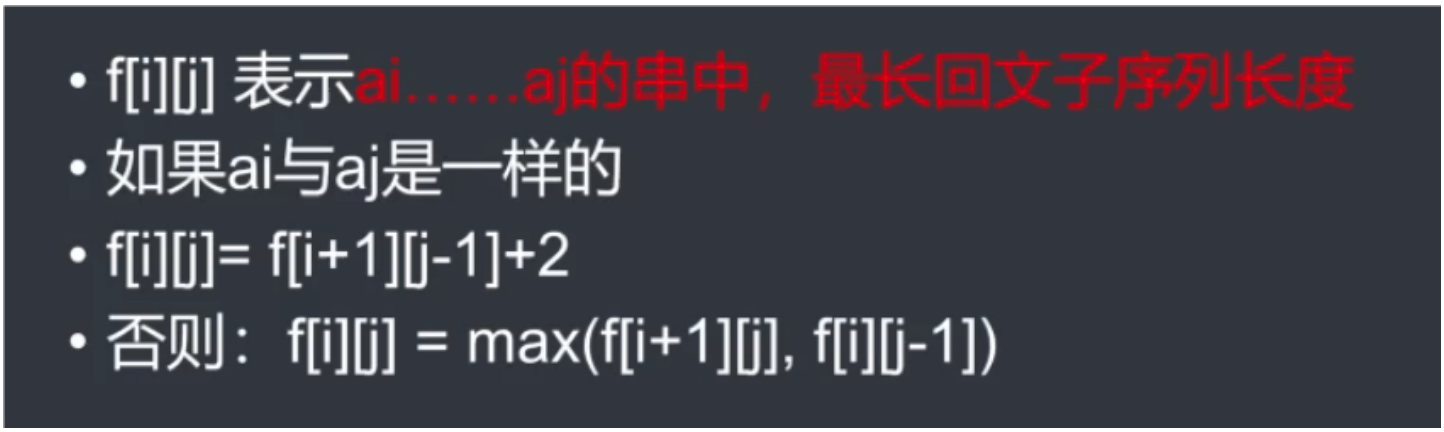
```
    int c[][] = new int[len1+1][len2+1];
    for (int i = 0; i <= len1; i++) {
        for( int j = 0; j <= len2; j++) {
            if(i == 0 || j == 0) {
                c[i][j] = 0;
            } else if (str1.charAt(i-1) == str2.charAt(j-1)) {
                c[i][j] = c[i-1][j-1] + 1;
                result = max(c[i][j], result);
            } else {
                c[i][j] = 0;
            }
        }
    }
    return result;//返回公共子串长度
}
//稍微修改一下，即可返回最长公共子串。
//endPosition记录公共子串的结尾字符位置。subStrlength记录公共子串的长度。
public static String lcs(String str1, String str2) {
            int len1 = str1.length();
            int len2 = str2.length();
            int subStrlength=0;
            int endPosition=0;
            int c[][] = new int[len1+1][len2+1];
            for (int i = 0; i <= len1; i++) {
                for( int j = 0; j <= len2; j++) {
                    if(i == 0 || j == 0) {
                        c[i][j] = 0;
                    } else if (str1.charAt(i-1) == str2.charAt(j-1)) {
                        c[i][j] = c[i-1][j-1] + 1;
                        if(subStrlength<c[i][j])
                        {
                                subStrlength=c[i][j]; //更新公共子串的最大长度
                                endPosition=i; //记录公共子串结尾字符的位置
                        }
                    } else {
                        c[i][j] = 0;
                    }
                }
            }
            return str1.substring(endPosition-subStrlength, endPosition); //返回公共子串
//=====================================================================
// 动态规划求解并输出所有LCS
#include <iostream>
#include <string>
#include <vector>
#include <set>
```

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

string x = "ABCBDAB";
string y = "BDCABA";
vector<vector<int>> table; // 动态规划表
set<string> setOflcs;      // set保存所有的LCS

/**
 * 构造表，并返回X和Y的LCS的长度
 */
int lcs(int m, int n)
{
    int biggest = 0;
    // 表的大小为(m+1)*(n+1)
    table = vector<vector<int>>(m+1, vector<int>(n+1));
    for(int i = 0; i < m+1; i++)
    {
        for(int j = 0; j < n+1; j++)
        {
            // 第一行和第一列置0
            if(i == 0 || j == 0)
                table[i][j] = 0;
            else if(x[i-1] == y[j-1])
            {
        table[i][j] = table[i-1][j-1] + 1;
        if(table[i][j] > biggest)
                    biggest = table[i][j]; // 存放LCS的长度
        }
        else
        table[i][j] = 0;
        }
    }
    return biggest;
}

/**
 * 求出所有的最长公共子串，并放入set中
 */
void traceBack(int m, int n, int lcs_len)
{
    string strOflcs;
    for(int i = 1; i < m+1; i++)
    {
        for(int j = 1; j < n+1; j++)
        {
            // 查到等于lcs_len的值，取字符
```

```
98              if(table[i][j] == lcs_len)
99              {
100                 int ii = i, jj = j;
101                 while(table[ii][jj] >= 1)
102                 {
103                     strOflcs.push_back(x[ii-1]);
104                     ii--;
105                     jj--;
106                 }
107                 string str(strOflcs.rbegin(), strOflcs.rend()); // strOflcs逆序
108                 if((int)str.size() == lcs_len)                  // 判断str
    的长度是否等于lcs_len
109                 {
110                     setOflcs.insert(str);
111                     strOflcs.clear();                           // 清空strOflcs
112                 }
113             }
114         }
115     }
116 }
117
118 // 输出set
119 void print()
120 {
121     set<string>::iterator iter = setOflcs.begin();
122     for(; iter != setOflcs.end(); iter++)
123         cout << *iter << endl;
124 }
125
126 int main()
127 {
128     int m = x.length();
129     int n = y.length();
130     int res = lcs(m, n);
131     cout << "res = " << res << endl;
132
133     traceBack(m, n, res);
134     print();
135
136     getchar();
137     return 0;
138 }
```

最长回文子序列（直接reverse一个 然后两个串跑一边最长公共总序列）

或者



最长回文子串（马拉车算法on 或者动态规划on2 ）

```
//================================马拉车算法================================
#include <iostream>
#include <cstdio>
#include <cstring>
#define Min(a, b) a > b ? b : a
#define Max(a, b) a > b ? a : b
using namespace std;
int Len[3000005];
char str[3000005], s[3000005];
int n, mx, id, len;

void init()
{
    memset(str, 0, sizeof(str));
    int k = 0;
    str[k++] = '$';
    for (int i = 0; i < len; i++)
    {
        str[k++] = '#';
        str[k++] = s[i];
    }
    str[k++] = '#';
    len = k;
}

int Manacher()
{
    Len[0] = 0;
    int sum = 0;
    mx = 0;
    for (int i = 1; i < len; i++)
    {
        if (i < mx)
```

```
            Len[i] = Min(mx - i, Len[2 * id - i]);
        else
            Len[i] = 1;
        while (str[i - Len[i]] == str[i + Len[i]])
            Len[i]++;
        if (Len[i] + i > mx)
        {
            mx = Len[i] + i;
            id = i;
            sum = Max(sum, Len[i]);
        }
    }
    return (sum - 1);
}

int main()
{
    scanf("%d", &n);
    while (n--)
    {
        scanf("%s", s);
        len = strlen(s);
        init();
        int temp = Manacher();
        printf("%d\n", temp);
    }
    return 0;
}
//=====================================================
import java.util.*;
//法1：动态规划 时间复杂度O(n^2),空间复杂度O(n^2)
public class Palindrome {
    public int getLongestPalindrome(String A, int n) {
        // write code here
        int[][] dp=new int[n][n];
        //dp[i][j]表示位置从i到j的元素，若为回文串则存其长度，若不为回文串则为0;
        for(int i=0;i<n;i++){
            dp[i][i]=1;
        }
        int max=1;
        char[]a=A.toCharArray();
        for(int len=2;len<=n;len++){
            //len表示其子串的长度，由于长度为1的子串必为回文串，所以len从2开始到其最大子串及它本身n
            for(int i=0;i<=n-len;i++){
                //i表示子串的开始位置，j表示子串结束的位置
                int j=i+len-1;
```

```
                //下面开始比较
                if(len==2&&a[i]==a[j]){
                    dp[i][j]=2;
                    max=2;
                }
                 else if(a[i]==a[j]&&dp[i+1][j-1]!=0){
                    dp[i][j]=len;
                    max=len;
                }
            }
        }
        return max;

    }
}
```

数据结构

字典树

相较于hash的区别，字典树可以方便查询是否出现过前缀

如果是查询某个字符串是否存在，可以另开一个 `exist` 数组，在插入完成时，把 `exist[叶子节点]` 设置为 `true` ，然后先按查询前缀的方法查询，在结尾处再判断一下 `exist` 的值。 这是一种常见的套路，即用叶子节点代表整个字符串，保存某些信息。

字典树是一种空间换时间的数据结构，我们牺牲了字符串个数×字符串平均字符数×字符集大小的空间，但可以用 $O(n)$ 的时间查询，其中 n 为查询的前缀或字符串的长度。

01字典树（用于解决一些异或问题）

st表（结构体封装）

```
const LL N = 5e05+10;
int Max[N][21];
int Min[N][21];

struct ST{
        void init(){
                for(int i = 1 ; i <= n ; i ++){
```

```
                        Max[i][0] = a[i];
                        Min[i][0] = a[i];
                }
        }
        void work(){
            for(int j = 1;j <= 21;j++)
                for(int i = 1;i + (1 << j) - 1 <= n ; i++){
                        Max[i][j] = max(Max[i][j - 1] , Max[i + (1 << (j - 1))][j - 1]);
                    Min[i][j] = min(Min[i][j - 1] , Min[i + (1 << (j - 1))][j - 1]);
                }
        }
        int QueryMax(int l,int r)
        {
            int k = log2(r-l+1);
            return max(Max[l][k],Max[r-(1<<k)+1][k]);//把拆出来的区间分别取最值
        }
        int QueryMin(int l , int r){
            int k = log2(r-l+1);
            return min(Min[l][k],Min[r-(1<<k)+1][k]);//把拆出来的区间分别取最值

        }
};

-------
ST stb;
stb.init();
stb.work();
//----------------------二维ST表-----------------------------------------
const int MAXN = 500; // 假设最大的尺寸
const int K = 9; // 2^K 是最大的区间长度，需要根据实际问题调整

int logTable[MAXN+1]; // 预处理对数表
int st[MAXN][MAXN][K][K]; // 二维ST表

// 构建二维ST表
void buildST(const vector<vector<int>>& matrix) {
    int n = matrix.size();
    int m = matrix[0].size();

    // 预处理对数表，便于查询时快速找到需要的k值
    logTable[1] = 0;
    for (int i = 2; i <= MAXN; ++i) {
        logTable[i] = logTable[i/2] + 1;
    }

```

```cpp
    // 初始化ST表
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            st[i][j][0][0] = matrix[i][j];
        }
    }

    // DP填表
    for (int x = 0; x < n; ++x) {
        for (int y = 0; y < m; ++y) {
            for (int kx = 1; (1 << kx) <= n; ++kx) {
                for (int ky = 1; (1 << ky) <= m; ++ky) {
                    int px = (1 << (kx - 1));
                    int py = (1 << (ky - 1));
                    st[x][y][kx][ky] = max(max(st[x][y][kx - 1][ky - 1], st[x + px][y][kx - 1][ky - 1]),
                                           max(st[x][y + py][kx - 1][ky - 1], st[x + px][y + py][kx - 1][ky - 1]));
                }
            }
        }
    }
}

// 查询二维区间最值
int queryST(int x1, int y1, int x2, int y2) {
    int kx = logTable[x2 - x1 + 1];
    int ky = logTable[y2 - y1 + 1];
    int px = (1 << kx);
    int py = (1 << ky);
    return max(max(st[x1][y1][kx][ky], st[x2 - px + 1][y1][kx][ky]),
               max(st[x1][y2 - py + 1][kx][ky], st[x2 - px + 1][y2 - py + 1][kx][ky]));
}

int main() {
    vector<vector<int>> matrix = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };
    buildST(matrix);

    int x1 = 0, y1 = 0, x2 = 2, y2 = 2;
    cout << "The maximum value in the range is: " << queryST(x1, y1, x2, y2) << endl;

```

```
    return 0;
}
```

带权并查集（维护点到根节点的距离）

带权并查集同时维护**每个结点到其祖先的距离，或者说将祖先看成根，维护每个点的深度dep**，代码实现就是在**路径压缩**的时候，维护一个dep[]数组，递归到根结点后回来的时候不断**dep[x] += dep[fa[x]]，同时让fa[x] = 祖先。**

```
//维护depth 要先find 再更新depth
// 维护depth 要先find 再更新depth
int find(int x)
{
    if (fa[x] == x)
        return fa[x];

    int tmp = fa[x];
    // 先find 这样子 fa[x]的depth才是正确的
    fa[x] = find(fa[x]);
    // depth代表的是到父节点的距离 dep[x] += dep[tmp] 是因为 dep[tmp]已经是x的父亲到根节点的距离  但是dep[x]保存的是x到fa[x]的距离
    // 那dep[x] + dep[tmp] 就是更新成 x到根节点的距离
    dep[x] += dep[tmp];

    return fa[x];
}

// 深度只有在find一次father后 才是正确的到根节点的距离
void merge(int x, int y)
{
    int fx = find(x);
    int fy = find(y);

    fa[fy] = fx;
    dep[y] = dep[x] + 1;
}

for (int i = 1; i <= n; i++)
{
    fa[i] = i;
    ans[i] = 0;
    dep[i] = 0;
}
//合并的时候 直接merge（a，b） 即可
```

高位前缀和

```
//二维
for(int i=1;i<=n;++i)
{
        for(int j=1;j<=m;++j)
        {
                sum[i][j]+=sum[i][j-1];
        }
}
for(int i=1;i<=n;++i)
{
        for(int j=1;j<=m;++j)
        {
                sum[i][j]+=sum[i-1][j];
        }
}
//三维
for(int i=1;i<=n;++i)
{
        for(int j=1;j<=m;++j)
        {
                for(int k=1;k<=q;++k)
                {
                        sum[i][j][k]+=sum[i-1][j][k];
                }
        }
}
for(int i=1;i<=n;++i)
{
        for(int j=1;j<=m;++j)
        {
                for(int k=1;k<=q;++k)
                {
                        sum[i][j][k]+=sum[i][j-1][k];
                }
        }
}
for(int i=1;i<=n;++i)
{
        for(int j=1;j<=m;++j)
        {
                for(int k=1;k<=q;++k)
                {
                        sum[i][j][k]+=sum[i][j][k-1];
                }
```

```
45          }
46  }
```

高次/多次前缀和（利用卷积和mod的性质）

智乃酱最近学习了前缀和、差分。她现在对于这两个操作产生了浓厚的兴趣。

具体来说，前缀和是这样一种操作，假设s数组是a数组的前缀和数组，则有
$\begin{cases} s_0 = a_0 \\ s_i = a_i + s_{i-1} (i \geq 1) \end{cases}$
而差分的操作则是前缀和反过来，假设d数组是a数组的差分数组，则有
$\begin{cases} d_0 = a_0 \\ d_i = a_i - a_{i-1} (i \geq 1) \end{cases}$
我们将先求出s数组，再把s中的值赋值回a称为对a数组做一次前缀和，同理，将先求出d数组，再把d中的值赋值回a数组称之为一次差分。

现在我们给定一个长度大小为N的数组a和一个参数k

- 当$k > 0$时，请输出对a做k次前缀和后的结果。
- 当$k = 0$时，请直接输出a数组。
- 当$k < 0$时，请输出对a做k次差分后的结果。

为了避免数字过大，你只用输出一个模998244353后的模系数字。
也就是说输出的数字应该大小在$[0, 998244353)$之间，即不要输出负数，如果由于差分导致数字的值小于0，则输出时需要再加上998244353变成一个非负整数。

> 第一行输入两个整数$N, k (1 \leq N \leq 10^5, -10^{18} \leq k \leq 10^{18})$表示数组长度和输入的参数。
> 接下来一行输入N个用空格隔开的整数$a_i (0 \leq a_i \leq 10^8)$表示数组中每个元素的初始值。

输出描述:

> 输出一行N个整数，表示进行若干次操作后数组a中每个元素的值。

示例1

输入

```
5 1
1 1 1 1 1
```

输出

```
1 2 3 4 5
```

```
1 #include <bits/stdc++.h>
2 using namespace std;
```

```cpp
namespace NTT
{
    const long long g = 3;
    const long long p = 998244353;
    long long wn[35];
    long long pow2(long long a, long long b)
    {
        long long res = 1;
        while (b)
        {
            if (b & 1)
                res = res * a % p;
            a = a * a % p;
            b >>= 1;
        }
        return res;
    }
    void getwn()
    {
        for (int i = 0; i < 25; i++)
            wn[i] = pow2(g, (p - 1) / (1LL << i));
    }
    void ntt(long long *a, int len, int f)
    {
        long long i, j = 0, t, k, w, id;
        for (i = 1; i < len - 1; i++)
        {
            for (t = len; j ^= t >>= 1, ~j & t;)
                ;
            if (i < j)
                swap(a[i], a[j]);
        }
        for (i = 1, id = 1; i < len; i <<= 1, id++)
        {
            t = i << 1;
            for (j = 0; j < len; j += t)
            {
                for (k = 0, w = 1; k < i; k++, w = w * wn[id] % p)
                {
                    long long x = a[j + k], y = w * a[j + k + i] % p;
                    a[j + k] = (x + y) % p;
                    a[j + k + i] = (x - y + p) % p;
                }
            }
        }
        if (f)
        {
```

```
            for (i = 1, j = len - 1; i < j; i++, j--)
                swap(a[i], a[j]);
            long long inv = pow2(len, p - 2);
            for (i = 0; i < len; i++)
                a[i] = a[i] * inv % p;
        }
    }
    void mul(long long *a, long long *b, int l1, int l2)
    {
        int len, i;
        for (len = 1; len <= l1 + l2; len <<= 1)
            ;
        for (i = l1 + 1; i <= len; i++)
            a[i] = 0;
        for (i = l2 + 1; i <= len; i++)
            b[i] = 0;
        ntt(a, len, 0);
        ntt(b, len, 0);
        for (i = 0; i < len; i++)
            a[i] = a[i] * b[i] % p;
        ntt(a, len, 1);
    }
};
const int MAXN = 300005;
const long long mod = 998244353;
int n;
long long a[MAXN], inv[MAXN], ki[MAXN], k;
void init(long long n)
{
    inv[0] = inv[1] = 1;
    for (long long i = 2; i <= n; i++)
    {
        inv[i] = ((mod - mod / i) * inv[mod % i]) % mod;
    }
    return;
}
void get_ki(long long k, int len)
{
    k = (k % mod + mod) % mod;
    ki[0] = 1;
    for (int i = 1; i < len; ++i)
    {
        ki[i] = ki[i - 1] * inv[i] % mod * ((k + i - 1) % mod) % mod;
    }
}
int main()
{
```

```
    NTT::getwn();
    init(100000);
    scanf("%d %lld", &n, &k);
    get_ki(k, n);

    for (int i = 0; i < n; ++i)
    {
        scanf("%lld", &a[i]);
    }
    NTT::mul(a, ki, n, n);
    for (int i = 0; i < n; ++i)
    {
        printf("%lld%c", a[i], i == n - 1 ? '\n' : ' ');
    }
    return 0;
}
```

各种前缀和

二进制子集前缀和

考虑一开始的问题：$\forall i, 0 \le i \le 2^n - 1$，，求$\sum_{j \subset i} a_j$，其中 j属于i 定义为j的二进制表示是i的二进制表示的子集

因为是正序枚举的，所以i^(1<<j)是是当前这一维度，而此时还在上一维，所以这样就实现了高维的前缀和。这样的时间复杂度是$O(n * 2^n)$,而如果去枚举子集来做前缀和的话，时间复杂度是n^3的。

大意：
给定一个长度为2^n的数组，对于每一个k，1<=k<=2^n-1，求出最大的ai+aj，其中iorj<=k

思路：
关键就是如何处理i or j<=k,不难发现这蕴含的意思其实就是iorj的结果是k的二进制表示下的子集

所以我们直接跑高维前缀和，维护一下每一个k对应的能够用的最大值以及次大值即可

```
for(int j=0;j<n;++j)
{
    for(int i=0;i<(1<<n);++i)
    {
        if(i&(1<<j))
        {
            dp[i]+=dp[i^(1<<j)]
        }
    }
}
```

二进制超集后缀和

> 超集（Superset）是数学中的一个概念，用于描述集合之间的包含关系。如果集合 A 是集合 B 的超集，表示集合 A 包含集合 B 的所有元素。用数学符号表示，若 A 是 B 的超集，可以记作 $A \supseteq B$ 或 $B \subseteq A$。

```
for(int j=0;j<n;++j)
{
    for(int i=0;i<(1<<n);++i)
    {
        if((i&(1<<j))==0)
        {
            dp[i]+=dp[i^(1<<j)]
        }
    }
}
```

离散化

其实就是 sort unique 然后lowerbound查找到自己

离散化的目的就是减小值域，离散化就是保持元素间相对大小关系不变，然后改变将原本覆盖很散的数值集中起来，节约内存

```
int C[MAXN], L[MAXN];// 在main函数中...
memcpy(C, A, sizeof(A)); // 复制
sort(C, C + n); // 排序
int l = unique(C, C + n) - C; // 去重
for (int i = 0; i < n; ++i)L[i] = lower_bound(C, C + l, A[i]) - C + 1; // 查找
```

树状数组应用

- 解决二维偏序问题 （

> 二维偏序是这样一类问题：已知点对的序列 $(a_1,b_1),(a_2,b_2),(a_3,b_3),\cdots$ 并在其上定义某种**偏序**关系 $\prec$，现有点 (a_i,b_i)，求满足 $(a_j,b_j)\prec(a_i,b_i)$ 的 (a_j,b_j) 的数量。

总而言之，最简单的二维偏序 $(a_j,b_j)\prec(a_i,b_i)\overset{def}{=\!=}a_j\le a_i \text{ and } b_j\le b_i$ 的处理方法是选 a 为第一关键词， b 为第二关键词进行排序；如果必要，将 b 离散化；然后按顺序把 b 一个一个推入树状数组，动态求前缀和。

而其他二维偏序关系，可以作不同的处理转化为最简单的二维偏序。例如：

- $(a_j,b_j)\prec(a_i,b_i)\overset{def}{=\!=}a_j<a_i \text{ and } ?$ ：把第一关键词的小于等于改成小于，需要在对第二关键词排序时进行**逆序排序**。
- $(a_j,b_j)\prec(a_i,b_i)\overset{def}{=\!=}a_j\ge a_i \text{ and } ?$ 把第一关键词的小于等于改成大于等于，需要在对第一关键词排序时进行**逆序排序**。
- $(a_j,b_j)\prec(a_i,b_i)\overset{def}{=\!=}a_j>a_i \text{ and } ?$ 把第一关键词的小于等于改成大于，需要在对两个关键词排序时都进行**逆序排序**。
- $(a_j,b_j)\prec(a_i,b_i)\overset{def}{=\!=}? \text{ and } b_j<b_i$ ：把第二关键词的小于等于改成小于，查询时使用 `query(x-1)` 而不是 `query(x)` 。
- $(a_j,b_j)\prec(a_i,b_i)\overset{def}{=\!=}? \text{ and } b_j\ge b_i$ ：把第二关键词的小于等于改成大于等于，对第二关键词**离散化**时进行**逆序排序**。
- $(a_j,b_j)\prec(a_i,b_i)\overset{def}{=\!=}? \text{ and } b_j\ge b_i$ ：把第二关键词的小于等于改成大于等于，对第二关键词**离散化**时进行**逆序排序**，并且查询时使用 `query(x-1)` 而不是 `query(x)` 。

前面两个例题没有对第二关键词逆序排序的原因，是它们都保证了第一关键词不重复，这样是小于还是小于等于就无所谓了，但一般的情形，还是需要自己写结构体实现的。

图论部分

最大生成树

3.2 实现

1、将图中所有边的边权变为相反数，再跑一遍最小生成树算法。相反数最小，原数就最大。

2、修改一下最小生成树算法：对于kruskal，将“从小到大排序”改为“从大到小排序”；

对于prim，将“每次选到所有蓝点代价最小的白点”改为“每次选到所有蓝点代价最大的点”

- 或者将每个边的权值变成相反数，跑一遍最小生成树。

二，最大生成树

最大生成树算法和最小并无区别，只需把cmp函数的`<`改成`>`即可
特性：加入的边数限制`k`，在kruskal算法中加入边数的计数器`op`若是等于`k`即刻跳出并输出答案
注意：**最大spanning tree不一定联通所有点**

- 典例：洛谷 P2121 拆地毯
- 典例解答： AC代码

次小生成树

- 非严格次小生成树

非严格次小生成树

定义

在无向图中，边权和最小的满足边权和 **大于等于** 最小生成树边权和的生成树

求解方法

- 求出无向图的最小生成树 T，设其权值和为 M
- 遍历每条未被选中的边 $e=(u,v,w)$，找到 T 中 u 到 v 路径上边权最大的一条边 $e'=(s,t,w')$，则在 T 中以 e 替换 e'，可得一棵权值和为 $M'=M+w-w'$ 的生成树 T'.
- 对所有替换得到的答案 M' 取最小值即可

如何求 u,v 路径上的边权最大值呢?

我们可以使用倍增来维护，预处理出每个节点的 2^i 级祖先及到达其 2^i 级祖先路径上最大的边权，这样在倍增求 LCA 的过程中可以直接求得。

- 严格次小生成树

严格次小生成树

定义

在无向图中，边权和最小的满足边权和 **严格大于** 最小生成树边权和的生成树

求解方法

考虑刚才的非严格次小生成树求解过程，为什么求得的解是非严格的?

因为最小生成树保证生成树中 u 到 v 路径上的边权最大值一定 **不大于** 其他从 u 到 v 路径的边权最大值。换言之，当我们用于替换的边的权值与原生成树中被替换边的权值相等时，得到的次小生成树是非严格的。

解决的办法很自然：我们维护到 2^i 级祖先路径上的最大边权的同时维护 **严格次大边权**，当用于替换的边的权值与原生成树中路径最大边权相等时，我们用严格次大值来替换即可。

这个过程可以用倍增求解，复杂度 $O(m \log m)$。

```cpp
#include <algorithm>
#include <iostream>

const int INF = 0x3fffffff;
const long long INF64 = 0x3fffffffffffffffLL;

struct Edge {
  int u, v, val;

  bool operator<(const Edge &other) const { return val < other.val; }
};

Edge e[300010];
bool used[300010];

int n, m;
long long sum;

class Tr {
 private:
  struct Edge {
    int to, nxt, val;
  } e[600010];

  int cnt, head[100010];

  int pnt[100010][22];
```

```cpp
  int dpth[100010];
  // 到祖先的路径上边权最大的边
  int maxx[100010][22];
  // 到祖先的路径上边权次大的边，若不存在则为 -INF
  int minn[100010][22];

public:
  void addedge(int u, int v, int val) {
    e[++cnt] = (Edge){v, head[u], val};
    head[u] = cnt;
  }

  void insedge(int u, int v, int val) {
    addedge(u, v, val);
    addedge(v, u, val);
  }

  void dfs(int now, int fa) {
    dpth[now] = dpth[fa] + 1;
    pnt[now][0] = fa;
    minn[now][0] = -INF;
    for (int i = 1; (1 << i) <= dpth[now]; i++) {
      pnt[now][i] = pnt[pnt[now][i - 1]][i - 1];
      int kk[4] = {maxx[now][i - 1], maxx[pnt[now][i - 1]][i - 1],
                   minn[now][i - 1], minn[pnt[now][i - 1]][i - 1]};
      // 从四个值中取得最大值
      std::sort(kk, kk + 4);
      maxx[now][i] = kk[3];
      // 取得严格次大值
      int ptr = 2;
      while (ptr >= 0 && kk[ptr] == kk[3]) ptr--;
      minn[now][i] = (ptr == -1 ? -INF : kk[ptr]);
    }

    for (int i = head[now]; i; i = e[i].nxt) {
      if (e[i].to != fa) {
        maxx[e[i].to][0] = e[i].val;
        dfs(e[i].to, now);
      }
    }
  }

  int lca(int a, int b) {
    if (dpth[a] < dpth[b]) std::swap(a, b);

    for (int i = 21; i >= 0; i--)
      if (dpth[pnt[a][i]] >= dpth[b]) a = pnt[a][i];
```

```
75
76      if (a == b) return a;
77
78      for (int i = 21; i >= 0; i--) {
79        if (pnt[a][i] != pnt[b][i]) {
80          a = pnt[a][i];
81          b = pnt[b][i];
82        }
83      }
84      return pnt[a][0];
85    }
86
87    int query(int a, int b, int val) {
88      int res = -INF;
89      for (int i = 21; i >= 0; i--) {
90        if (dpth[pnt[a][i]] >= dpth[b]) {
91          if (val != maxx[a][i])
92            res = std::max(res, maxx[a][i]);
93          else
94            res = std::max(res, minn[a][i]);
95          a = pnt[a][i];
96        }
97      }
98      return res;
99    }
100 } tr;
101
102 int fa[100010];
103
104 int find(int x) { return fa[x] == x ? x : fa[x] = find(fa[x]); }
105
106 void Kruskal() {
107   int tot = 0;
108   std::sort(e + 1, e + m + 1);
109   for (int i = 1; i <= n; i++) fa[i] = i;
110
111   for (int i = 1; i <= m; i++) {
112     int a = find(e[i].u);
113     int b = find(e[i].v);
114     if (a != b) {
115       fa[a] = b;
116       tot++;
117       tr.insedge(e[i].u, e[i].v, e[i].val);
118       sum += e[i].val;
119       used[i] = 1;
120     }
121     if (tot == n - 1) break;
```

```cpp
  }
}

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

  std::cin >> n >> m;
  for (int i = 1; i <= m; i++) {
    int u, v, val;
    std::cin >> u >> v >> val;
    e[i] = (Edge){u, v, val};
  }

  Kruskal();
  long long ans = INF64;
  tr.dfs(1, 0);

  for (int i = 1; i <= m; i++) {
    if (!used[i]) {
      int _lca = tr.lca(e[i].u, e[i].v);
      // 找到路径上不等于 e[i].val 的最大边权
      long long tmpa = tr.query(e[i].u, _lca, e[i].val);
      long long tmpb = tr.query(e[i].v, _lca, e[i].val);
      // 这样的边可能不存在，只在这样的边存在时更新答案
      if (std::max(tmpa, tmpb) > -INF)
        ans = std::min(ans, sum - std::max(tmpa, tmpb) + e[i].val);
    }
  }
  // 次小生成树不存在时输出 -1
  std::cout << (ans == INF64 ? -1 : ans) << '\n';
  return 0;
}
```

tarjan求强连通分量（缩点变成dag

就是找自环 把换缩成一个点

给一个n个点m条边的无向连通图，从小到大输出所有割点的编号。

输入格式

第一行包含两个数n,m(1≤n≤105,1≤m≤2×105)

接下来m行，每行两个整数u,v表示一条无向边，没有自环，但可能有重边。

输出格式

首先输出一个整数，表示割点个数。接下来一行，若干个从小到大的数，表示割点的编号。

```cpp
const int N = 4e5 + 10, M = 5e3 + 10, mod = 998244353;

int n, m;
int dfn[N], ins[N], low[N], cnt, idx;
int stk[N], top;
vector<vector<int>> scc;
vector<int> e[N];

void dfs(int u)
{
    dfn[u] = low[u] = ++idx;
    ins[u] = 1;
    stk[++top] = u;
    for (auto i : e[u])
    {
        if (!dfn[i])
        {
            dfs(i);
            low[u] = min(low[u], low[i]);
        }
        else if (ins[i])
            low[u] = min(low[u], dfn[i]);
    }
    if (low[u] == dfn[u])
    {
        vector<int> tmp;
        while (top)
        {
            int x = stk[top--];
            tmp.push_back(x);
            ins[x] = 0;
            if (x == u)
                break;
        }
        sort(tmp.begin(), tmp.end());
        scc.push_back(tmp);
    }
}

void solve()
{
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= m; i++)
    {
```

```
45          int u, v;
46          scanf("%d%d", &u, &v);
47          e[u].push_back(v);
48      }
49      for (int i = 1; i <= n; i++)
50      {
51          if (!dfn[i])
52              dfs(i);
53      }
54      sort(scc.begin(), scc.end());
55      for (auto i : scc)
56      {
57          for (auto j : i)
58              printf("%d ", j);
59          printf("\n");
60      }
61  }
```

---

tarjan求割边

给一个n个点m条边的无向连通图，从小到大输出所有割边的编号。注意：边从1开始标号。

输入格式

第一行包含两个数n,m(1≤n≤105,1≤m≤2×105)

接下来m行，每行两个整数u,v表示一条无向边，没有自环，但可能有重边。

输出格式

首先输出一个整数，表示割边个数。接下来一行，若干个从小到大的数，表示割边的编号。

```
1  const int N = 1e6 + 10;
2  vector<array<int, 2>> e[N];
3  vector<int> bridge;
4  int dfn[N], low[N], idx;
5  int n, m;
6
7  void dfs(int u, int id)
8  {
9      dfn[u] = low[u] = ++idx;
10
11     for (auto [v, i] : e[u])
12     {
13         if (!dfn[v])
14         {
```

```
            dfs(v, i);
            low[u] = min(low[u], low[v]);
        }
        else if (id != i)
            low[u] = min(low[u], dfn[v]);
    }

    if (low[u] == dfn[u] && id)
        bridge.push_back(id);
}

void run()
{
    cin >> n >> m;

    for (int i = 1; i <= m; i++)
    {
        int u, v;
        cin >> u >> v;
        e[u].push_back({v, i});
        e[v].push_back({u, i});
    }

    dfs(1, 0);

    sort(bridge.begin(), bridge.end());

    cout << bridge.size() << endl;

    for (int i : bridge)
        cout << i << ' ';
    cout << endl;
}
```

---

tarjan求割点 （low更新的时候必须规范 割边和scc的low比较随性

给一个n个点m条边的无向连通图，从小到大输出所有割点的编号。

输入格式

第一行包含两个数n,m(1≤n≤105,1≤m≤2×105)

接下来m行，每行两个整数u,v表示一条无向边，没有自环，但可能有重边。

输出格式

首先输出一个整数，表示割点个数。接下来一行，若干个从小到大的数，表示割点的编号。

```
#define pb push_back
const int N = 1e5 + 10;
vector<int> e[N];
// st为1代表是割点
int n, m, cnt, dfn[N], low[N], st[N];

void dfs(int u, int f)
{
    dfn[u] = low[u] = ++cnt;

    int cnt = 0;
    for (auto idx : e[u])
    {
        if (!dfn[idx])
        {
            cnt++;
            dfs(idx, u);
            low[u] = min(low[u], low[idx]);
            if (low[idx] >= dfn[u])
                st[u] = 1;
        }
        else if (idx != f)
            low[u] = min(low[u], dfn[idx]);
    }
    if (u == 1 && cnt == 1)
        st[u] = 0;
}
void run()
{
    cin >> n >> m;

    for (int i = 1; i <= m; i++)
    {
        int u, v;
        cin >> u >> v;

        e[u].pb(v);
        e[v].pb(u);
    }

    dfs(1, 0);

    int cnt = 0;
    for (int i = 1; i <= n; i++)
```

```
        if (st[i])
            cnt++;

    cout << cnt << endl;
    for (int i = 1; i <= n; i++)
        if (st[i])
            cout << i << ' ';
}

/*
1）当前节点为树根时，成为割点的条件是“要有多于一个子树”
（如果只有一棵子树，去掉这个点也没有影响，如果有两颗子树，去掉这个点，两颗子树就不连通了）
2）当前节点不是树根的时候，条件是“low [ v ] >= dfn [ u ] ”，
也就是在u之后遍历的点，能够向上翻，最多到u。
（如果能翻到u的上方，那就有环了，去掉u之后，图仍然连通。）
所以，保证v向上翻最多到u才可以
*/
```

---

dfn求lca（比倍增求lca快

如题，给定一棵有根多叉树，请求出指定两个点直接最近的公共祖先。

输入格式

第一行包含三个正整数 *N*,*M*,*S*，分别表示树的结点个数、询问的个数和树根结点的序号。

接下来 *N*−1 行每行包含两个正整数*x*,*y*，表示 *x* 结点和 *y* 结点之间有一条直接连接的边（数据保证可以构成树）。

接下来 *M* 行每行包含两个正整数 *a*,*b*，表示询问 *a* 结点和 *b* 结点的最近公共祖先。

输出格式

输出包含 *M* 行，每行包含一个正整数，依次为每一个询问的结果。

```
const int N = 5e5 + 5;
int n, m, R, dn, dfn[N], mi[19][N];
vector<int> e[N];

int get(int x, int y) { return dfn[x] < dfn[y] ? x : y; }
void dfs(int id, int f)
{
    mi[0][dfn[id] = ++dn] = f;
    for (int it : e[id])
        if (it != f)
            dfs(it, id);
```

```
}
int lca(int u, int v)
{
    if (u == v)
        return u;
    if ((u = dfn[u]) > (v = dfn[v]))
        swap(u, v);
    int d = __lg(v - u++);
    return get(mi[d][u], mi[d][v - (1 << d) + 1]);
}

int main()
{
    scanf("%d %d %d", &n, &m, &R);

    for (int i = 2, u, v; i <= n; i++)
    {
        scanf("%d %d", &u, &v);
        e[u].push_back(v), e[v].push_back(u);
    }

    dfs(R, 0);

    for (int i = 1; i <= __lg(n); i++)
        for (int j = 1; j + (1 << i) - 1 <= n; j++)
            mi[i][j] = get(mi[i - 1][j], mi[i - 1][j + (1 << i - 1)]);
    for (int i = 1, u, v; i <= m; i++)
        scanf("%d %d", &u, &v), printf("%d\n", lca(u, v));
```

求树的直径

方法一：O(2n)

优点 : 可以通过一个新的数组记录路径信息(例如父节点与子节点之间的关系)

缺点 : 无法处理 负边权(遇到 负边权 凉凉)

两次dfs：随便选一个点作为起点 进行第一次dfs 搜到距离起点最远的一个点A，然后用这个最远的点A作为起点 再搜一次 再搜到距离这个点最远的点B，那么 AB 就是树的直径的两个端点。

```
1
```

方法二：树形DP

优点 : 可以有效处理 负边权

缺点 : 对于记录路径的信息效率较低

需要开两个数组F1[]和F2[]分别记录到某个点的最远距离和次远距离，这样我们最后把每个点的两个数组相加取个最大值就可以找到树的直径了。

```
void dp(int x,int father)
{
        for(int i=h[x];i!=-1;i=ne[i])链式前向星存树
        {
                int j=e[i];
                if(j==father) continue;//防止原路返回
                dp(j,x);//dp过程应该是由叶节点开始的，也就是说先递归到叶节点再开始进行状态转移
                if(f1[x]<f1[j]+w[i])//如果子节点的最大距离+子节点与父节点之间边的权重大于父节点的最大距离，那么父节点的最大距离和次大距离都要得到相应更新
                {
                        f2[x]=f1[x];
                        f1[x]=f1[j]+w[i];
                }
                else if(f2[x]<f1[j]+w[i])//若子节点的最大距离+子节点与父节点之间边的权重小于父节点的最大距离，再判断与父节点的次大距离的关系
                        f2[x]=f1[j]+w[i];
                ans=max(ans,f1[x]+f2[x]);//在搜索过程中找到树的直径
        }
}
```

普通环计数

> 给定一个简单图，求图中简单环的数目。简单环是指没有重复顶点或边的环。
>
> 结点数目 $1 \leq n \leq 19$。

```
#include <bits/stdc++.h>
using namespace std;

int n, m;

struct Edge
{
    int to, nxt;
} edge[500];
```

```
10
11 int cntEdge, head[20];
12
13 void addEdge(int u, int v)
14 {
15     edge[++cntEdge] = {v, head[u]}, head[u] = cntEdge;
16 }
17
18 long long answer, dp[1 << 19][20];
19
20 int main()
21 {
22     scanf("%d%d", &n, &m);
23     for (int i = 1; i <= m; i++)
24     {
25         int u, v;
26         scanf("%d%d", &u, &v);
27         addEdge(u, v);
28         addEdge(v, u);
29     }
30     for (int i = 1; i <= n; i++)
31         dp[1 << i - 1][i] = 1;
32     for (int s = 1; s < (1 << n); s++)
33         for (int i = 1; i <= n; i++)
34         {
35             if (!dp[s][i])
36                 continue;
37             for (int j = head[i]; j; j = edge[j].nxt)
38             {
39                 int u = i, v = edge[j].to;
40                 if ((s & -s) > (1 << v - 1))
41                     continue;
42                 if (s & (1 << v - 1))
43                 {
44                     if ((s & -s) == (1 << v - 1))
45                         answer += dp[s][u];
46                 }
47                 else
48                     dp[s | (1 << v - 1)][v] += dp[s][u];
49             }
50         }
51     printf("%lld\n", (answer - m) / 2);
52     return 0;
53 }
```

无向图三元环计数（根号算法）

题目描述

给定一个 n 个点， m 条边的简单无向图，求其三元环个数。 $1 \leq n \leq 10^5$， $1 \leq m \leq 2 \times 10^5$。保证图没有 **重边** 和 **自环**。但是不保证图联通。

正解：

我们把无向图变成 **有向图**，给每个边加一个**方向**。

加方向的规则是，对于一条边 (u, v) 而言，我们让其由度数小的点指向度数大的点。如果度数一样，就让编号小的点指向编号大的点。

具体来讲， $u \to v$ 的条件是 $deg_u < deg_v$ 或者 $deg_u = deg_v$ 并且 $u < v$。

我们发现，在这样的条件下，形成的有向图 **一定无环**，可以把连边规则看作是优先级，那么有向边就是某一组优先关系，所以一定不会存在环。 进一步，我们发现，原图中的所有环一定等价于所有的形如 $u \to v$， $u \to k$， $v \to k$ 的三元关系。我们只要找出来所有这样的三元关系就好了。

方法是我们枚举一个点 u 的所有出边，并且把所有出点 v 打上时间戳为 u 的标记。然后枚举所有 v 的出边，如果有一个出点 k 的时间戳是 u，那么让答案加1。这样做等价于 以每一个点作为环中优先级最低的点，找出所有这样的环的数量。所以是不重不漏的。

我们来分析复杂度。

首先就是任意一个点的出度不会超过 $\sqrt{m}$。因为如果一个点在原图中的度数小于 $\sqrt{m}$，那么在有向图上它都出度不会超过原图上的度数。所以小于 $\sqrt{m}$。

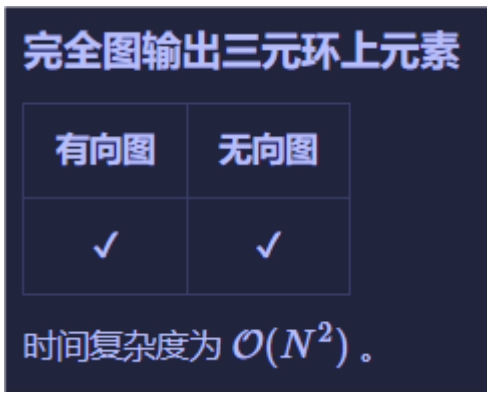
如果一个点在原图中的度数大于 $\sqrt{m}$，但是连边的条件是度数小的点朝度数大的点连边，因为总边数为 m，所以度数大于 $\sqrt{m}$ 的点不会超过 $\sqrt{m}$ 个，因此这个点的出度也不会超过 $\sqrt{m}$。

每一个点对复杂度的贡献是 $In_i \times Out_i$， In_i 是 i 号点的入度， Out_i 是 i 号点的出度。因为 Out_i 是 $\sqrt{m}$ 量级，而 $\sum_{i=1}^{n} In_i = m$。所以总复杂度是 $O(m \times \sqrt{m})$。

```cpp
#include <bits/stdc++.h>
#define N 100100
#define M 200100
#define pb push_back
#define LL long long
using namespace std;
int n, m, u[M], v[M], deg[N], tim[N];
vector<int> E[N];
LL res;
int main()
{
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= m; i++)
    {
        scanf("%d%d", &u[i], &v[i]);
        deg[u[i]]++;
        deg[v[i]]++;
    }
    for (int i = 1; i <= m; i++)
    {
        if (deg[u[i]] != deg[v[i]])
        {
            if (deg[v[i]] > deg[u[i]])
                swap(u[i], v[i]);
            E[v[i]].pb(u[i]); // 每一个里面放比它度数大的
```

```
        }
        else
        {
            if (u[i] > v[i])
                swap(u[i], v[i]);
            E[u[i]].pb(v[i]);
        }
    }
    for (int u = 1; u <= n; u++)
    {
        for (auto v : E[u])
        { // 扫描出边， 复杂度O（m）
            tim[v] = u;
        }
        for (auto v : E[u])
        {
            for (auto x : E[v])
            { // 扫描出边的出边，每一条边会被扫到一次，一个边贡献√m的复杂度，所以总复杂度是O（m√m） 的
                if (tim[x] == u)
                    res++;
            }
        }
    }
    printf("%lld\n", res);
    return 0;
}
```

输出任意一个三元环上的全部元素




```
//该代码理论上适用于非完全图和无向图
bool Solve() {
    int n; cin >> n;
    vector<vector<int> > a(n + 1, vector<int> (n + 1));
    for (int i = 1; i <= n; ++ i) {
```

```
            for (int j = 1; j <= n; ++ j) {
                    char x; cin >> x;
                    if (x == '1') a[i][j] = 1;
            }
    }

    vector<int> vis(n + 1);
    function<void(int, int)> dfs = [&] (int x, int fa) {
            vis[x] = 1;
            for (int y = 1; y <= n; ++ y) {
                    if (a[x][y] == 0) continue;
                    if (a[y][fa] == 1) {
                            cout << fa << " " << x << " " << y;
                            exit(0);
                    }
                    if (!vis[y]) dfs(y, x); // 这一步的if判断很关键
            }
    };
    for (int i = 1; i <= n; ++ i) {
            if (!vis[i]) dfs(i, -1);
    }
    cout << -1;
    return 0;
}
```

四元环计数

类似地，**四元环** 就是指四个点 a, b, c, d 满足 (a, b)，(b, c)，(c, d) 和 (d, a) 均有边连接。

考虑先对点进行排序。度数小的排在前面，度数大的排在后面。

考虑枚举排在最后面的点 a，此时只需要对于每个比 a 排名更前的点 c，都求出有多少个排名比 a 前的点 b 满足 (a, b)，(b, c) 有边。然后只需要从这些 b 中任取两个都能成为一个四元环。求 b 的数量只需要遍历一遍 b 和 c 即可。

注意到我们枚举的复杂度本质上与枚举三元环等价，所以时间复杂度也是 $O(m\sqrt{m})$（假设 n, m 同阶）。

值得注意的是，(a, b, c, d) 和 (a, c, b, d) 可以是两个不同的四元环。

另外，度数相同的结点的排名将不相同，并且需要注意判断 $a \neq c$。

```
#include <bits/stdc++.h>
using namespace std;
```

```cpp

int n, m, deg[100001], cnt[100001];
vector<int> E[100001], E1[100001];

long long total;

int main()
{
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= m; i++)
    {
        int u, v;
        scanf("%d%d", &u, &v);
        E[u].push_back(v);
        E[v].push_back(u);
        deg[u]++, deg[v]++;
    }
    for (int u = 1; u <= n; u++)
        for (int v : E[u])
            if (deg[u] > deg[v] || (deg[u] == deg[v] && u > v))
                E1[u].push_back(v);
    for (int a = 1; a <= n; a++)
    {
        for (int b : E1[a])
            for (int c : E[b])
            {
                if (deg[a] < deg[c] || (deg[a] == deg[c] && a <= c))
                    continue;
                total += cnt[c]++;
            }
        for (int b : E1[a])
            for (int c : E[b])
                cnt[c] = 0;
    }
    printf("%lld\n", total);
    return 0;
}
```

最小环相关算法

输出图上最小环大小

给出一张无向图，求解该图最小环的大小。

```cpp
//=====================（其一）： flody （n^3）
```

```
================================
int flody(int n) {
    for (int i = 1; i <= n; ++ i) {
        for (int j = 1; j <= n; ++ j) {
            val[i][j] = dis[i][j]; // 记录最初的边权值
        }
    }
    int ans = 0x3f3f3f3f;
    for (int k = 1; k <= n; ++ k) {
        for (int i = 1; i < k; ++ i) { // 注意这里是没有等于号的
            for (int j = 1; j < i; ++ j) {
                ans = min(ans, dis[i][j] + val[i][k] + val[k][j]);
            }
        }
    for (int i = 1; i <= n; ++ i) { // 往下是标准的flody
        for (int j = 1; j <= n; ++ j) {
            dis[i][j] = min(dis[i][j], dis[i][k] + dis[k][j]);
            }
        }
    }
    return ans;
}
//====================（其二）：
bfs============================================
auto bfs = [&] (int s) {
    queue<int> q; q.push(s);
    dis[s] = 0;
    fa[s] = -1;
    while (q.size()) {
        auto x = q.front(); q.pop();
        for (auto y : ver[x]) {
            if (y == fa[x]) continue;
            if (dis[y] == -1) {
                dis[y] = dis[x] + 1;
                fa[y] = x;
                q.push(y);
            }
            else ans = min(ans, dis[x] + dis[y] + 1);
        }
    }
};
for (int i = 1; i <= n; ++ i) {
    fill(dis + 1, dis + 1 + n, -1);
    bfs(i);
}
cout << ans;
```

简单环相关模板

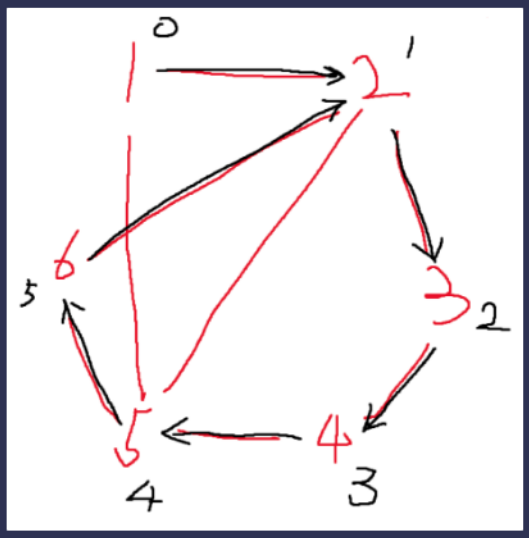
输出图上简单环数量： 状压dp

```
1
```

输出图上任意一个简单环：dfs序

注意限定：**是简单环而不是最小环。**

时间仓促，直接引用做题时的截图为例，在下图中，最小环为 $2-6-5-2$，而使用 **dfs** 会输出 $2-3-4-5-6-2$ 这个简单环。

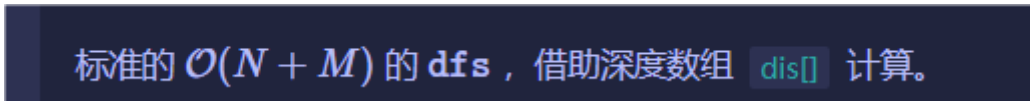


虽然这个做法不能找到最小环，但是由于其优秀的复杂度，在某些题目的解题过程中是必不可少的。

```
function<void(int, int)> dfs = [&] (int x, int f) {
    for (auto y : ver[x]) {
        if (y == f) continue;
        if (dis[y] == -1) {
            dis[y] = dis[x] + 1;
            fa[y] = x;
            dfs(y, x);
        }
        else if (dis[y] < dis[x] && dis[x] - dis[y] <= k - 1) { // 遇到了更小的时间戳
            cout << dis[x] - dis[y] + 1 << endl; // 输出简单环的大小
            int pre = x;
            cout << pre << " "; // 输出环上元素
            while (pre != y) {
                pre = fa[pre];
                cout << pre << " ";
            }
            exit(0);
        }
```

```
    }
};
dis[1] = 0;
dfs(1, -1);
```

输出有向图简单环大小 dfs染色




```
vector<int> vis(n + 1), dis(n + 1), ring;
function<void(int)> dfs = [&] (int x) {
        vis[x] = 1;
        for (auto y : ver[x]) {
                if (vis[y] == 0) {
                        dis[y] = dis[x] + 1;
                        dfs(y);
                }
                else if (vis[y] == 1) {
                        ring.push_back(dis[x] - dis[y] + 1);
                }
        }
        vis[x] = 2;
};
for (int i = 1; i <= n; ++ i) {
        if (!vis[i]) dfs(i);
}

for (auto it : ring) {
        cout << it << " ";
}
```

环相关模板

判断图上是否存在环 topsort

- **拓扑排序的步骤**

> 1. 找到一个入度为0的点，将它放入拓扑序列中。
> 2. 删除这个点所关联的所有边，即它的子节点的入度全部-1
> 3. 回到第一步，知道所有的点已被删除。

- **拓扑排序的应用**

由于根据拓扑排序的定义以及实现，若图中存在环，则拓扑排序会中断，即找不到一个入度为0的点，也就是说若图中存在环，拓扑排序后的元素个数不足 n。

因此，我们只需要将图的拓扑序的长度与节点数比较，若相等则图为 DAG （有向无环图），否则则存在环。

```cpp
#include <bits/stdc++.h>
using namespace std;
int n,m;
vector<int> g[100010];
int top[100010],in[100010],t=0;
queue<int> q;
bool vis[100010]={false};
void topsort(int s){
        q.push(s);  //找到第一个入度为0的点放入队列
        while(!q.empty())
          {
                  int u=q.front();
                  q.pop();
                  vis[u]=true;  //表示已经删除了
                  top[++t]=u;  //记录
                  for(int i=0;i<g[u].size();i++)
                    {
                      in[g[u][i]]--;  //删边
                      if(!in[g[u][i]]&&!vis[g[u][i]]) q.push(g[u][i]);  //若有入度为0的点则放入队列
                    }
          }
}
int main(){
        cin>>n>>m;
        int a,b;
        for(int i=1;i<=m;i++)
          {
                  cin>>a>>b;
                  g[a].push_back(b);
                  in[b]++;  //记录入度
          }
        for(int i=1;i<=n;i++)
          if(!in[i])
            {
```

```
                    topsort(i);
                    break;
                }
        if(t==n) cout<<"Yes";
        else cout<<"No";
        cout<<endl;
        //for(int i=1;i<=t;i++)
          //cout<<top[i]<<" ";
        return 0;
}
```

判断图上是否存在环 dfs染色

初始时所有点颜色均为 0，开始对这个点进行 dfs 前将其染为 1，当结束对这个点的 dfs 时将其染为 2。当在 dfs 的过程中遇到 1 时说明存在环。

```
function<void(int)> dfs(int x) {
    vis[x] = 1;
    for (auto y : ver[x]) {
        if (vis[y] == 0) dfs(y); //如果未被搜索过
        else if (vis[y] == 1) ++ ans; //如果已经被搜索过，说明找到了一个环
    }
    vis[x] = 2;
};
for (int i = 1; i <= n; ++ i) if (vis[i] == 0) {
    dfs(i);
}
cout << ans;
```

判断无向图是否存在环 dsu

dsu判断新连接的两个点是否具有同一祖先

```

```

输出有向图环上元素 tarjan

```
namespace SCC { // 在有向图中将强连通分量缩点后重建图
    vector<PII> ver[N];
    int time[N], time_cnt, upper[N];
    int color[N], color_cnt;
    stack<int> S; int v[N];
```

```
6
7       void clear(int n) {
8           for (int i = 1; i <= n; ++ i) {
9               ver[i].clear();
10              v[i] = time[i] = upper[i] = color[i] = 0;
11          }
12          time_cnt = color_cnt = 0;
13          while (!S.empty()) S.pop();
14      }
15      void add(int x, int y, int w) { ver[x].push_back({y, w}); }
16      void tarjan(int x) {
17          time[x] = upper[x] = ++ time_cnt;
18          S.push(x); v[x] = 1; // v数组用于记录x点此时是否在S中
19          for (auto [y, w] : ver[x]) {
20              if (!time[y]) {
21                  tarjan(y);
22                  upper[x] = min(upper[x], upper[y]);
23              }
24              else if (v[y] == 1) upper[x] = min(upper[x], time[y]);
25          }
26          if (upper[x] == time[x]) {
27              int pre = 0; ++ color_cnt; // colorCnt代表强连通分量的数量
28              do {
29                  pre = S.top(); S.pop();
30                  v[pre] = 0;
31                  color[pre] = color_cnt; // 给相同强连通分量内的点染色
32              } while (pre != x);
33          }
34      }
35      void solve(int n, function<void(int, int, int)> add) {
36          for (int i = 1; i <= n; ++ i) { // 若原图不连通
37              if (time[i] == 0) tarjan(i);
38          }
39          //基于已染的颜色和附加条件重建图
40          for (int x = 1; x <= n; ++ x) {
41              for (auto [y, w] : ver[x]) {
42                  int X = color[x], Y = color[y];
43                  if (X != Y) add(X, Y, w); //【这里很容易写错】
44              }
45          }
46      }
47  } // namespace SCC
```

判断负环

- tarjan 判断负环 O(v + e)

tarjan有点危险原因如图（doge）

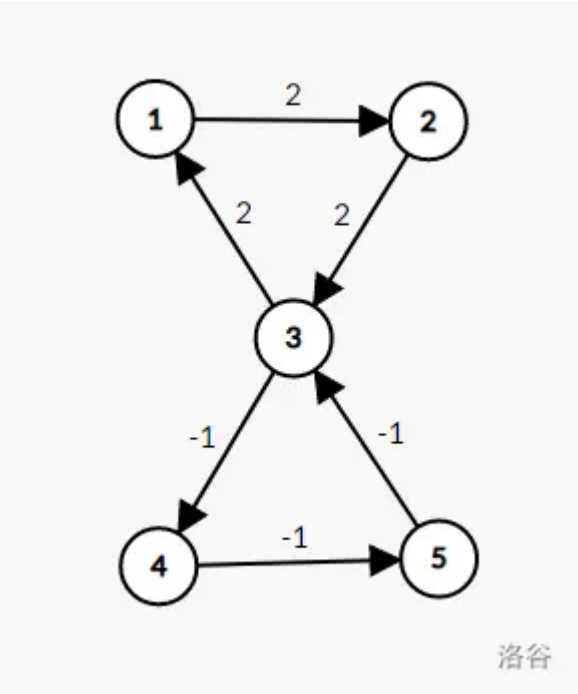
gpt的结果是：可以缩完点以后，在每一个scc里面跑一个其他算法。。。

比如spfa？或者bellman-ford。。。

复杂度分析：

- **Tarjan算法找到所有强连通分量**：时间复杂度为 $O(V+E)$。
- **对每个强连通分量构建子图并使用Bellman-Ford算法检测负权环**：
  - 最坏情况下，所有顶点和边都在一个强连通分量中，此时Bellman-Ford算法的时间复杂度为 $O(VE)$。
  - 在实际应用中，多个强连通分量会减少每次Bellman-Ford算法的图规模，从而降低总体复杂度。

因此，总体时间复杂度为 $O(V+E)+O(VE)=O(VE)$，这在检测负权环的问题中是可以接受的。



- spfa 判断负环 O (*kn*)
- DFS-SPFA 求负环，时间复杂度 ☒(☒2)
- Bellman-Ford 算法求负环，时间复杂度 ☒(☒☒)
- Floyd 判断负环 O(n ^ 3)

```
//=========================tarjan 判断负环 O( v + e ) ======================
计算每个强连通分量中所有边的权重和，如果存在负值，则该连通分量中存在负权环
//=========================spfa 判断负环   O (kn )============================
int cnt[MAXN],dis[MAXN];
```

```cpp
queue<int>q;
bool vis[MAXN];
int spfa(int s){
        memset(vis,0,sizeof(vis));
        memset(dis,0x7f,sizeof(dis));
        memset(cnt,0,sizeof(cnt));
        while(!q.empty())
                q.pop();
        vis[s]=1;
        dis[s]=0;
        q.push(s);
        while(!q.empty())
        {
                int u=q.front();
                vis[u]=0;
                q.pop();
                for(int i=head[u];i;i=e[i].next)
                {
                        int v=e[i].v;
                        if(dis[v]>dis[u]+e[i].w)
                        {
                                dis[v]=dis[u]+e[i].w;
                                if(!vis[v])
                                {
                                        if(++cnt[v]>n)
                                                return 1;
                                        q.push(v);
                                        vis[v]=1;
                                }
                        }
                }
        }
        return 0;
}
//=========================DFS-SPFA 求负环，时间复杂度  ( 2):
=====================
int dis[MAXN];
int stk[MAXN],sp;
int pre[MAXN],cnt[MAXN];//从哪个点搜过来的 / 从一个点出发的搜索分支数bool
vis[MAXN],instk[MAXN];
int dfs_spfa(int s){
        memset(vis,0,sizeof(vis));
        memset(dis,0x7f,sizeof(dis));
        memset(cnt,0,sizeof(cnt));
        memset(instk,0,sizeof(instk));
        sp=0;
        stk[++sp]=s;
```

```
        dis[s]=0;
        instk[s]=1;
        while(sp)
        {
                int u=stk[sp--];instk[u]=0;
                vis[u]=1;
                for(int i=head[u];i;i=e[i].next)
                {
                        int v=e[i].v;
                        if(dis[v]>dis[u]+e[i].w)
                        {
                                dis[v]=dis[u]+e[i].w;
                                if(vis[v])
                                        return 1;
                                if(!instk[v])
                                        stk[++sp]=v,
                                        cnt[u]++,
                                        pre[v]=u,
                                        instk[v]=1;
                        }
                }
                while(u&&cnt[u]==0)
                {
                        vis[u]=0;
                        u=pre[u];
                        cnt[u]--;
                }
        }
        return 0;
}
//=======================Bellman-Ford 算法求负环，时间复杂度（  ）================
int dis[MAXN],vis[MAXN];
int bellman_ford(int s){
        memset(dis,0x7f,sizeof(dis));
        memset(vis,0,sizeof(vis));
        dis[s]=0;
        vis[s]=1;
        for(int i=1;i<n;i++)
        {
                bool flag=1;
                for(int j=1;j<=tot;j++)//tot:边数 if(dis[e[j].v]>dis[e[j].u]+e[j].w)
                        {
                                dis[e[j].v]=dis[e[j].u]+e[j].w;
                                vis[e[j].v]|=vis[e[j].u];
                                flag=0;
```

```
                        }
                if(flag)
                        return 0;
        }
        for(int j=1;j<=tot;j++)
                if(dis[e[j].v]>dis[e[j].u]+e[j].w)
                {
                        dis[e[j].v]=dis[e[j].u]+e[j].w;
                        if(vis[e[j].u])
                                return 1;
                }
        return 0;
}
//=========================Floyd 判断负环  O( n ^ 3 )============================
bool check(){
        for (register int k = 1; k <= n; k++)
                for (register int i = 1; i <= n; i++)
                {
                        for (register int j = 1; j <= n; j++)
                        {
                                int res = dis[i][k] + dis[k][j];
                                if (dis[i][j] > res)
                                        dis[i][j] = res;
                        }
                        if (dis[i][i] < 0)
                                return 1;
                }
        return 0;
}
int main(){
        T = read();
        while (T--)
        {
                n = read(), m = read(), W = read();
                for (register int i = 1; i <= n; i++)
                        for (register int j = 1; j <= n; j++)
                                dis[i][j] = INF;
                for (int i = 1; i <= m; i++)
                {
                        int u = read(), v = read(), w = read();
                        if (dis[u][v] > w)
                                dis[u][v] = dis[v][u] = w;
                }
                for (int i = 1; i <= W; i++)
                {
                        int u = read(), v = read(), w = read();
```

```
141                         dis[u][v] = -w;
142                 }
143                 if (check())
144                         printf("YES\n");
145                 elseprintf("NO\n");
146         }
147         return 0;
148 }
```

最短路算法

次短路：大于最短路的最小路径；

最短路：求法很多，不说了

最短路计数：计算最短路的数量，以dijstra算法解最短路来说，多开一个计数的数组num[N]，当我们跟新某一个点时，dis[u]>dis[v]+e[u][v] (u点与v点有路)，更新num[u]=num[v]；如果dis[u]=dis[v]+e[u][v],那就单独更新数量：num[u]+=num[v];

次短路算法：在最短路的基础上，次短路可以由次短路+边更新，也可以由最短路+边 跟新，这里注意一点，因为次短路更新时也会对其它次短路产生影响，所以更新次短路时也需要入队，我们先尝试更新最短路，成功的话就把原来的最短路给次短路，不成功的话就单独尝试更新次短路；

次短路数量：和最短路计数差不多，优先更新最短路，最短路更新成功时，次短路的数量就是原来最短路的数量，不成功的话就单独更新次短路，方法和最短路相同，为了知道我们每次取出的点是最短路的还是次短路的，所以有必要在队列里记录一样。

```
1  //方法一
2
3  typedef long long LL;
4  const int Maxn = 1e5 + 7;
5  const int Inf = 1e9 + 7;
6  int N , M;
7  int dis1[Maxn] , dis2[Maxn];
8  struct node{
9          int v , w;
10         friend bool operator < (node a , node b){
11                 return a.w > b.w;
12         }
13 };
14 vector <node> G[Maxn];
15 void Dijkstra(){
16         priority_queue <node> que;
17         fill(dis1 , dis1+N+1 , Inf);
18         fill(dis2 , dis2+N+1 , Inf);
19         int start = 1;
20         dis1[start] = 0;
21         que.push((node){start , 0});
22         node q;
23         int v , w;
24         while(!que.empty()){
25                 q = que.top();          que.pop();
```

```
                v = q.v , w = q.w;
                if(dis2[v] < w)         continue;
                int to_v , to_w;
                for(int i = 0 ; i < G[v].size() ; i++){
                        to_v = G[v][i].v , to_w = G[v][i].w + w;
                        if(dis1[to_v] > to_w){
                                que.push((node){to_v , to_w});
                                swap(dis1[to_v] , to_w);
                        }
                        if(dis2[to_v] > to_w && dis1[to_w] < to_w){
                                dis2[to_v] = to_w;
                                que.push((node){to_v , to_w});
                        }
                }
        }

}

int main()
{
        while(~scanf(" %d %d",&N,&M)){
                for(int i = 1 ; i <= M ; i++){
                        int u , v , w;          scanf(" %d %d %d",&u,&v,&w);
                        G[u].push_back((node){v,w});
                        G[v].push_back((node){u,w});
                }
                Dijkstra();
                printf("%d\n",dis2[N]);
        }

}
//方法二

typedef pair<int,int>PII;
const int N=1010,M = 10010;
int n,m,s,f;
int head[N],nextt[M*2],to[M*2],len[M*2],cnt;
int dis[N][2],num[N][2],vis[N][2];

void add(int u,int v,int w){
    to[cnt]=v,len[cnt]=w,nextt[cnt]=head[u],head[u]=cnt++;
}
struct node{
    int to,dis,kind;
    bool operator <(const node a)const{
        return a.dis<dis;
    }
```

```
};

void dijstra(){
    memset(num,0,sizeof(num));
    memset(vis,0,sizeof(vis));
    memset(dis,0x3f,sizeof(dis));
    dis[s][0]=0,num[s][0]=1;
    priority_queue<node>q;
    q.push({s,dis[s][0],0});
    while(q.size()){
        int u=q.top().to,ds=q.top().dis,kind=q.top().kind;q.pop();
        if(vis[u][kind])continue;
        vis[u][kind]=1;
         for(int i=head[u];i;i=nextt[i]){
            int v=to[i];
            if(dis[v][0]>dis[u][kind]+len[i]){
                dis[v][1]=dis[v][0];num[v][1]=num[v][0];
                q.push({v,dis[v][1],1});
                dis[v][0]=dis[u][kind]+len[i];
                num[v][0]=num[u][kind];
                q.push({v,dis[v][0],0});
            }
            else if(dis[v][0]==dis[u][kind]+len[i]){
                num[v][0]+=num[u][kind];
            }
            else if(dis[v][1]>dis[u][kind]+len[i]){
                dis[v][1]=dis[u][kind]+len[i];
                num[v][1]=num[u][kind];
                q.push({v,dis[v][1],1});
            }
            else if(dis[v][1]==dis[u][kind]+len[i]){
                num[v][1]+=num[u][kind];
            }
         }
    }
}

int main(){
    int t; cin>>t;
    while(t--){
        cin>>n>>m;
        cnt=1;
        memset(head,0,sizeof(head));
        for(int i=1;i<=m;i++){
            int u,v,w;
            cin>>u>>v>>w;
            if(u==v)continue;
```

```
120             add(u,v,w);
121         }
122         cin>>s>>f;
123         dijstra();
124         // cout<<dis[f]<<" "<<dis1[f]<<" \n";
125         // cout<<num[f]<<" "<<num1[f]<<" \n";
126         cout<<(dis[f][1]==dis[f][0]+1?num[f][1]+num[f][0]:num[f][0])<<endl;
127     }
128     return 0;
129 }
```

数论

Lucas定理（用于n m可能大于p的组合数求解）

正常的组合数运算可以通过递推公式求解（详见 排列组合），但当问题规模很大，而模数是一个不大的质数的时候，就不能简单地通过递推求解来得到答案，需要用到 Lucas 定理。 p的范围不能够太大，一般在1e5左右。

```
long long Lucas(long long n, long long m, long long p) {
  if (m == 0) return 1;
  return (C(n % p, m % p, p) * Lucas(n / p, m / p, p)) % p;
}
```

exLucas 定理

Lucas 定理中对于模数 P 要求必须为素数，那么对于 P 不是素数的情况，就需要用到 exLucas 定理。

求逆元的方法

- 当p是质数 若▯为素数，▯为正整数，且▯、▯互质。 则有▯^▯−1 ≡ 1 (mod▯)

```
const int P=998244353;
int qpow(ll a,ll p){
    ll r=1;
    for(;p;p>>=1,a=a*a%P) if(p&1) r=r*a%P;
    return r;
}
int inv(ll a){
```

```
    return qpow(a,P-2);
}

```

- 欧几里得 这个x 就是要的逆元　当 ⊠⊥⊠（互质），但 ⊠ 不是质数的时候也可以使用

```
void Exgcd(ll a, ll b, ll &x, ll &y) {
    if (!b) x = 1, y = 0;
    else Exgcd(b, a % b, y, x), y -= a / b * x;
}
int main() {
    ll x, y;
    Exgcd (a, p, x, y);
    x = (x % p + p) % p;
    printf ("%d\n", x); //x是a在mod p下的逆元
}
```

- 递推 只适用于模数为质数的情况

```
inv[1] = 1;
for(int i = 2; i < p; ++ i)
    inv[i] = (p - p / i) * inv[p % i] % p;
```

- 线性求任意 n 个数的逆元

$$\text{求任意给定 } n \text{ 个数 } (1 \le a_i < p) \text{ 的逆元}$$

```
s[0] = 1;
for (int i = 1; i <= n; ++i) s[i] = s[i - 1] * a[i] % p;
sv[n] = qpow(s[n], p - 2);
// 当然这里也可以用 exgcd 来求逆元,视个人喜好而定.
for (int i = n; i >= 1; --i) sv[i - 1] = sv[i] * a[i] % p;
for (int i = 1; i <= n; ++i) inv[i] = sv[i] * s[i - 1] % p;
```

求组合数的方法

- a 较小的情况下，如 a <= 1e3 然后mod不是质数。

```
#define MAX 4000
#define MOD 6662333

int C[MAX][MAX], n;

void solve(void)
{
    for (int i = 0; i <= n; i++)
        for (int j = 0; j <= i; j++)
            if (j == 0)
                C[i][j] = 1;
            else
                C[i][j] = (C[i - 1][j] + C[i - 1][j - 1]) % MOD;
}
```

- *a* 较大的情况下，如 a <= 1e5 适用于mod是质数的情况

预处理后，C_n^k = fact[n] * infact[k] * infact[n - k]。注意：计算过程中可能会溢出，要进行模运算。

```
#define MAX 4000
#define MOD 6662333

int fact[MAX], infact[MAX], n;

typedef long long ll;

int qum(int x, int n, int mod)
{
    ll ret = 1;
    while (n > 0)
    {
        if (n & 1)
            ret = ret * x % MOD;
        x = (ll)x * x % MOD;
        n >>= 1;
    }
    return ret;
}

void solve(void)
{
    fact[0] = infact[0] = 1;
    for (int i = 1; i <= n; i++)
    {
        fact[i] = (ll)fact[i - 1] * i % MOD;
```

```
        infact[i] = (ll)infact[i - 1] * qum(i, MOD - 2, MOD) % MOD;
    }
}
```

- Lucas定理

核心：卢卡斯定理

$$C_a^b \equiv C_{a \bmod p}^{b \bmod p} \cdot C_{\lfloor a/p \rfloor}^{\lfloor b/p \rfloor} \pmod p$$

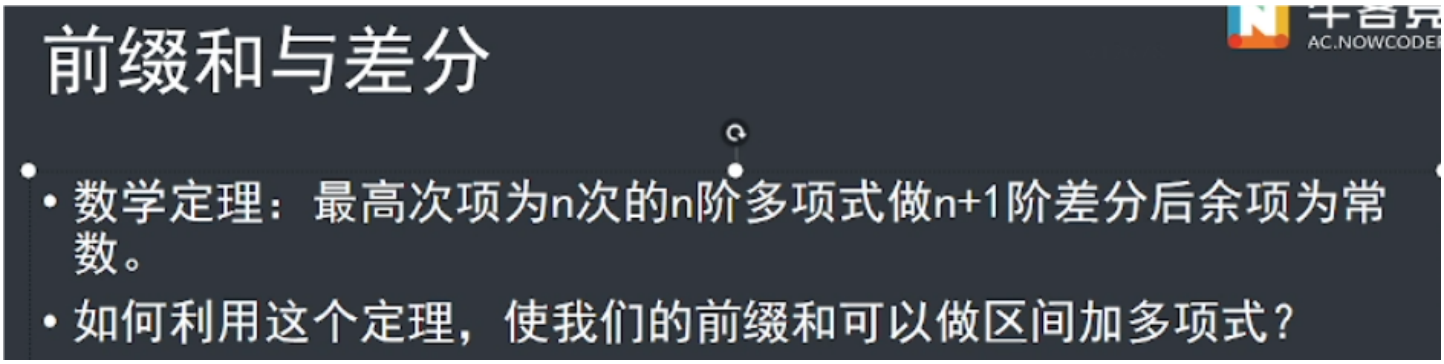
适用范围：a 很大的情况，比如 $a \leq 10^{18}$。
算法简析：若 a 和 b 很大，我们可以通过卢卡斯定理缩小 a 和 b，直至 a，$b < p$ (p 一般是较小的素数)。这时，在使用前两种方法求解。

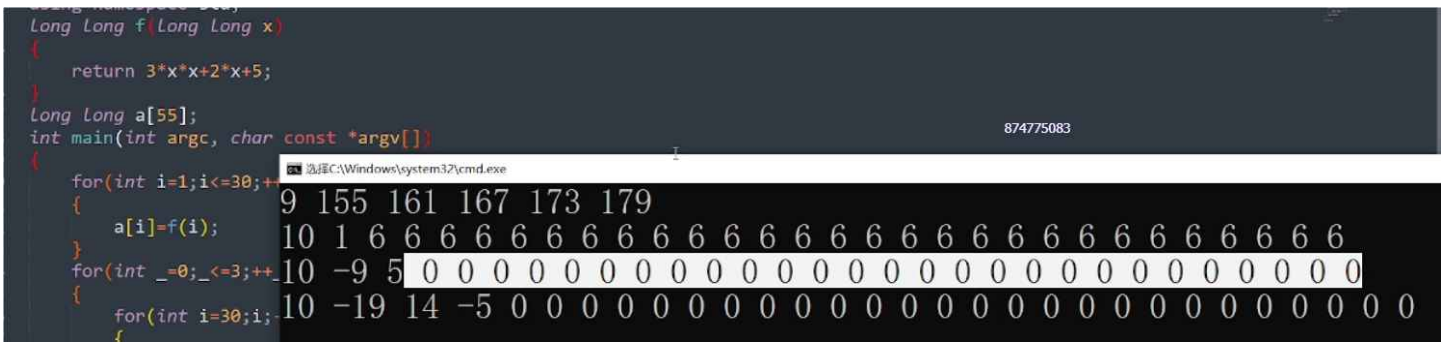
```
#define MAX 4000
#define MOD 6662333

int fact[MAX], n;

typedef long long ll;

int qum(int x, int n, int mod)
{
    ll ret = 1;
    while (n > 0)
    {
        if (n & 1)
            ret = ret * x % MOD;
        x = (ll)x * x % MOD;
        n >>= 1;
    }
    return ret;
}

void init(void)
{
    fact[0] = 1;
    for (int i = 1; i <= n; i++)
    {
        fact[i] = (ll)fact[i - 1] * i % MOD;
    }
}

int C(int a, int b, int mod)
{
```

```
    if (a < b)
        return 0;
    return (ll)fact[a] * qum(fact[b], mod - 2, mod) % mod * qum(fact[a - b],
mod - 2, mod) % mod;
}

int lucas(int a, int b, int mod)
{
    if (a < mod && b < mod)
        return C(a, b, mod);
    return (ll)C(a % mod, b % mod, mod) * lucas(a / mod, b / mod, mod) % mod;
}
```

原数组是多项式 怎么利用差分操作

$$式f(x) = c_0x^k + c_1x^{k-1} + ... + c_{k-1}x^1 + c_k,$$





欧拉函数

欧拉函数（x）定义为小于x但与x互质的正整数数量，比如f（12）=4（其中1，5，7，11互质），特别的，规定f（1）=1

对于欧拉函数，主要有如下性质：

- 若 p 是质数，则 $\varphi(p^n) = p^{n-1}(p-1)$
- 若 $a|x$ ，则 $\varphi(ax) = a\varphi(x)$
- 若 a, b 互质，则 $\varphi(a)\varphi(b) = \varphi(ab)$

```
//求解欧拉函数值    最坏复杂度：根号n
int phi(int n)
{
    int res = n;
    for (int i = 2; i * i <= n; i++)
    {
        if (n % i == 0)
            res = res / i * (i - 1); // 先除再乘防止溢出
        while (n % i == 0) // 每个质因数只处理一次，可以把已经找到的质因数除干净
            n /= i;
    }
    if (n > 1)
        res = res / n * (n - 1); // 最后剩下的部分也是原来的n的质因数
    return res;
}
```

欧拉筛（线性筛）

```
//时间复杂度 on
void init(int n)
{
    phi[1] = 1;
    for (int i = 2; i <= n; i++)
    {
        if (!isnp[i])
            primes.push_back(i), phi[i] = i - 1; // 性质一，指数为1的情形
        for (int p : primes)
        {
            if (p * i > n)
                break;
            isnp[p * i] = 1;
            if (i % p == 0)
            {
                phi[p * i] = phi[i] * p; // 性质二
                break;
```

```
            }
            elsephi[p * i] = phi[p] * phi[i]; // 这时肯定互质，用性质三
        }
    }
```

杂项

A*求第k短路

给定一张 N个点（编号 1,2…N1,2…），M 条边的有向图，求从起点 S 到终点 T的第 K 短路的长度，路径允许重复经过点或边。

注意： 每条最短路中至少要包含一条边。

输入格式

第一行包含两个整数 N 和 M。

接下来 M 行，每行包含三个整数 A,B和 L，表示点 A 与点 B 之间存在有向边，且边长为 L。

最后一行包含三个整数 S,T 和 K，分别表示起点 S，终点 T 和第 K短路。

输出格式

输出占一行，包含一个整数，表示第 K 短路的长度，如果第 K 短路不存在，则输出 −1−1。

数据范围

1≤S,T≤N≤1000 0≤M≤104 1≤K≤1000 1≤L≤100

```
#define x first
#define y second

using namespace std;

typedef pair<int, int> PII;
typedef pair<int, PII> PIII;
const int N = 1010, M = 200010;

int n, m, S, T, K;
int h[N], rh[N], e[M], w[M], ne[M], idx;
int dist[N], cnt[N];
bool st[N];

void add(int h[], int a, int b, int c)
{
```

```
    e[idx] = b;
    w[idx] = c;
    ne[idx] = h[a];
    h[a] = idx++;
}

void dijkstra()
{
    priority_queue<PII, vector<PII>, greater<PII>> heap;
    heap.push({0, T}); // 终点
    memset(dist, 0x3f, sizeof dist);
    dist[T] = 0;

    while (heap.size())
    {
        auto t = heap.top();
        heap.pop();

        int ver = t.y;
        if (st[ver])
            continue;
        st[ver] = true;

        for (int i = rh[ver]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (dist[j] > dist[ver] + w[i])
            {
                dist[j] = dist[ver] + w[i];
                heap.push({dist[j], j});
            }
        }
    }
}

int astar()
{
    priority_queue<PIII, vector<PIII>, greater<PIII>> heap;
    // 谁的d[u]+f[u]更小 谁先出队列
    heap.push({dist[S], {0, S}});
    while (heap.size())
    {
        auto t = heap.top();
        heap.pop();
        int ver = t.y.y, distance = t.y.x;
        cnt[ver]++;
        // 如果终点已经被访问过k次了 则此时的ver就是终点T 返回答案
```

```
64
65          if (cnt[T] == K)
66              return distance;
67
68          for (int i = h[ver]; i != -1; i = ne[i])
69          {
70              int j = e[i];
71              if (cnt[j] < K)
72                  heap.push({distance + w[i] + dist[j], {distance + w[i], j}});
73          }
74      }
75      // 终点没有被访问k次
76      return -1;
77  }
78
79  int main()
80  {
81      cin >> m >> n;
82      memset(h, -1, sizeof h);
83      memset(rh, -1, sizeof rh);
84      for (int i = 0; i < n; i++)
85      {
86          int a, b, c;
87          cin >> a >> b >> c;
88          add(h, a, b, c);
89          add(rh, b, a, c);
90      }
91      cin >> S >> T >> K;
92      // 起点==终点时 则d[S→S] = 0 这种情况就要舍去 ,总共第K大变为总共第K+1大
93      if (S == T)
94          K++;
95      // 从各点到终点的最短路距离 作为估计函数f[u]
96      dijkstra();
97      cout << astar();
98      return 0;
99  }
```

如何快速筛选出需要的边，并且不多枚举

由于存在类菊花图的可能，所以考虑如何降低，边数多的点所存的边 考虑三元环计数的方式，将边存成单向边，那么对于每一个点的边数， 都会被降低到$\sqrt{m}$

```
1  // u v w
2  for (int i = 1; i <= m; i++)
3  {
```

```
    cin >> ar[i][0] >> ar[i][1] >> ar[i][2];
    d[ar[i][0]]++;
    d[ar[i][1]]++;
}

for (int i = 1; i <= m; i++)
{
    auto &[x, y, z] = ar[i];

    if (d[x] > d[y] || (d[x] == d[y] && x > y))
        swap(x, y);

    g[x].push_back({y, z});
}
//枚举顶点的边可以进行下面的操作
//cnt个顶点 然后操作完 tmp数组 就是cnt个顶点的所有边
//st保存了顶点
//每个顶点最多根号级别个边
int cnt;
cin >> cnt;

vector<int> a;
set<int> st;

for (int i = 1; i <= cnt; i++)
{
    int idx;
    cin >> idx;
    a.push_back(idx);
    st.insert(idx);
}

vector<array<int, 3>> tmp;

for (int i = 0; i < cnt; i++)
{
    int idx = a[i];

    for (int j = 0; j < g[idx].size(); j++)
    {
        if (st.count(g[idx][j].first))
        {
            tmp.push_back({g[idx][j].second, a[i], g[idx][j].first});
        }
    }
}
```

MEX的各种奇怪东西

不涉及修改数组的话，维护前缀mex和后缀mex

注意到隐藏在数据范围中的关键条件：$0 \le a_i \le n$。

由此，就可以求出 a 数组的前缀、后缀 MEX。

考虑用 set 实现，预先添加 $0 \sim n$ 的数字到 set 中，每出现一个 a_i，就从 set 中把 a_i 删除（注意每个相同的 a_i 只能删一次）。当前每个位置的 MEX 就是 set 中最小的数。

```
    int a[n + 1] = {};
    int pre[n + 1] = {};
    int sub[n + 1] = {};

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

    set<int> s1, s2;

    for (int i = 0; i <= n; i++)
    {
        s1.insert(i);
        s2.insert(i);
    }

    for (int i = 1; i <= n; i++)
    {
        s1.erase(a[i]);
        pre[i] = *s1.begin();
    }

    for (int i = n; i >= 1; i--)
    {
        s2.erase(a[i]);
        sub[i] = *s2.begin();
    }
```

结论

1. 给你一棵树，每次选一条路径覆盖，最少多少次可以覆盖整棵树 （叶子节点 / 2 上取整）

2. n是12附近 反正不大 状态压缩的时候 如果想枚举S（也就是当前状态的）的子集 比如 1101 其中有一个子集是1100 可以通过 **for(int p = s; p != 0; p = (p - 1) & s)** (感性上理解就是 每次p的末尾肯定是 1 0000的形式 具体就是几个0的区别而已 可能没有0 可能有多个0　-1　就变成 0 1111　那这样子做到的效果就是 把最后一个1取掉 然后不影响后面末尾 因为都提供的1 去和原来的s & 肯定能保留下来末尾的元素)
3. 若一张有向完全图存在环，则一定存在三元环
4. 有向图最少加max(p, q)条边变成强连通分量 （p为出度为0的点的数量，q为入度为0的点的数量）
5. 无向图最少加（cnt + 1）/ 2 下取整 （cnt为缩完点之后度数为1的点的个数）
6. 数组如果想要做出等差的效果 只要对原数组的两次差分即可
7. 数组如果想要做出1 4 9 16的效果 只要对原数组的三次差分即可
8. 一个二分图中的最大匹配数等于这个图中的最小点覆盖数。

知识点（概念）

1. 增广路：就是一条从起点到终点的路径
2. 最大流最小割定理：

   ：在一个流网络中，最大流量等于最小割的容量。换句话说，从源节点到汇节点能够达到的最大流量，恰好等于阻断所有流量的最小割的边容量之和（感性理解：就是一个图是 最大流量取决于最弱的那几个变容量和，相当于一座桥，物品的传输上线就是桥的上线，最大流量也就是最小割的上限）
3. 最小点覆盖：找到最少的一些点，使二分图所有的边都至少有一个端点在这些点之中。也就是说，删除包含这些点的边，可以删掉所有的边。
4. 最大匹配问题：就是假如你是红娘，可以撮合任何一对有暧昧关系的男女，那么你最多能成全多少对情况（在二分图中最多能找到多少条，在这些边中，边和边之间没有公共端点的边）
5. 最小费用最大流：就是 最大流的走法不唯一，但是现在变成了如下，最小费用最大流就是求这个的

现在网络上的每条边，除了容量外，还有一个属性：**单位费用**。一条边上的费用等于流量×单位费用。我们知道，网络最大流往往可以用多种不同的方式达到，所以现在要求：在保持流最大的同时，找到总费用最少的一种。

如下图，有多种方式可以达到最大流3，但是S->3->T (2) + S->3->2->T (1)这种流法的费用是7×2+5×1=**19**，而S->3->T (2) + S->1->2->T (1)这种流法的费用则是7×2+4×1=**18**，后者比前者的费用更低。事实上，后者正是这个网络的最小费用最大流。