1.5 图论

差分约束

- 对于 n 个变量 $x_1, x_2, \ldots, x_n$ 和 m 个约束条件 $x_i \leq x_j + c_k$，我们需要求出一组解或判断出无解。
- 若求的是 x_i 的最大值，不难发现 $x_i \leq x_j + c_k$ 与单源最短路中的三角形不等式 $dis[y] \leq dis[x] + z$ 极其相似，可从点 j 向点 i 连一条权为 c_k 的边，无解即存在负环。
- 若求的是 x_i 的最小值，可将约束条件变形为 $x_j \geq x_i - c_k$，与单源最长路中的不等式 $dis[y] \geq dis[x] + z$ 极其相似，可从点 i 向点 j 连一条权为 $-c_k$ 的边，无解即存在正环。
- 求判断正环或者负环可通过 SPFA 或 Bellman-Ford 算法实现，具体若起始点到该点最短路径长度大于等于 n 则无解。

```
inline bool SPFA(int src)
{   // 从 src 开始找负环，注意队列大小要比 nm 稍大
    for (int i = 1; i <= n; ++i)
        dis[i] = Maxn, cnt[i] = 0, vis[i] = false;
        //vis 必须要清空，中间可能 return
    dis[que[qr = 1] = src] = 0;
    for (int i = 1, x, y; i <= qr; ++i)
    {
        vis[x = que[i]] = false;
        for (arc *e = adj[x]; e; e = e->nxt)
            if (dis[y = e->to] > dis[x] + e->cst)
            {
                dis[y] = dis[x] + e->cst;
                cnt[y] = cnt[x] + 1;
                if (cnt[y] >= n) return true;
                if (!vis[y])
                    vis[que[++qr] = y] = true;
            }
    }
    return false;
}
```

强连通分量

- $dfn[x]$ 为结点 x 搜索的时间次序。
- $low[x]$ 为 u 或 u 的子树（经过最多一条后向边或栈中横叉边）能够回溯到的最早的栈中结点的编号。
- 由定义可以得出：

$$
\begin{aligned}
&low[x] = \min \\
&\{ \\
&dfn[x] \\
&low[y], (x,y)\text{为树枝边，}x\text{为}y\text{的父结点} \\
&dfn[y], (x,y)\text{为后向边或指向栈中结点的横叉边} \\
&\}
\end{aligned}
$$

```cpp
inline void Tarjan(int x)
{
    dfn[x] = low[x] = ++tis;
    stk[++top] = x;
    ins[x] = true;
    int y;
    for (arc *e = adj[x]; e; e = e->nxt)
        if (!dfn[y = e->to])
        {
            Tarjan(y);
            CkMin(low[x], low[y]);
        }
        else if (ins[y])
            CkMin(low[x], dfn[y]);
    if (dfn[x] == low[x])
    {
        ++C;
        do
        {
            y = stk[top--];
            ins[y] = false;
            col[y] = C;
        }while (y != x);
    }
}
```

桥

- 无向图中 $low[x]$ 为 x 或 x 的子树经过最多一条后向边能够追溯到的树中结点的次序号。
- 根据定义，有：

> $low[x] = \min$
> $\{$
> $dfn[x]$
> $dfn[y], (x,y)$为后向边
> $low[y], (x,y)$为树枝边
> $\}$

- 桥 (x,y) 的判断条件：(x,y) 为树枝边且 $dfn[x] < low[y]$。
- 将桥标记后用并查集确定边双连通分量。
- **结论** 对于一个有桥的连通图，求加最少数量的边，使其变为边双连通图。用 Tarjan 求出边双，将边双缩为一点，则原图变为一颗树。记这棵无根树中叶子结点的个数为 $leaf$，若 $leaf = 1$，无需加边，否则所加最少边数为 $\lfloor \frac{leaf+1}{2} \rfloor$。

> **证明** 可归纳证明，每次优先选择路径上有至少两个支链的叶子结点合并。

```cpp
inline void Tarjan(int x)
{
    dfn[x] = low[x] = ++tis;
    int y;
    for (int e = adj[x]; e; e = nxt[e])
    {
        if (e == (up[x] ^ 1)) //树枝边的反向边要注意判断
            continue;
        if (!dfn[y = to[e]])
        {
```

```
            up[y] = e;
            Tarjan(y);
            CkMin(low[x], low[y]);
            if (dfn[x] < low[y])
                bridge[e] = bridge[e ^ 1] = true;
        }
        else
            CkMin(low[x], dfn[y]);
    }
}
```

割点

- $low[x]$ 定义同上。
- x 为割点的判断条件：

1. x 为树根，且 x 有多于一个的子树。
2. x 不为树根，(x,y) 为树枝边且 $dfn[x] \leq low[y]$。

- 在求割点的过程中就能顺便求出点双连通分量。
- 在搜索图时，每找到一条树枝边或后向边（注意实现时后向边的反向边不应加入栈中），就把这条边加入栈中。若某点 x 满足 (x,y) 为树枝边且 $dfn[x] \leq low[y]$，把边从栈顶一个个取出，直到遇到了边 (x,y)，取出的这些边与其相连的点，组成一个点双连通分量。
- 与求割点不同，求点双时并不需要判断树根，方便将所有点双取出。

典例 POJ2942

题目大意

- 给定 n 个骑士和 m 对厌恶关系，一个骑士能够不被驱逐当且仅当存在包含他的奇数个骑士，使得他们围坐一桌时任意的相邻两个骑士均不存在厌恶关系。
- 求必须被驱逐的骑士数量，$n \leq 10^3, m \leq 10^6$。

解法

- 以下的奇环和偶环均指简单环。
- 建出原图的补图，则一名骑士能够参加会议当且仅当他在图中的一个奇环上。
- 易知奇环一定只在某个点双内部。
- **结论** 若某个点双内存在奇环，则对于该点双内所有点，都存在某个奇环，使得该点在该奇环上。

> **证明** 若某点在偶环上，则一定存在一个偶环与已知的奇环有公共边，则可将偶环和已知的奇环合并得到一个新的奇环。

- 用二分图染色判断每个点双中是否存在奇环即可。
- 完整代码：

```
#include <cstdio>
#include <iostream>

template <class T>
inline void read(T &res)
{
    char ch;
    while (ch = getchar(), !isdigit(ch));
    res = ch ^ 48;
    while (ch = getchar(), isdigit(ch))
        res = res * 10 + ch - 48;
```

```cpp
}

template <class T>
inline void CkMin(T &x, T y) {x > y ? x = y : 0;}

const int N = 1e3 + 5;
const int M = 2e6 + 5;

int fa[N], col[N], dfn[N], low[N], stkx[M], stky[M];
int tis, n, m, top;
bool edge[N][N], inc[N], ans[N];

struct arc
{
    int to;
    arc *nxt;
}p[M], *adj[N], *T = p;

inline void linkArc(int x, int y)
{
    (++T)->nxt = adj[x]; adj[x] = T; T->to = y;
    (++T)->nxt = adj[y]; adj[y] = T; T->to = x;
}

inline bool dfsColoring(int x)
{
    for (arc *e = adj[x]; e; e = e->nxt)
    {
        int y = e->to;
        if (col[y] == -1)
        {
            col[y] = col[x] ^ 1;
            if (!dfsColoring(y))
                return false;
        }
        else if (col[y] == col[x])
            return false;
    }
    return true;
}

inline void solvePBC(int x, int y)
{
    T = p;
    for (int i = 1; i <= n; ++i)
        adj[i] = NULL;
    int u, v;
    do
    {
        u = stkx[top], v = stky[top];
        linkArc(u, v);
        inc[u] = inc[v] = true;
        --top;
    }while (x != u || y != v);

    for (int i = 1; i <= n; ++i)
        col[i] = -1;
    col[x] = 0;
```

```
70
71      if (!dfsColoring(x))
72      {
73          for (int i = 1; i <= n; ++i)
74              if (inc[i])
75                  ans[i] = true;
76      }
77
78      for (int i = 1; i <= n; ++i)
79          inc[i] = false;
80  }
81
82  inline void Tarjan(int x)
83  {
84      dfn[x] = low[x] = ++tis;
85      for (int y = 1; y <= n; ++y)
86      {
87          if (y == fa[x] || !edge[x][y])
88              continue;
89          if (dfn[y] < dfn[x]) // 包含条件 !dfn[y]
90          {
91              ++top;
92              stkx[top] = x;
93              stky[top] = y;
94          }
95          if (!dfn[y])
96          {
97              fa[y] = x;
98              Tarjan(y);
99              CkMin(low[x], low[y]);
100             if (dfn[x] <= low[y])
101                 solvePBC(x, y);
102         }
103         else
104             CkMin(low[x], dfn[y]);
105     }
106 }
107
108 int main()
109 {
110     while (1)
111     {
112         read(n); read(m);
113         if (!n && !m)
114             break ;
115         for (int i = 1; i <= n; ++i)
116             ans[i] = false;
117         for (int i = 1; i <= n; ++i)
118             for (int j = 1; j <= n; ++j)
119                 edge[i][j] = i == j ? false : true;
120         for (int i = 1, x, y; i <= m; ++i)
121         {
122             read(x); read(y);
123             edge[x][y] = edge[y][x] = false;
124         }
125         tis = top = 0;
126         for (int i = 1; i <= n; ++i)
127             dfn[i] = low[i] = fa[i] = 0;
```

```
        for (int i = 1; i <= n; ++i)
            if (!dfn[i])
                Tarjan(i);
        int fans = 0;
        for (int i = 1; i <= n; ++i)
            fans += !ans[i];
        printf("%d\n", fans);
    }
    return 0;
}
```

DAG 上支配树

- 每个点的支配点为 DAG 上所有入点在支配树上对应结点的 LCA 及其祖先，表示从某一入度为 0 的点出发到达该点必须经过的结点。
- 为方便处理，可新建一个超级源点连向所有入度为 0 的点。

圆方树

- 对每个点双连通分量建一个新点（方点），每个点（圆点）向其所属的点双连边，形成树结构，**树中点数至多是原图的两倍。**
- 将方点的权值设为对应点双的大小，将每个圆点的权值设为 -1，则两圆点在圆方树上的路径点权和，恰好等于原图中两点之间所有简单路径的并集大小减 2。
- 给定一个点集，若需求任意两点间所有简单路径的并集的圆点点权最值，另设方点的权值为其子结点（必为圆点）的点权最值，圆点的权值为 0，设点集中的点按照圆方树上 DFS 序排序得到的结果为 $x_1, x_2, \ldots, x_k$。。
  - 设 $\mathrm{mx}(x, y)$ 表示圆方树上 x 到 y 的点权最值，若 $\mathrm{LCA}(x_1, x_k)$ 为方点，答案为 $\max / \min\{\mathrm{mx}(x_1, x_k), \max\limits_{i=1}^{k-1} / \min\limits_{i=1}^{k-1}\{\mathrm{mx}(x_i, x_{i+1})\}\}$。
  - 若 $\mathrm{LCA}(x_1, x_k)$ 为圆点，答案还要再和其父结点的点权取最值。
  - 这样设权值也很方便修改。
- **仙人掌** 任意一条边至多只出现在一条简单回路的无向连通图。
- **仙人掌上两点间最短路** 将某个圆点连向方点的边权记为圆点到方点父结点的最短路（位于同一个简单回路中，可预处理）。
  - 若两点 x, y 在圆方树上的 LCA 为圆点，直接求边权和。
  - 若两点 x, y 在圆方树上的 LCA 为方点 z，通过倍增找到 x 到 z 路径上与 z 相邻的点 u，y 到 z 路径上与 z 相邻的点 v，求 x 到 u 和 y 到 v 的边权和，再加上 u 到 v 在简单回路上的最短路。

2-SAT

- 2-SAT 问题指的是解下列形式的布尔方程：

$$(a \vee b) \wedge (c \vee d) \wedge (e \vee f) \wedge \ldots$$

- 其中 $a, b, c, \ldots$ 称为文字，是一个布尔变量或其否定。
- 利用 $\Rightarrow$（蕴含） 将每个子句 $(a \vee b)$ 写成等价形式 $(\neg a \Rightarrow b \wedge \neg b \Rightarrow a)$，对每个布尔变量 x 构造两个顶点 x 和 $\neg x$，以 $\Rightarrow$ 关系为边建立有向图。
- 若存在 x 和 $\neg x$ 在同一强连通分量内，则无解。
- 对强连通分量缩点后的图求拓扑序，则若 x 所在的强连通分量的拓扑序在 $\neg x$ 所在的强连通分量的拓扑序之后，则 x 为真，否则 $\neg x$ 为真。

- 强连通分量的编号即为逆拓扑序。
- 常见的等价形式转换：
  - $(x=1)\Leftrightarrow(\neg x\Rightarrow x),(x=0)\Leftrightarrow(x\Rightarrow\neg x)$
  - $\neg(a\wedge b)\Leftrightarrow(a\Rightarrow\neg b\wedge b\Rightarrow\neg a)$
  - k 个点中至多选一个，令这 k 个点分别为 $a_1,a_2,\dots a_k$，新建 $2k$ 个点，pre_i 表示 $[1,i]$ 均不选，suf_i 表示 $[i,k]$ 均不选，作如下连边：
    - $a_i\Rightarrow pre_{i-1},a_i\Rightarrow suf_{i+1}$
    - $pre_i\Rightarrow pre_{i-1},pre_i\Rightarrow\neg a_i$
    - $suf_i\Rightarrow suf_{i+1},suf_i\Rightarrow\neg a_i$
- 上述算法只适用于判断可行性并给出一种可行方案。

欧拉回路

- 设图 $G=(V,E)$。
- **欧拉回路/路径** 图 G 中经过每条边一次并且仅一次的回路/路径。
- **欧拉图** 存在欧拉回路的图。
- **半欧拉图** 存在欧拉路径但不存在欧拉回路的图。
- **基图** 忽略有向图所有边的方向，得到的无向图。
- **定理1** 无向图 G 为欧拉图，当且仅当 G 为连通图且所有顶点的度为偶数。
- **定理2** 无向图 G 为半欧拉图，当且仅当 G 为连通图且除了两个顶点的度为奇数之外，其它所有顶点的度为偶数。
- **定理3** 有向图 G 为欧拉图，当且仅当 G 的基图连通，且所有顶点的入度等于出度。
- **定理4** 有向图 G 为半欧拉图，当且仅当 G 的基图连通，且存在顶点 x 的入度比出度大 1、y 的入度比出度小 1，其它所有顶点的入度等于出度。
- 求欧拉图 G 的欧拉回路：

```
inline void findCircuit(int x)
{
    for (int &e = adj[x]; e; e = nxt[e])
        if (!vis[e])
        {
            int c = e;
            vis[c] = true;
            if (type & 1) // type = 1 为无向图，type = 0 为有向图
                vis[c ^ 1] = true;
            findCircuit(to[c]);
            stk[++top] = c;
        }
}

```

- 若题目要求用简单环覆盖图中所有边，则先求出欧拉回路，将欧拉回路上的边依次入栈，一旦入栈过程中发现有重点，则不断弹栈至重点处，则弹栈取出的所有边组成一个简单环，最终即可得到一个简单环的覆盖方案。

Prüfer 序列

- **对树建立 Prüfer 序列**
  - 每次选择一个编号最小的叶子结点删除，在序列中记录它连接的那个结点。
  - 重复 $n-2$ 次直至剩下两个结点，算法结束。
  - 线性实现上述过程只需用指针 p 记录当前编号最小的叶子结点，若删点后产生的新的叶子结点比 p 小则继续删除这个叶子结点不产生新的叶子结点或产生的叶子结点比 p 大。

- **Prüfer 序列的性质**
  - 构造完 Prüfer 序列原树剩下的两个结点之一一定是编号最大的结点 n。
  - 每个结点在序列中出现的次数是其度数减一，没有出现的就是叶子结点。
- **用 Prüfer 序列重建树**
  - 由 Prüfer 序列的性质还原出每个点的度数。
  - 依次枚举 Prüfer 序列上的点，选择一个度数为 1 且编号最小的结点与之连接，同时将两者的度数减一。
  - 重复 $n-2$ 次后只剩下两个度数为 1 的点，将它们建立连接，算法结束。
  - 线性实现上述过程同样是用指针 p 记录度数为 1 且编号最小的结点，具体做法类似。

```
inline void TreeToPrufer()
{
    for (int i = 1, x; i < n; ++i)
    {
        read(fa[i]);
        ++deg[fa[i]];
    }
    // 这里的 fa[i] 指以 n 为根时结点 i 的父结点
    int p = 1, x = 0;
    for (int i = 1; i <= n - 2; ++i)
        if (x && x < p)
        {
            ans[i] = fa[x];
            x = !--deg[fa[x]] ? fa[x] : 0;
        }
        else
        {
            while (deg[p])
                ++p;
            ans[i] = fa[p];
            x = !--deg[fa[p]] ? fa[p] : 0;
            ++p;
        }
}

inline void PruferToTree()
{
    for (int i = 1; i <= n - 2; ++i)
    {
        read(ans[i]);
        ++deg[ans[i]];
    }
    int p = 1, x = 0;
    for (int i = 1; i <= n - 2; ++i)
        if (x && x < p)
        {
            fa[x] = ans[i];
            x = !--deg[ans[i]] ? ans[i] : 0;
        }
        else
        {
            while (deg[p])
                ++p;
            fa[p] = ans[i];
            x = !--deg[ans[i]] ? ans[i] : 0;
            ++p;
```

```
        }
    for (int i = 1; i < n; ++i)
        if (!fa[i])
        {
            fa[i] = n;
            break ;
        }
}
```

- **Cayley 公式** 完全图 K_n 有 n^{n-2} 棵生成树。

> **证明** 由构造和还原过程可知，任意一个长度为 $n-2$、值域为 $[1,n]$ 的整数序列都可以通过 Prüfer 序列双射对应一个生成树。

- **结论1** n 个结点有标号有根树的数量为 n^{n-1}。
- **结论2** n 个结点的度数依次为 $d_1,d_2,\ldots,d_n$ 的无根树的数量为 $\binom{n-2}{d_1-1\ d_2-1\ \ldots\ d_n-1}=\frac{(n-2)!}{\prod\limits_{i=1}^{n}(d_i-1)!}$。
- **结论3** 把 n 个点划分为 k 个连通块，已知第 i 个连通块的内部连边情况和大小 a_i，包含所有连通块的生成树数量为 $n^{k-2}\prod\limits_{i=1}^{k}a_i$。

> **证明** 设将第 i 个连通块作为一个整体时的度数为 $d_i(d_i\geq 0)$，先不考虑由具体哪个内部结点连边，总方案数为：
>
> $$\sum_{\sum\limits_{i=1}^{k}d_i=2k-2}\binom{k-2}{d_1-1\ d_2-1\ \ldots\ d_k-1}\prod_{i=1}^{k}a_i^{d_i}$$
>
> 设 $e_i=d_i-1$，通过多元二项式定理进行代换，得到：
>
> $$\sum_{\sum\limits_{i=1}^{k}e_i=k-2}\binom{k-2}{e_1\ e_2\ \ldots\ e_k}\prod_{i=1}^{k}a_i^{e_i+1}=\left(\sum_{i=1}^{k}a_i\right)^{k-2}\prod_{i=1}^{k}a_i=n^{k-2}\prod_{i=1}^{k}a_i$$

Boruvka 算法

- Boruvka 算法是一种古老的求解最小生成树的算法。
- 初始时视 n 个点为 n 个连通块，每次遍历所有点和边，连接一个连通块中和其它连通块相连的最小的一条边，直到合并成一个连通块。
- 每次连通块个数至少减少一半，可用并查集实现，时间复杂度 $\mathcal{O}((n+m)\log n)$。

树哈希

- 给定 M 个无根树，按同构关系分成若干等价类，输出与每个树同构的树的最小编号。
- 树哈希及其换根的形式见代码。

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

using std::ios;
using std::cin;
using std::cout;
typedef unsigned long long ull;

const ull C = (ull)(1e17) + 13;
const int N = 60;
```

```cpp
const ull mask =
std::chrono::steady_clock::now().time_since_epoch().count();

ull shift(ull x)
{
    x ^= mask;
    x ^= x << 13;
    x ^= x >> 7;
    x ^= x << 17;
    x ^= mask;
    return x;
}

std::vector<int> e[N];
ull sub[N], root[N];
std::map<ull, int> trees;

void getSub(int x)
{
    sub[x] = C;
    for (int y : e[x])
    {
        getSub(y);
        sub[x] += shift(sub[y]);
    }
}

void getRoot(int x)
{
    for (int y : e[x])
    {
        root[y] = sub[y] + shift(root[x] - shift(sub[y]));
        getRoot(y);
    }
}

int main()
{
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    cout.tie(nullptr);

    int m;
    cin >> m;
    for (int t = 1; t <= m; t++)
    {
        int n, rt = 0;
        cin >> n;
        for (int i = 1; i <= n; i++)
        {
            int fa;
            cin >> fa;
            if (fa)
                e[fa].emplace_back(i);
            else
                rt = i;
        }
        getSub(rt);
```

```
        root[rt] = sub[rt];
        getRoot(rt);
        ull hash = C;
        for (int i = 1; i <= n; i++)
            hash += shift(root[i]);
        if (!trees.count(hash))
            trees[hash] = t;
        cout << trees[hash] << '\n';
        for (int i = 1; i <= n; ++i)
            e[i].clear();
    }
}
```

- 另一种更简单的方法是直接以树的重心为根 DP，因为树的重心不会超过两个，两棵树同构当且仅当重心数目相同且对应的哈希值相同。
- 上述所有方法在判断两棵树同构之前都应确保两棵树的结点数相同。

无向图三/四元环计数

- 先给所有的边定向，若两端点度数不同，则由度数较小的点向度数较大的连边，否则由编号较小的向编号较大的连边，具体统计过程见代码。
- 考虑图中的一条边 $u \to v$，设 v 在新图中的出度为 out_v，总复杂度即 $\sum out_v$。
  - 若 v 在原图中的度数小于等于 $\sqrt{m}$，则显然有 $out_v \leq \sqrt{m}$。
  - 若 v 在原图中的度数大于 $\sqrt{m}$，在新图中它只能向度数大于 $\sqrt{m}$ 的点连边，原图中这样的点不会超过 $2\sqrt{m}$ 个，所以有 $out_v \leq 2\sqrt{m}$。
- 综上，该算法的时间复杂度为 $\mathcal{O}(m\sqrt{m})$，同时也意味着答案的规模也为 $\mathcal{O}(m\sqrt{m})$。

```
const int N = 1e5 + 5;
const int M = 2e5 + 5;
vector<int> e[N], o[N];
int px[M], py[M], deg[N], vis[N];

inline bool cmp(const int &x, const int &y)
{
    return deg[x] < deg[y] || deg[x] == deg[y] && x < y;
}

inline int countCycle3()
{
    int res = 0;
    for (int i = 1; i <= m; ++i)
        ++deg[px[i]], ++deg[py[i]];
    for (int i = 1; i <= m; ++i)
    {
        if (!cmp(px[i], py[i]))
            std::swap(px[i], py[i]);
        e[px[i]].emplace_back(py[i]);
    }
    for (int x = 1, y; x <= n; ++x)
    {
        for (int y : e[x])
            vis[y] = x;
        for (int y : e[x])
            for (int z : e[y])
                res += vis[z] == x;
```

```
    }
    return res;
}
```

- 四元环计数与三元环计数建新图的过程相同，为表示方便，设新图中 $u \to v$ 连边的条件为 $u < v$ ，则以下代码中枚举四元环各点间的关系为 $x < z, y_1 < z, y_2 < z$，而 (x, y_1) 和 (x, y_2) 是在原图中存在的边，显然这样不会重复计数，时间复杂度分析与三元环计数相同。

```
const int N = 1e5 + 5;
const int M = 2e5 + 5;
vector<int> e[N], o[N];
int px[M], py[M], deg[N], vis[N];

inline bool cmp(const int &x, const int &y)
{
    return deg[x] < deg[y] || deg[x] == deg[y] && x < y;
}

inline int countCycle4()
{
    ll res = 0;
    for (int i = 1; i <= m; ++i)
    {
        ++deg[px[i]], ++deg[py[i]];
        o[px[i]].emplace_back(py[i]);
        o[py[i]].emplace_back(px[i]);
    }
    for (int i = 1; i <= m; ++i)
    {
        if (!cmp(px[i], py[i]))
            std::swap(px[i], py[i]);
        e[px[i]].emplace_back(py[i]);
    }
    for (int x = 1, y; x <= n; ++x)
    {
        for (int y : o[x])
            for (int z : e[y])
                if (cmp(x, z))
                    res += vis[z]++;
        for (int y : o[x])
            for (int z : e[y])
                if (cmp(x, z))
                    vis[z] = 0;
    }
    return res;
}
```

LCA

DFS 序 + ST 表

- 预处理时间复杂度 $\mathcal{O}(n \log n)$，空间复杂度 $\mathcal{O}(n \log n)$，单次询问时间复杂度 $\mathcal{O}(1)$。
- 设结点 x 的 DFS 序编号为 $dfn[x]$，则 $x, y (dfn[x] < dfn[y])$ 的 LCA 为 $[dfn[x] + 1, dfn[y]]$ 上深度最小的结点的父亲。

```

```

```cpp
inline void dfs(int x)
{
    dfn[x] = ++tis;
    dep[x] = dep[fa[x]] + 1;
    f[0][dfn[x]] = fa[x];
    for (arc *e = adj[x]; e; e = e->nxt)
    {
        int y = e->to;
        if (y == fa[x])
            continue ;
        fa[y] = x;
        dfs(y);
    }
}

inline int queryLCA(int x, int y)
{
    if (x == y)
        return x;
    x = dfn[x], y = dfn[y];
    if (x > y) std::swap(x, y);
    ++x;
    int k = Log[y - x + 1];
    return depMin(f[k][x], f[k][y - (1 << k) + 1]);
}

inline void init()
{
    Log[0] = -1;
    for (int i = 1; i <= n; ++i)
        Log[i] = Log[i >> 1] + 1;
    dfs(rt);
    for (int j = 1; j <= Log[n]; ++j)
        for (int i = 1; i + (1 << j) - 1 <= n; ++i)
            f[j][i] = depMin(f[j - 1][i], f[j - 1][i + (1 << j - 1)]);
}
```

Tarjan

- 离线，时间复杂度和空间复杂度均为线性。
- 这里因为使用了 `vector` 和 `pair` 实测常数较大。

```cpp
inline int ufs_find(int x)
{
    if (fa[x] != x)
        return fa[x] = ufs_find(fa[x]);
    return x;
}

inline void Tarjan(int x)
{
    fa[x] = x;
    vis[x] = true;
    for (arc *e = adj[x]; e; e = e->nxt)
    {
        int y = e->to;
        if (vis[y])
```

```
            continue ;
        Tarjan(y);
        fa[y] = x;
    }
    for (pir e : query[x])
        if (vis[e.first])
            ans[e.second] = ufs_find(e.first);
}
```

树上路径并

- 给出两条路径 $(u_1,v_1),(u_2,v_2)$，它们路径并的两个端点为 $(u_1,u_2),(u_1,v_2),(v_1,u_2),(v_1,v_2)$ 四对点的 LCA 中深度最大的那两个点，是否存在路径并只要判断求出的其中一个点是否同时在两条路径上。

> **证明** 分类讨论即可。

重链剖分

- 令 $size[x]$ 为以 x 为根的子树的结点个数，令 y 为 x 所有子结点中 $size$ 值最大的子结点，则 (x,y) 为重边，y 称为 x 的重儿子，x 到其余子结点的边为轻边。
- 若 (x,y) 为轻边，则 $size[y]\le\lfloor\frac{size[x]}{2}\rfloor$，从根到某结点的路径上的轻边个数为 $\mathcal{O}(\log n)$，因此重路径数目也为 $\mathcal{O}(\log n)$。
- 对于任意两点 x,y，可将 x 到 y 的路径划分为 $\mathcal{O}(\log n)$ 个重路径，对应序列上的 $\mathcal{O}(\log n)$ 个区间，同时**不在这条路径上的所有点**也可以对应序列上的 $\mathcal{O}(\log n)$ 个区间。

```
inline void dfs1(int x)
{
    sze[x] = 1;
    for (arc *e = adj[x]; e; e = e->nxt)
    {
        int y = e->to;
        if (y == fa[x])
            continue ;
        fa[y] = x;
        dep[y] = dep[x] + 1;
        dfs1(y);
        sze[x] += sze[y];
        if (sze[y] > sze[son[x]])
            son[x] = y;
    }
}

inline void dfs2(int x)
{
    if (son[x])
    {
        pos[son[x]] = ++v;
        top[son[x]] = top[x];
        idx[v] = son[x];
        dfs2(son[x]);
    }
    int y;
    for (arc *e = adj[x]; e; e = e->nxt)
        if (!top[y = e->to])
        {
```

```
            pos[y] = ++V;
            idx[V] = y;
            top[y] = y;
            dfs2(y);
        }
}

inline void Init()
{
    dfs1(1);
    pos[1] = idx[1] = top[1] = V = 1;
    dfs2(1);
}

inline int pathQuery(int x, int y)
{
    int res = 0;
    while (top[x] != top[y])
    {
        if (dep[top[x]] < dep[top[y]])
            std::swap(x, y);
        res += querySum(1, 1, n, pos[top[x]], pos[x]);
        x = fa[top[x]];
    }
    if (dep[x] > dep[y])
        std::swap(x, y);
    return res + querySum(1, 1, n, pos[x], pos[y]);
}
```

长链剖分

- 令 $mx[x]$ 为以 x 为根的子树的最大深度，令 y 为 x 所有子结点中 mx 值最大的子结点，则 (x,y) 为重边，y 称为 x 的重儿子，x 到其余子结点的边为轻边。
- 沿着父结点跳，**经过轻边到达另一条长链时的长度至少增加 1，因此轻重边的切换次数至多为** $\mathcal{O}(\sqrt{n})$。

K 级祖先

- 类似可以得到，**从某一结点沿着父结点往上跳** k **步到达的长链长度一定大于等于** k。
- 因而可以记录从每条长链顶往上和往下跳长链长度个数到达的结点，对于询问某点的 k 级祖先，通过预处理找到 i 满足 $2^i \le k < 2^{i+1}$，先通过倍增数组跳 2^i 步，则根据 $k - 2^i < 2^i$，可以通过记录的数组确认跳剩余 $k - 2^i$ 步到达的结点。
- 预处理 $\mathcal{O}(n \log n)$，单次询问 $\mathcal{O}(1)$，实际意义不大，但可借鉴其思想。

```
inline void Create(int x)
{
    upl[x] = nowu;
    nowu += mx[x];
    downl[x] = nowd;
    nowd += mx[x];
}

inline void dfs(int x)
{
    dep[x] = dep[anc[x][0]] + 1;
    for (int i = 0; anc[x][i]; ++i)
```

```
        anc[x][i + 1] = anc[anc[x][i]][i];
    for (arc *e = adj[x]; e; e = e->nxt)
    {
        int y = e->to;
        dfs(y);
        if (mx[x] < mx[y] + 1)
        {
            mx[x] = mx[y] + 1;
            son[x] = y;
        }
    }
}

inline void dfs2(int x)
{
    if (son[x])
    {
        top[son[x]] = top[x];
        dfs2(son[x]);
    }
    for (arc *e = adj[x]; e; e = e->nxt)
    {
        int y = e->to;
        if (y == son[x])
            continue ;
        top[y] = y;
        dfs2(y);
    }
    if (!anc[x][0] || (anc[x][0] && son[anc[x][0]] != x))
    {
        Create(x);
        int u = x;
        for (int i = 0; i < mx[x]; ++i)
        {
            downl[x][i] = u;
            u = son[u];
        }
        u = anc[x][0];
        for (int i = 0; i < mx[x]; ++i)
        {
            upl[x][i] = u;
            u = anc[u][0];
        }
    }
}

inline void Init()
{
    dfs(rt);
    top[rt] = rt;
    dfs2(rt);
    maxb = 0;
    for (int i = 1; i <= n; ++i)
    {
        if (i >> maxb + 1 & 1)
            ++maxb;
        high[i] = maxb;
    }
```

```
}

inline int askAncestor(int x, int k)
{
    if (k == 0)
        return x;
    int t = high[k];
    x = anc[x][t];
    k -= 1 << t;
    if (k == 0)
        return x;
    if (mx[top[x]] - mx[x] >= k)
        return downl[top[x]][mx[top[x]] - mx[x] - k];
    else
        return upl[top[x]][k - (mx[top[x]] - mx[x]) - 1];
}
```

典例（优化DP）洛谷P5904

题目大意

- 给定一棵树，在树上选 3 个点，要求两两距离相等，求方案数。

解法

- 三个点两两距离相等，等价于选出三条长度相等且只有唯一公共点的路径。
- 设 $f_{x,i}$ 表示点 x 的子树内与点 x 距离为 i 的点数，转移为：

$$f_{x,i} = \sum_{y \in \text{child}_x} f_{y,i-1}$$

- 设 $g_{x,i}$ 表示在点 x 的子树内选取两点 y, z，两者的 LCA 为 w，满足 $\text{dist}(x,w) + i = \text{dist}(y,w) = \text{dist}(z,w)$ 的方案数。
- $g_{x,i}$ 的转移只要额外考虑 LCA 为点 x （$d = i$）的情况，其余情况可以由子节点的 g 得到，即（f', g' 表示处理 y 这棵子树前的值）：

$$g_{x,i} = \sum_{y \in \text{child}_x} \left(g_{y,i+1} + f'_{x,i} \times f_{y,i-1} \right)$$

- 最终答案为：

$$ans = \sum_{x=1}^{n} \left(g_{x,0} + \sum_{y \in \text{child}_x} \sum_{i} \left(f'_{x,i} \times g_{y,i+1} + g'_{x,i} \times f_{y,i-1} \right) \right)$$

- 考虑若 x 直接根据深度继承其重儿子的 f, g 值，则 f 所需的存储空间为总链长，g 所需的存储空间为总链长的两倍，同时总的转移复杂度也等同于总链长，因而时间复杂度和空间复杂度均为 $\mathcal{O}(n)$。

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

using std::ios;
using std::cin;
using std::cout;

typedef long long ll;
const int N = 1e5 + 5;
const int N2 = 2e5 + 5;
```

```cpp
const int M = 1e6 + 5;

int n, mx[N], son[N];
ll ans, tmpf[N], tmpg[N2], *f[N], *g[N];
ll *nowf = tmpf, *nowg = tmpg;

struct Edge
{
    int to; Edge *nxt;
}p[N2], *lst[N], *P = p;


inline void Link(int x, int y)
{
    (++P)->nxt = lst[x]; lst[x] = P; P->to = y;
    (++P)->nxt = lst[y]; lst[y] = P; P->to = x;
}

inline void Create(int x)
{
    f[x] = nowf;
    nowf += mx[x] + 1;
    g[x] = nowg + mx[x];
    nowg += mx[x] << 1 | 1;
}

inline void dfs1(int x, int Fa)
{
    for (Edge *e = lst[x]; e; e = e->nxt)
    {
        int y = e->to;
        if (y == Fa)
            continue;
        dfs1(y, x);

        if (mx[y] + 1 > mx[x])
        {
            mx[x] = mx[y] + 1;
            son[x] = y;
        }
    }
}

inline void dfs2(int x, int Fa)
{
    if (son[x])
    {
        f[son[x]] = f[x] + 1;
        g[son[x]] = g[x] - 1;
        dfs2(son[x], x);
    }
    f[x][0] = 1;
    ans += g[x][0];
    for (Edge *e = lst[x]; e; e = e->nxt)
    {
        int y = e->to;
        if (y == son[x] || y == Fa)
            continue;
```

```
        Create(y);
        dfs2(y, x);
        for (int i = 0; i <= mx[y]; ++i)
        {
            ans += g[x][i + 1] * f[y][i];
            if (i)
                ans += f[x][i - 1] * g[y][i];
        }
        for (int i = 0; i <= mx[y]; ++i)
        {
            g[x][i + 1] += f[y][i] * f[x][i + 1];
            f[x][i + 1] += f[y][i];
            if (i)
                g[x][i - 1] += g[y][i];
        }
    }
}

int main()
{
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    cout.tie(nullptr);

    cin >> n;
    for (int i = 1, x, y; i < n; ++i)
    {
        cin >> x >> y;
        Link(x, y);
    }
    dfs1(1, 0);
    Create(1);
    dfs2(1, 0);
    cout << ans << '\n';
}
```

典例（贪心）BZOJ3252

题目大意

- 给定一棵 n 个点的树，求 k 条从 1 号结点出发的链，使得这 k 条链的并包含的点权和最大。

解法

- 考虑以点权和为标准进行长链剖分，只需要选择点权和最大的前 k 条长链求和即为答案，不难证明其正确性：
  - 若某个轻儿子未被选择，则重儿子和长链点权和更大的轻儿子一定先于前被选择，保证了最优性。
  - 若某个轻儿子被选择，则其祖先所在的长链一定先于其被选择，点权和的计算不会遗漏。

虚树

- 对于 DFS序 连续的三个点 x, y, z，我们有 $\mathrm{LCA}(x, z) = \mathrm{LCA}(x, y)$ 或 $\mathrm{LCA}(y,z)$，因此只要将所有关键点按照 DFS序 排序，再将所有 $\mathrm{LCA}(x_i, x_{i+1})$ 与所有 x_i 作为虚树中的结点即可。
- 将虚树中的所有结点按照 DFS序 排序，按照 DFS序 的顺序模拟虚树的 DFS，用栈维护虚树中的根到当前结点的那一条路径，不难得到虚树结点之间的连边。

- 对于树上任意 k 个点的 LCA，类比上述结论，可知即为 k 个点中 DFS序 最小和最大的点的 LCA。
- 设按 DFS序 排完序后得到的结点为 $x_1, x_2, \ldots, x_k$，则虚树边权和的两倍为 $\operatorname{dist}(x_1, x_k) + \sum_{i=1}^{k-1} \operatorname{dist}(x_i, x_{i+1})$。

```cpp
inline bool cmp(const int &x, const int &y)
{
    return dfn[x] < dfn[y];
}

inline bool isSubtree(int x, int y)
{
    return dfn[y] >= dfn[x] && dfn[y] <= dfn[x] + sze[x] - 1;
}

inline void auxTree()
{
    top = 0;
    std::sort(vir + 1, vir + m + 1, cmp);
    for (int i = 1; i <= m; ++i)
        key[vir[i]] = true;
    for (int i = 1, im = m; i < im; ++i)
        vir[++m] = queryLCA(vir[i], vir[i + 1]);
    std::sort(vir + 1, vir + m + 1, cmp);
    m = std::unique(vir + 1, vir + m + 1) - vir - 1;
    for (int i = 1; i <= m; ++i)
    {
        while (top && !isSubtree(stk[top], vir[i]))
            --top;
        if (top)
            par[vir[i]] = stk[top];
        stk[++top] = vir[i];
    }
}
```

Kruskal 重构树

- 主要解决满足以下条件的问题：
  - 图的形态不变。
  - 每次询问从某点出发，只能通过边权小于或大于某个值的边，在线查询能到达的且满足某种性质的点或点数。
- 算法流程如下：
  - 初始时有 n 个孤立的点，其点权设为 $-\infty$。
  - 在 Kruskal 算法求最小生成树的过程中，遇到一条连接两个不同集合的边，我们在并查集中分别找到两个集合的根 x, y，新建一个结点 z，合并两个集合，并且令 z 为新集合的根。
  - 在重构树上令 z 为 x, y 的父结点，且 z 的点权为 (x, y) 的边权。
- 性质：
  - 为二叉树，满足大根堆的性质，原图中的点为其叶子结点。
  - 对于点对 (x, y)，它们在原图的所有路径中最大边权最小的路径的最大边权为两点在重构树中 LCA 的权值。
  - 对于一个叶子结点 x，找到它的一个深度最小的祖先 z，使得 z 的点权不超过 v，则 x 在原图中经过边权不超过 v 的边，所能到达的点集即为以 z 为根的子树内所有的叶子结点，可用传

统的数据结构维护。

网络流

- 设源点为 s，汇点为 t，每条边 e 的流量上限为 $c(e)$，流量为 $f(e)$。
- **割** 指对于某一顶点集合 $P \subset V$，从 P 出发指向 P 外部的那些原图中的边的集合，记作割 $(P, V/P)$。这些边的容量被称为割的容量。若 $s \in P, t \in V/P$，则称此时的割为 $s-t$ 割。
- 对于任意的 $s-t$ 流 F 和任意的 $s-t$ 割 $(P, V/P)$ 割，由每个点的流量平衡条件得：

$$F_{\text{的流量}} = P_{\text{出边总流量}} - P_{\text{入边总流量}} \leq \text{割的容量}$$

- 对于在残量网络中不断增广得到的流 F，设其对应的残量网络中从 s 出发可到达的顶点集为 S，则对于 S 指向 V/S 的边 e 有 $f(e) = c(e)$，而对 V/S 指向 S 的边有 $f(e) = 0$，则：

$$F_{\text{的流量}} = S_{\text{出边总流量}} - S_{\text{入边总流量}} = S_{\text{出边总流量}} = \text{割的容量}$$

- 因而 F 为最大流，同时 $(S, V/S)$ 为最小割，即**最大流等于最小割。**

Dinic 算法

- 主要思想即每次寻找最短的增广路，构造分层图，并沿着它多路增广。
- 每次多路增广完成后最短增广路长度至少增加 1，构造分层图次数为 $\mathcal{O}(n)$，在同一分层图中，每条增广路都会被至少一条边限制流量（我们称之为瓶颈），显然任意两条增广路的瓶颈均不相同，因而增广路总数为 $\mathcal{O}(m)$，加上当前弧优化，我们就能避免对无用边多次检查，寻找单条增广路的时间复杂度为 $\mathcal{O}(n)$，总时间复杂度 $\mathcal{O}(n^2 m)$。
- 这只是一个粗略的上界，实际的复杂度分析要结合具体的图模型，例如若最大流上限较小（设为 F），时间复杂度 $\mathcal{O}(F(n+m))$。

```
template <class T>
inline T Min(T x, T y) {return x < y ? x : y;}

const int N = 1e4 + 5;
const int M = 2e5 + 5;
const ll Maxn = 1e15;
int nxt[M], to[M], adj[N], que[N], cur[N], lev[N]; ll cap[M];
int n, m, src, des, qr, T = 1;

inline void linkArc(int x, int y, ll w)
{
    nxt[++T] = adj[x]; adj[x] = T; to[T] = y; cap[T] = w;
    nxt[++T] = adj[y]; adj[y] = T; to[T] = x; cap[T] = 0;
}

inline bool Bfs()
{
    for (int x = 1; x <= n; ++x)
        cur[x] = adj[x], lev[x] = -1;
    // 初始化具体的范围视建图而定，这里点的范围为 [1,n]
    que[qr = 1] = src;
    lev[src] = 0;
    for (int i = 1; i <= qr; ++i)
    {
        int x = que[i], y;
        for (int e = adj[x]; e; e = nxt[e])
            if (cap[e] > 0 && lev[y = to[e]] == -1)
            {
                lev[y] = lev[x] + 1;
```

```
                que[++qr] = y;
                if (y == des)
                    return true;
            }
    }
    return false;
}

inline ll Dinic(int x, ll flow)
{
    if (x == des)
        return flow;
    int y, delta; ll res = 0;
    for (int &e = cur[x]; e; e = nxt[e])
        if (cap[e] > 0 && lev[y = to[e]] > lev[x])
        {
            delta = Dinic(y, Min(flow - res, (ll)cap[e]));
            if (delta)
            {
                cap[e] -= delta;
                cap[e ^ 1] += delta;
                res += delta;
                if (res == flow)
                    break ;
                //此时 break 保证下次 cur[x] 仍有机会增广
            }
        }
    if (res != flow)
        lev[x] = -1;
    return res;
}

inline ll maxFlow()
{   //求完最大流后与 src 属于同一点集的点满足 lev[x] != -1
    ll res = 0;
    while (Bfs())
        res += Dinic(src, Maxn);
    return res;
}
```

- **单位网络** 在该网络中，所有边的流量均为 1，除源汇点以外的所有点，都满足入边或者出边最多只有一条。
- **结论** 对于包含二分图最大匹配在内的单位网络，Dinic 算法求解最大流的时间复杂度为 $\mathcal{O}(m\sqrt{n})$ 。

> **证明** 对于单位网络，每条边最多被考虑一次，一轮增广的时间复杂度为 $\mathcal{O}(m)$。
>
> 假设我们已经完成了前 $\sqrt{n}$ 轮增广，还需找到 d 条增广路才能找到最大流，每条增广路的长度至少为 $\sqrt{n}$。这些增广路不会在源点和汇点以外的点相交，因而至少经过了 $d\sqrt{n}$ 个点，$d \leq \sqrt{n}$，则至多还需增广 $\sqrt{n}$ 轮，总时间复杂度 $\mathcal{O}(m\sqrt{n})$。

经典模型

- **二者选其一的最小割** 有 n 个物品和两个集合 A, B，若第 i 个物品没有放入 A 集合花费 a_i，没有放入 B 集合花费 b_i，还有若干个限制条件，若 u_i 和 v_i 不在一个集合则花费 w_i。
    - 源点 s 向第 i 个点连一条容量为 a_i 的边，第 i 个点向汇点 t 连一条容量为 b_i 的边，在 u_i 和 v_i 之间连容量为 w_i 的双向边，最小割即最小花费。

- **最大权闭合子图** 给定一张有向图，每个点都有一个权值（可为负），选择一个权值和最大的子图，使得子图中每个点的后继都在子图中。
  - 若点权为正，则 s 向该点连一条容量为点权的边。
  - 若点权为负，则该点向 t 连一条容量为点权的相反数的边。
  - 原图上所有边的容量设为 $+\infty$。
  - 则答案为正点权之和减去最小割。
- **分糖果问题** n 个糖果 m 个小孩，小孩 i 对糖果 j 有偏爱度 $a_{i,j}=1/2$，设 $c_{i,j}=0/1$ 表示小孩 i 是否分得了糖果 j，小孩 i 觉得高兴当且仅当 $\sum\limits_{j=1}^{n} c_{i,j}a_{i,j}\geq b_i$，判断是否存在方案使所有小孩都高兴。
  - 偏爱度 $a_{i,j}=1/2$ 不好建图，转换思路先分配所有 $a_{i,j}=2$ 的糖果。
  - s 向所有糖果连一条容量为 1 的边，小孩 i 向 t 连一条容量为 $\lfloor\frac{b_i}{2}\rfloor$ 的边。
  - 对于所有满足 $a_{i,j}=2$ 的边，令糖果 j 向小孩 i 连一条容量为 1 的边。
  - 求得最大流 ans，则存在方案当且仅当 $ans+n\geq\sum\limits_{j=1}^{m} b_i$。
- **动态流问题** 宽为 w 的河上有 n 块石头，第 i 块坐标 (x_i,y_i)，同一时刻最大承受人数为 c_i，现有 m 个游客想要渡河，每人每次最远跳 d 米，单次耗时 1 秒，求全部渡河的最少时间（$n,m\leq 50$）。
  - 答案取值范围较小可暴力枚举，将每一时刻的石头都视作一点，在石头上跳跃可视作从第 t 时刻的石头 i 跳向第 $t+1$ 时刻的石头 j，每次将时刻加一，建出新的点和边后跑最大流，直至总流量大于等于 m。

费用流

- 在最大流的前提下使该网络总花费最小。

SSP 算法

- 每次寻找单位费用最小的增广路进行增广，直至图中不存在增广路为止。
- 设流量为 i 的时候最小费用为 f_i，假设初始网络上没有负圈，$f_0=0$。
- 假设用 SSP 算法求出的 f_i 是最小费用，我们在 f_i 的基础上，找到一条最短的增广路，从而求出 f_{i+1}，此时 $f_{i+1}-f_i$ 就是这条最短增广路的长度。
- 假设存在更小的 f_{i+1}，设其为 f'_{i+1}，则 $f'_{i+1}-f_i$ 一定对应一个经过至少一个负圈的增广路。若残量网络中存在至少一个负圈，则可在不增加 s 流出的流量的情况下使费用减小，与 f_i 是最小费用矛盾。
- 综上，SSP 算法可以正确求出无负圈网络的最小费用最大流，设最大流为 F，总时间复杂度 $\mathcal{O}(Fnm)$。

```cpp
template <class T>
inline T Min(T x, T y) {return x < y ? x : y;}

const int N = 5e3 + 5;
const int M = 1e5 + 5;
const int Maxn = 1e9;
int nxt[M], to[M], cap[M], que[M], cst[M], adj[N], dis[N];
bool vis[N]; int n, m, src, des, ans, T = 1, qr;

inline void linkArc(int x, int y, int w, int z)
{
    nxt[++T] = adj[x]; adj[x] = T; to[T] = y; cap[T] = w; cst[T] = z;
    nxt[++T] = adj[y]; adj[y] = T; to[T] = x; cap[T] = 0; cst[T] = -z;
}

```

```cpp
inline bool SPFA()
{
    for (int x = 1; x <= n; ++x)
        dis[x] = Maxn, vis[x] = false;
    // 初始化具体的范围视建图而定，这里点的范围为 [1,n]
    dis[que[qr = 1] = src] = 0;
    for (int i = 1, x, y; i <= qr; ++i)
    {
        vis[x = que[i]] = false;
        for (int e = adj[x]; e; e = nxt[e])
            if (cap[e] > 0 && dis[y = to[e]] > dis[x] + cst[e])
            {
                dis[y] = dis[x] + cst[e];
                if (!vis[y])
                    vis[que[++qr] = y] = true;
            }
    }
    return dis[des] < Maxn;
}

inline int Dinic(int x, int flow)
{
    if (x == des)
    {
        ans += flow * dis[des];
        return flow;
    }
    vis[x] = true;
    int y, delta, res = 0;
    for (int e = adj[x]; e; e = nxt[e])
        if (!vis[y = to[e]] && cap[e] > 0 && dis[y] == dis[x] + cst[e])
        // vis 数组防止 dfs 在总费用为 0 的环上死循环
        {
            delta = Dinic(y, Min(flow - res, cap[e]));
            if (delta)
            {
                cap[e] -= delta;
                cap[e ^ 1] += delta;
                res += delta;
                if (res == flow)
                    break ;
            }
        }
    return res;
}

inline int MCMF()
{
    ans = 0;
    int res = 0;
    while (SPFA())
        res += Dinic(src, Maxn);
    return res;
}
```

经典模型

- **餐巾计划问题** 一家餐厅在接下来的 T 天内需用餐巾，第 i 天需要 r_i 块干净餐巾，餐厅可任意购买单价为 p 的干净餐巾，或者将脏餐巾送往快洗部（单块餐巾需洗 m 天，花费 f）或慢洗部（单块餐巾需洗 n 天，花费 s），求这 T 天内的最小花费。
  - 主要的难点在于需将干净餐巾和脏餐巾区分开，第 i 天分设点 i, i'，流向这两点的流量分别表示第 i 天的干净餐巾和脏餐巾。
  - s 向 i 连一条容量为 $+\infty$ 、费用为 p 的边，i 向 t 连一条容量为 r_i、费用为 0 的边。
  - s 向 i' 连一条容量为 r_i、费用为 0 的边，i' 向 $i+m$ 连一条容量为 $+\infty$，费用为 f 的边，向 $i+n$ 连一条容量为 $+\infty$，费用为 s 的边，向 $(i+1)'$ 连一条容量为 $+\infty$，费用为 0 的边。
  - 显然该建图能使到 t 的边满流，则最小费用即为所求。
- **费用流转二分图最大匹配** 若除去费用的建图为二分图且费用不为 0 的边均在二分图一侧，可将有费用的边从小到大排序，按照这一顺序跑匈牙利算法，容易证明最大匹配即对应最小费用，可将时间复杂度降至 $\mathcal{O}(nm)$。

上下界网络流

- 以下 (x, y, l, r) 代指 x 向 y 容量上限为 l 下限为 r 的边，(x, y, w) 代指 x 向 y，连容量为 w 的边。

无源汇上下界可行流

- 对原图中每条边 (x, y, l, r)，假设每条边已有初始流量 l，在新图中连边 $(x, y, r-l)$。
- 对于图中每个点 x，记点 x 的初始流入流量减去初始流出流量为 Δ_x，新建超级源汇点 S' 和 T'：
  - 若 $\Delta_x > 0$，连边 (S', x, Δ_x)。
  - 若 $\Delta_x < 0$，连边 $(x, T', -\Delta_x)$。
- 对新图跑最大流算法，若 S' 的出边全部满流则存在可行流，可行流中每条边的流量等于初始流量加上当前流量。

```
    cin >> n >> m;
    src = n + 1, des = n + 2;
    for (int i = 1, x, y, l, r; i <= m; ++i)
    {
        cin >> x >> y >> l >> r;
        linkArc(x, y, r - l);
        ide[i] = T; lim[i] = l;
        delt[x] -= l, delt[y] += l;
    }
    ll tot = 0;
    for (int i = 1; i <= n; ++i)
        if (delt[i] > 0)
            linkArc(src, i, delt[i]), tot += delt[i];
        else if (delt[i] < 0)
            linkArc(i, des, -delt[i]);
    if (maxFlow() == tot) //注意结点初始化的范围为n+2
    {
        cout << "YES\n";
        for (int i = 1; i <= m; ++i)
            cout << cap[ide[i]] + lim[i] << '\n';
    }
    else cout << "NO\n";
```

有源汇上下界可行流

- 在无源汇上下界可行流的基础上，在原有的源汇点 S 和 T 间增加连边 (T, S, ∞)，跑最大流后该边反向弧的流量即为可行流的流量。

```
    cin >> n >> m >> src >> des;
    for (int i = 1, x, y, l, r; i <= m; ++i)
    {
        cin >> x >> y >> l >> r;
        linkArc(x, y, r - l);
        delt[x] -= l, delt[y] += l;
    }
    linkArc(des, src, Maxn);
    int key = T;
    src = n + 1, des = n + 2;
    ll tot = 0;
    for (int i = 1; i <= n; ++i)
        if (delt[i] > 0)
            linkArc(src, i, delt[i]), tot += delt[i];
        else if (delt[i] < 0)
            linkArc(i, des, -delt[i]);
    if (maxFlow() == tot) //注意结点初始化的范围为 n+2
        cout << cap[key] << '\n';
    else
        cout << "please go home to sleep\n";
```

有源汇上下界最大流

- 在有源汇上下界可行流的基础上，直接跑原图 S 到 T 的最大流，此时 (T, S, ∞) 的反向弧流量会被退回，因此所求即为有源汇上下界最大流。

```
    cin >> n >> m >> src >> des;
    for (int i = 1, x, y, l, r; i <= m; ++i)
    {
        cin >> x >> y >> l >> r;
        linkArc(x, y, r - l);
        delt[x] -= l, delt[y] += l;
    }
    linkArc(des, src, Maxn);
    int _src = src, _des = des;
    src = n + 1, des = n + 2;
    ll tot = 0;
    for (int i = 1; i <= n; ++i)
        if (delt[i] > 0)
            linkArc(src, i, delt[i]), tot += delt[i];
        else if (delt[i] < 0)
            linkArc(i, des, -delt[i]);
    if (maxFlow() == tot) //注意结点初始化的范围为 n+2
    {
        src = _src, des = _des;
        cout << maxFlow() << '\n';
    }
    else
        cout << "please go home to sleep\n";
```

有源汇上下界最小流

- 在有源汇上下界可行流的基础上，删去 (T, S, ∞)，之后跑 T 到 S 的最大流，此时可行流减去最大流的流量即为有源汇上下界最小流。

```
    cin >> n >> m >> src >> des;
    for (int i = 1, x, y, l, r; i <= m; ++i)
    {
        cin >> x >> y >> l >> r;
        linkArc(x, y, r - l);
        delt[x] -= l, delt[y] += l;
    }
    int _src = src, _des = des;
    src = n + 1, des = n + 2;
    ll tot = 0;
    for (int i = 1; i <= n; ++i)
        if (delt[i] > 0)
            linkArc(src, i, delt[i]), tot += delt[i];
        else if (delt[i] < 0)
            linkArc(i, des, -delt[i]);
    linkArc(_des, _src, Maxn);
    if (maxFlow() == tot)
    {
        ll allow = cap[T];
        src = _des, des = _src;
        cap[T] = cap[T - 1] = 0;
        cout << allow - maxFlow() << '\n';
    }
    else
        cout << "please go home to sleep\n";
```

最小费用可行流

- 在可行流的边中增加费用即可，注意要加上初始流量的费用。

最小割树

- 即 Gomory-Hu Tree，用于求无向图中任意两点间的最小割。
- 设当前的点集为 V，从 V 取出两点 s 和 t，求 **原图** 中 s 到 t 的最小割 w，在新图中连边 (s, t, w)，最小割将 V 分为两个点集 V_1, V_2，递归下去直至每个点集仅包含一点，新图构成了一棵树，若采用的是 Dinic 算法，时间复杂度 $\mathcal{O}(n^3m)$。
- **结论** 原图中任意两点间的最小割等于树上两点间的路径最小值。

> **证明** 先考虑证明三个点的情况，即存在两条边 (a, b, w_1) 和 (b, c, w_2)，需证明 a 和 c 间的最小割 $w = \min\{w_1, w_2\}$。
>
> w 将原图分为两个点集，仅可能存在两种情况之一，故得证。
>
> - a, b 属于同一点集，最小割为 w_2。
> - b, c 属于同一点集，最小割为 w_1。
>
> 对于一条路径，不断选择一端的三点，用 $(a, c, \min\{w_1, w_2\})$ 替换 (a, b, w_1) 和 (b, c, w_2) 即可。

```
inline void solve(int l, int r)
{   // curp[i] 初始为 i
    if (l == r)
```

```
        return ;
    src = curp[l], des = curp[l + 1];
    for (int i = 2; i <= T; i += 2)
    {
        int w = cap[i] + cap[i ^ 1] >> 1;
        cap[i] = cap[i ^ 1] = w;
    }
    int w = maxFlow();
    e[src].emplace_back(des, w);
    e[des].emplace_back(src, w);

    vector<int> _cur;
    for (int i = l; i <= r; ++i)
    {
        int x = curp[i];
        if (lev[x] != -1)
            _cur.emplace_back(x);
    }
    int mid = l + _cur.size() - 1;
    for (int i = l; i <= r; ++i)
    {
        int x = curp[i];
        if (lev[x] == -1)
            _cur.emplace_back(x);
    }
    for (int i = l; i <= r; ++i)
        curp[i] = _cur[i - l];
    solve(l, mid);
    solve(mid + 1, r);
}
```

无向图最小割

- Stoer-Wagner 算法可在 $\mathcal{O}(n^3)$ 时间内求解无向图最小割。
- 具体流程为，每次 $\mathcal{O}(n^2)$ 找到一对点 s,t 的最小割（见函数 `calcMinCut` 和以下描述），之后将 s,t 及其之间的边缩为一点，直至图中仅剩一点时，过程中求得的最小割即为答案。
- 定义 $d(x,y)$ 为邻接矩阵，权值函数 $w(A,x)=\sum_{y\in A}d(x,y)$，每次将 $w(A,x)$ 最大且不属于 A 的 x 加入 A，直至 $A=V$。
- **结论** 设 A_x 表示所有在 x 之前加入的点的点集，s,t 为最后加入 A 的两个结点，则 $w(A_t,t)$ 为 s,t 的最小割。

> **证明** 即证明对于任意 $s-t$ 割 C，总有 $w(A_t,t)\le w(C)$，其中 $w(C)=\sum_{(x,y)\in C}d(x,y)$。
>
> 定义一个点 x 被 **激活** 当且仅当 A_x 中最后一个点与 x 分属 C 的两侧，且 C_x 为 C 在 $A_x\cup\{x\}$ 下的诱导割，现在归纳证明，对于一个被激活的点 x，恒有：
>
> $$w(A_x,x)\le w(C_x)$$
>
> - 对于第一个被激活的点，显然有 $w(A_x,x)=w(C_x)$。
> - 否则，设当前被激活的点为 x，上一个被激活的点为 y，则
>
> $$\begin{aligned}w(A_x,x)&=w(A_y,x)+w(A_x-A_y,x)\\&\le w(A_y,y)+w(A_x-A_y,x)\\&=w(C_y)+w(A_x-A_y,x)\\&\le w(C_x)\end{aligned}$$
>
> 由 $w(A_t,t)$ 的定义，t 一定被激活，代入 t 即得到 $w(A_t,t)\le w(C_t)=w(C)$，证毕。

- 该结论也意味着对于一个无向图，总存在一条边 (s,t) 使得 t 的所有邻边是 s,t 的最小割，并且可在类似 Dijkstra 的时间复杂度内找到。

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

template <class T>
inline void CkMin(T &x, T y) {x > y ? x = y : 0;}

using std::ios;
using std::cin;
using std::cout;

const int N = 60;
const int Maxn = 1e9;
int n, m, src, des;
int dis[N][N], w[N], ord[N];
bool del[N], vis[N];

inline int calcMinCut(int x)
{
    for (int i = 1; i <= n; ++i)
        w[i] = 0, vis[i] = false;
    for (int i = 1; i <= n - x + 1; ++i)
    {
        int id = 0;
        for (int j = 1; j <= n; ++j)
            if (!del[j] && !vis[j] && (!id || w[j] > w[id]))
                id = j;
        vis[id] = true, ord[i] = id;
        for (int j = 1; j <= n; ++j)
            if (!del[j] && !vis[j])
                w[j] += dis[id][j];
    }
    src = ord[n - x], des = ord[n - x + 1];
    return w[des];
}

inline void Contraction()
{
    del[des] = true;
    for (int j = 1; j <= n; ++j)
    {
        dis[src][j] += dis[des][j];
        dis[j][src] += dis[j][des];
    }
}

inline int StoerWagner()
{
    int res = Maxn;
    for (int i = 1; i < n; ++i)
    {
        CkMin(res, calcMinCut(i));
        Contraction();
    }
    return res;
}
```

```cpp

int main()
{
    srand(time(0));
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    cout.tie(nullptr);

    cin >> n >> m;
    int x, y, z;
    while (m--)
    {
        cin >> x >> y >> z;
        dis[x][y] += z;
        dis[y][x] += z;
    }
    cout << StoerWagner() << '\n';
    return 0;
}
```

二分图最大匹配

- 记图 $G=(V,E)$。
  - **匹配** G 中两两没有公共端点的边集合 $M\subseteq E$。
  - **边覆盖** G 中的任意顶点都至少是 F 中某条边的边集合 $F\subseteq E$。
  - **独立集** G 中两两互不相连的顶点集 $S\subseteq V$。
  - **点覆盖** G 中任意边都有至少一个端点属于 P 的顶点集合 $P\subseteq V$。
- **结论1** 对于**不存在孤立点**的图，$|\text{最大匹配}|+|\text{最小边覆盖}|=|V|$。

> **证明** 设最大匹配数为 x，则其覆盖的点数为 $2x$，剩余点两两之间没有边。则最少需要增加 $|V|-2x$ 条边才能将所有点覆盖，则最小边覆盖数为 $x+|V|-2x=|V|-x$。

- **结论2** $|\text{最大独立集}|+|\text{最小点覆盖}|=|V|$。

> **证明** 只需证明独立集和点覆盖一一对应且互为关于 V 的补集即可。取一个点覆盖关于 V 的补集，若其不是一个独立集，则存在两点有公共边，不难发现这与点覆盖的定义矛盾。

- **结论3** 在**二分图**中，$|\text{最大匹配}|=|\text{最小点覆盖}|$。

> **证明** 设最大匹配数为 x，即让匹配中的每条边都和其一个端点关联。x 个点是足够的，否则若存在一条边未被覆盖，加入这条边能得到一个更大的匹配。x 个点是必需的，因为匹配的 x 条边两两无公共点。

- **结论4** 在**有向无环图**中，$|\text{最小点不相交路径覆盖}|+|\text{最大匹配}|=|V|$。这里的最大匹配指的是，将原图每个点拆成入点和出点两点，对于有向边 $x\to y$，将 x 的入点向 y 的出点连边，在该二分图上求得的最大匹配。
  - 若需求 $|\text{最小点可相交路径覆盖}|$，可以每个点为起点 **BFS** 求出传递闭包，传递闭包上建图即可，传递闭包中图的边数可能达到 $\mathcal{O}(|V|^2)$，时间复杂度 $\mathcal{O}(|V||E|+|V|^{2.5})$，空间复杂度 $\mathcal{O}(|V|^2)$，输出方案记录每个入点的匹配点即可，时间复杂度 $(|V|)$。
  - 若 $|E|$ 较小，也可以不求传递闭包，将有向边的容量设为 $+\infty$，并将所有的出点向对应的入点连一条容量为 $+\infty$ 的边，即可实现传递闭包的效果，直接跑网络流即可，因为最大流上限为 $|V|$，时间复杂度 $\mathcal{O}(|V|(|V|+|E|))$，空间复杂度 $\mathcal{O}(|V|+|E|)$，实际上严格优于前述做法。该做法的输出方案较为麻烦，具体来说，我们根据所有匹配边建出一张图，若一个点的出边比入边多则它可以作为一条链的起点，暴力往下跳即可，实现时将本质相同的边压缩成一条可将空间复杂度降至 $\mathcal{O}(|V|+|E|)$，时间复杂度 $\mathcal{O}(|V|^2)$，参考代码如下：

```
    // C 为二分图一侧的点数，src = 2 * C + 1
    // 注意孤立点以及删去一条链可能会产生新的起点的情况
    for (int x = 1; x <= C; ++x)
        for (int e = adj[x]; e; e = nxt[e])
            if (to[e] > C && to[e] < src && to[e] - C != x && cap[e ^ 1] > 0)
            {
                re[x].emplace_back(std::make_pair(to[e] - C, cap[e ^ 1]));
                ++lre[x];
                sre[x] += cap[e ^ 1];
                ind[to[e] - C] += cap[e ^ 1];
            }
    tis = 0;
    for (int t = 1; t <= C; ++t)
    {
        for (int x = 1; x <= C; ++x)
            while (sre[x] > ind[x])
            {
                ++tis;
                int u = x;
                ans[u] = tis;
                while (sre[u])
                {
                    pir &v = re[u][lre[u] - 1];
                    int vid = v.first;
                    if (--v.second == 0)
                        re[u].pop_back(), --lre[u];
                    --sre[u];
                    u = vid;
                    ans[u] = tis;
                    --ind[u];
                }
            }
    }
```

> **证明** 只需证明二分图中匹配与原图中的路径覆盖一一对应且总和为 $|V|$。对于每个没有匹配边的出点，我们都能构造一条路径，若其对应的入点存在匹配边，则将该匹配边加入该路径同时继续考虑该匹配边的出点直至其对应的入点不存在匹配边。得到的所有路径恰能覆盖所有点，且没有匹配边的出点和入点恰好能两两配对分别构成所有路径的起点和终点。

- **Dilworth定理** 对于任意有限偏序集，其最大反链中的元素个数必等于最小链划分中链的个数。
  - 有限偏序集的最小链划分即其哈斯图对应的有向无环图的 $|\text{最小点可相交路径覆盖}|$，用上述方法求解即可。

> **证明** 设 m 为最大反链的元素个数，只要证明偏序集 X 可被划分成 m 个链即可。
>
> 考虑用归纳法证明，当 $n = 1$ 时显然成立，下面证明 $n > 1$ 时的情形，设该有限偏序集中所有极小点构成的集合为 L，所有极大点构成的集合为 G，分两种情况讨论：
>
> 1. 若存在最大反链 A，使得 $A \neq L$ 且 $A \neq G$。构造：
>
> $$A^+ = \{x | x \in X \wedge \exists a \in A, a \leq x\}$$
> $$A^- = \{x | x \in X \wedge \exists a \in A, x \leq a\}$$
>
> 则容易得到 $|A^+| < |X|$、$|A^-| < |X|$ 且 $A^+ \cup A^- = X, A^+ \cap A^- = A$，由归纳假设可将 A^+ 和 A^- 的 m 链划分拼接起来即可得到 X 的 m 链划分。

2. 若只存在反链 $A=L$ 或 $A=G$，取极小元 x 和极大元 y 满足 $x\leq y$（x,y 可相等），则由归纳假设 $X-\{x,y\}$ 最大反链的元素个数为 $m-1$，存在 $m-1$ 链划分，增加链 $x\leq y$ 可得到 X 的 m 链划分。

- **Dilworth定理的对偶形式** 最长链中元素个数必等于最小反链划分中反链的个数。

> **证明** 设最长链的长度为 m，最小反链划分的数目为 M。
>
> 1. 由于在任意一个反链中选取超过一个元素都违反链的定义，容易得到 $m\leq M$。
> 2. 考虑每次删去当前偏序集中的所有极小点，恰好删除 m 次能够将所有点删除，设第 i 次删点构成的极小点集合为 A_i，则 $A_1,A_2,\ldots,A_m$ 构成一个反链划分，故有 $M\leq m$。
>
> 综上所述，$m=M$，原命题得证。

匈牙利算法

- 从每个点出发尝试找到一条增广路，时间复杂度 $\mathcal{O}(nm)$。

```
inline bool Hungary(int x)
{
    int y;
    for (arc *e = adj[x]; e; e = e->nxt)
        if (!mateR[y = e->to])
            return mateR[y] = x, true;
    for (arc *e = adj[x]; e; e = e->nxt)
    {
        if (vis[y = e->to] == tis)
            continue ;
        vis[y] = tis;
        if (Hungary(mateR[y]))
            return mateR[y] = x, true;
    }
    return false;
}

inline int maxMatch()
{
    int cnt = 0;
    for (int i = 1; i <= n; ++i)
    {
        ++tis;
        if (Hungary(i))
            ++cnt;
    }
    return cnt;
}
```

二分图最大权匹配

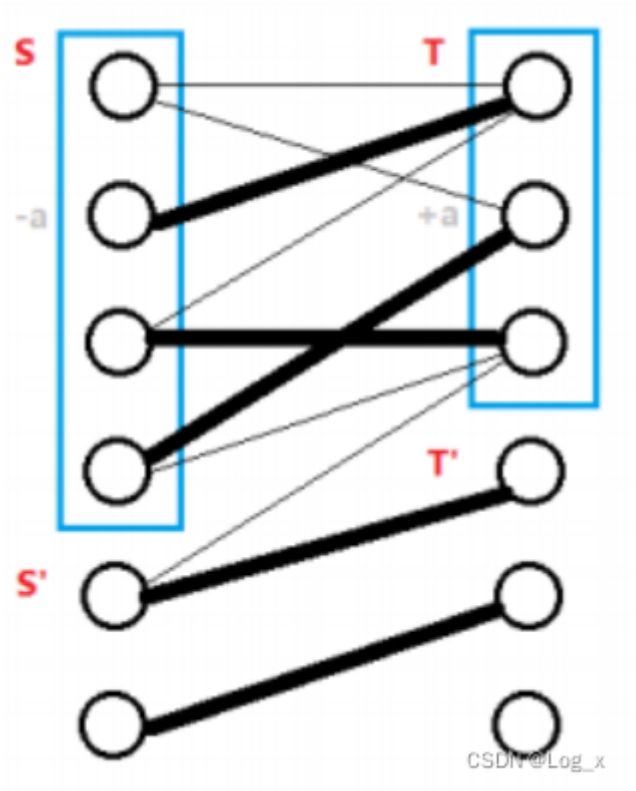
Kuhn-Munkres算法

- 该算法用于 $\mathcal{O}(n^3)$ 求二分图的最大权完美匹配，若二分图左右部的点数不相同，需将点数补至相同，并将所有不存在的边的权设为 0。
- 若题目对是否是完备匹配有要求，需将不存在的边设为 $-\infty$（具体值需保证一旦选这种边就比所有不选这种边的方案更劣），此时也能处理原图边权有负数的情况。
- **可行顶标** 对于二分图 $<X,E,Y>$，对于 $x\in X$ 分配权值 $\mathrm{labx}(x)$，对于 $y\in Y$ 分配权值 $\mathrm{laby}(y)$，对于所有边 $(x,y)\in E$ 满足 $w(x,y)\leq \mathrm{labx}(x)+\mathrm{laby}(y)$。

- **相等子图** 在一组可行顶标下原图的生成子图，包含所有点但只满足包含 $\text{labx}(x)+\text{laby}(y)=w(x,y)$ 的边 (x,y)。
- **定理** 对于某组可行顶标，若其相等子图内存在完美匹配，该匹配为原图的最大权完美匹配。

> **证明** 设可行顶标下相等子图中的完美匹配为 M'，对于原二分图任意一组完美匹配 M，其边权和
>
> $$\text{val}(M)=\sum_{(x,y)\in M}w(x,y)\le\sum_{x\in X}\text{labx}(x)+\sum_{y\in Y}\text{laby}(y)=\text{val}(M')$$
>
> 故若 M' 存在，M' 即为最大权完美匹配。

- 下面的算法过程将证明一定能够通过调整顶标，使得相等子图内存在完美匹配。
- 初始可行顶标 $\text{labx}(x)=\max\limits_{y\in Y}\{w(x,y)\},\text{laby}(y)=0$，在相等子图内找一个未匹配的点，尝试寻找增广路，若能找到直接增广，否则将形成一棵交错树。
- 交错树是由从一个未匹配点出发沿着 非匹配边-匹配边 交替的所有路径构成的树，将树中顶点与 X 的交集记为 S，与 Y 的交集记为 T，令 $S'=X-S,T'=Y-T$（$S/S',T/T'$ 程序中分别用 $\text{visx}(x)$ 和 $\text{visy}(y)$ 标示），如下图所示。



- 在相等子图中：
  - $S-T'$ 的边不存在，若其为匹配边违反交错树的定义，若其为非匹配边则会形成增广路或者使交错树扩大。
  - $S'-T$ 的边一定是非匹配边，否则将使交错树扩大。
- 将 S 中的顶标 $-a$，将 T 中的顶标 $+a$，可以发现：
  - $S-T$ 和 $S'-T'$ 在相等子图中的边均无变化。
  - $S-T'$ 的边对应顶标和减少，可能作为非匹配边加入相等子图，由上述性质，增加的边将形成增广路或使交错树扩大。
  - $S'-T$ 的边对应顶标和增加，不可能加入相等子图。
- 设 $\text{slacky}(y)=\min\limits_{x\in S}\{\text{labx}(x)+\text{laby}(y)-w(x,y)\}$（当且仅当 $y\in T'$ 该值才有意义），则令 $a=\min\limits_{y\in T'}\{\text{slacky}(y)\}$，每次修改顶标时，必定会有至少一条产生增广路或使交错树扩大的非匹配边加入相等子图。

    - 在遍历交错树的过程中，记 $\mathrm{pre}(y)=x(x\in X,y\in Y)$ 表示交错树上 x,y 通过非匹配边相连，若产生了增广路，根据 pre 更新匹配边即可完成增广。
    - 若使交错树扩大， 更新 slacky，在交错树增加的结点中重复上述过程直至完成增广，由于左右部点数相同，这一过程一定能够结束。
- 遍历交错树的部分通过 BFS 实现。总共需要完成 n 次增广，每次增广中 BFS 的总时间复杂度为 $\mathcal{O}(n^2)$，更新顶标的次数为 $\mathcal{O}(n)$（每次更新都会使 T 扩大），每次更新后维护 slacky 并确定 a 的时间复杂度也为 $\mathcal{O}(n)$，故总的时间复杂度为 $\mathcal{O}(n^3)$。

```
typedef long long ll;
const int N = 405;
const int Maxn = 2e9;
int que[N], w[N][N], slacky[N];
int labx[N], laby[N], matex[N], matey[N], pre[N];
bool visx[N], visy[N];
int nl, nr, qr, n, m; ll ans;

inline bool Augment(int y)
{
    if (matey[y])
    {
        que[++qr] = matey[y];
        visx[matey[y]] = visy[y] = true;
        return false;
    }
    else
    {
        while (y)
        {
            int x = pre[y];
            matey[y] = x;
            std::swap(matex[x], y);
        }
        return true;
    }
}

inline void bfsHungary(int src)
{
    for (int i = 1; i <= n; ++i)
    {
        pre[i] = 0;
        visx[i] = visy[i] = false;
        slacky[i] = Maxn;
    }
    visx[que[qr = 1] = src] = true;
    while (1)
    {
        for (int i = 1, x; i <= qr; ++i)
        {
            x = que[i];
            for (int y = 1; y <= n; ++y)
                if (!visy[y])
                {
                    int delta = labx[x] + laby[y] - w[x][y];
                    if (delta > slacky[y])
                        continue ;
                    pre[y] = x;
```

```
                    if (delta > 0)
                        slacky[y] = delta;
                    else if (Augment(y))
                        return ;
                }
        }
        int nxt, delta = Maxn;
        for (int y = 1; y <= n; ++y)
            if (!visy[y] && slacky[y] < delta)
                delta = slacky[y], nxt = y;
        for (int i = 1; i <= n; ++i)
        {
            if (visx[i])
                labx[i] -= delta;
            if (visy[i])
                laby[i] += delta;
            else
                slacky[i] -= delta;
        }
        qr = 0;
        if (Augment(nxt))
            return ;
    }
}

int main()
{
    read(nl); read(nr); read(m);
    n = Max(nl, nr);
    int x, y, z;
    while (m--)
    {
        read(x); read(y); read(z);
        CkMax(w[x][y], z);
        CkMax(labx[x], z);
    }
    for (int i = 1; i <= n; ++i)
        bfsHungary(i);

    for (int i = 1; i <= n; ++i)
        ans += w[i][matex[i]];
    put(ans), putchar('\n');
    for (int i = 1; i <= nl; ++i)
        put(w[i][matex[i]] ? matex[i] : 0), putchar(' ');
}
```

二分图稳定匹配

问题描述

- 对于任意的 $x \in X$，存在一个由 Y 内所有结点构成的偏爱列表 $prf_X[x]$，列表中结点排名越靠前对 x 适配度越高，记 $rk_X[x][y]$ 表示 y 在 $prf_X[x]$ 中的排名。
- 对于任意的 $y \in Y$，同样定义 $prf_Y[y]$ 和 $rk_Y[y][x]$。
- 对于二分图 $< X, Y, E >$，某一匹配是**稳定**的（Stable Matching)当且仅当（记 x 的匹配结点为 $mat_X[x]$，y 的匹配结点为 $mat_Y[y]$）：

- 不存在 x, y 满足 $x \neq mat_Y[y]$，且：
  - $rk_X[y] < rk_X[mat_X[x]]$
  - $rk_Y[x] < rk_Y[mat_Y[y]]$

Gale-Shapley 算法

- 描述该算法的 python 代码如下：

```python
rkx = rky = matx = maty = dict()
for x in prfx.keys():
    for i in range(len(prfx[x])):
        rkx[(x, prfx[x][i])] = i
for y in prfy.keys():
    for i in range(len(prfy[y])):
        rky[(y, prfy[y][i])] = i

for x in prfx.keys():
    prfx[x].reverse()
list = [x for x in prfx.keys()]
while len(list) > 0:
    x = list.pop()
    while len(prfx[x]) > 0:
        y = prfx[x].pop()
        if y not in maty:
            maty[y] = x
            matx[x] = y
            break
        elif rky[(y, x)] < rky[(y, maty[y])]:
            del matx[maty[y]]
            list.append(maty[y])
            maty[y] = x
            matx[x] = y
            break
```

- 关于该算法的正确性：
  - 如果 $|X| = |Y|$，最后总能找到一个匹配，且 $prf_X[x]$ 中的元素不会重复遍历，时间复杂度 $\mathcal{O}(|X||Y|)$。
  - 考虑任何不在现有匹配中的 (x, y)：
    - 若 x 从未尝试与 y 匹配，则说明 $rk_X[x][mat_X[x]] < rk_X[x][y]$。
    - 若 x 尝试过与 y 匹配，则说明 $rk_Y[y][mat_Y[y]] < rk_Y[y][x]$。
  - 故该匹配是稳定的。
- **定义1** 配对 (x, y) 是**有效**的当且仅当存在一个稳定匹配使之包含 (x, y)。
- **定义2** 记 y_x 表示在列表 $prf_X[x]$ 中最靠前的 y，一个匹配为 X-**最优匹配**当且仅当若 (x, y_x) 是有效的，则该匹配一定包含 (x, y_x)。
- **结论** Gale-Shapley 算法产生的匹配 M 一定为 X-最优匹配。

> **证明** 记 $y = y_x$，假设 (x, y) 是有效的且 M 不包含 (x, y)。
>
> 设 M 中与 y 匹配的是 x'，由算法流程 $rk_Y[y][x'] < rk_Y[y][x]$（要么 x 匹配后被 x' 替代，要么 x' 先匹配后无法被 x 替代）。
>
> 由有效配对的定义，一定存在稳定匹配 M' 包含 (x, y)。

> 记 M' 中与 x' 匹配的是 y'，能够得到匹配 M 的条件是 $rk_X[x'][y] < rk_X[x'][y']$，则由于 (x', y) 的存在，M' 是不稳定的，与假设矛盾。

图的着色

点着色

- 设图 G 的点色数为 $\chi(G)$，则显然有 $\chi(G) \leq \Delta(G) + 1$。
- **Brooks 定理** 若 G 不为完全图或奇环，则 $\chi(G) \leq \Delta(G)$。

> **证明** 设 $|V(G)| = n$，考虑数学归纳法。
>
> 首先，$n \leq 3$ 时，命题显然成立。
>
> 根据归纳法，假设对于 $n-1$ 的命题成立。
>
> 不妨只考虑 $\Delta(G)$-正则图，因为对于非正则图来说，可以看作在正则图里删去一些边构成的，而这一过程并不会影响结论。
>
> 对于任意不是完全图也不是奇圈的正则图 G，任取其中一点 v，考虑子图 $H = G - v$，由归纳假设知 $\chi(H) \leq \Delta(H) \leq \Delta(G)$，接下来我们只需证明在 H 中插入 v 不会影响结论即可。
>
> 若 $\Delta(H) < \Delta(G)$，无需再做证明，我们只考虑 $\Delta(H) = \Delta(G)$ 的情况。
>
> 令 $\Delta = \Delta(G)$，设 H 染的 C 种颜色分别为 $c_1, c_2, \ldots, c_\Delta$，$v$ 的 Δ 个邻接点为 $v_1, v_2, \ldots, v_\Delta$。若 v 的邻接点个数不足 Δ 个或存在任意两点颜色相同，同样无需再做证明。
>
> 设所有在 H 中染成 c_i 或 c_j 的点以及它们之间的所有边构成子图 $H_{i,j}$。不妨假设任意 2 个不同的点 v_i, v_j 一定在 $H_{i,j}$ 的同一个连通分量中，否则若在两个连通分量中的话，可以交换其中一个连通分量所有点的颜色，从而使 v_i, v_j 颜色相同，即能有多余的颜色对 v 进行染色，无需再做证明。
>
> > 这里的交换颜色指的是若图中只有两种颜色 a, b，那么把图中原来染成颜色 a 的点全部染成颜色 b，把图中原来染成颜色 b 的点全部染成颜色 a。
>
> 我们设上述连通分量为 $C_{i,j}$，取出 $C_{i,j}$ 中一条路径记作 $P_{i,j}$，则恒有 $C_{i,j} = P_{i,j}$。因为 v_i 在 H 中的度为 $\Delta - 1$，所以 v_i 在 H 中的邻接点颜色一定两两不同，否则可以给 v_i 染别的颜色，从而和 v 的其他邻接点颜色重复，所以 v_i 在 $C_{i,j}$ 中邻接点数量为 1。若 $C_{i,j} \neq P_{i,j}$，设在 $C_{i,j}$ 中从 v_i 开始沿着 $P_{i,j}$ 遇到的第一个度数大于 2 的点为 u，注意到 u 的邻接点最多只用了 $\Delta - 2$ 种颜色，所以 u 可以重新染色，从而使 v_i, v_j 不连通。
>
> 沿用这一技术，我们可以证明对于 3 个不同的点 v_i, v_j, v_k，$V(C_{i,j}) \cap V(C_{j,k}) = \{v_j\}$。假设存在 $w \in V(C_{i,j}) \cap V(C_{j,k})$，若 $w \neq v_j$，则 w 必被染色为 c_j，且恰有两个被染色为 c_i 的邻接点和两个被染色为 c_j 的邻接点，注意到 w 的邻接点最多只用了 $\Delta - 2$ 种颜色，所以 w 同样可以重新染色。
>
> 若 v 的邻接点两两相邻，则必有 $\Delta = n$，即 G 为完全图。否则不妨设 v_1, v_2 不相邻，在 $C_{1,2}$ 取 v_1 的邻接点 w，交换 $C_{1,3}$ 中的颜色，则 $w \in V(C_{1,2}) \cap V(C_{2,3})$，与上述结论矛盾。
>
> 至此命题证明完毕。

斯坦纳树

- 以以下题面为例，主要把握其思想：
  - 给定连通图 G 中的 n 个点与 k 个关键点，求能包含 k 个关键点的生成树的最小边权，$k \ll n$。
- 设 $f_{x,S}$ 表示以 x 为根且包含关键点集合 S 的生成树的最小边权，在外层从小到大枚举 S，转移分为两类：
  - $f_{x,S} = \min\{f_{x,S}, f_{x,S-T} + f_{x,T}\}$

    - 松弛，可通过 Dijkstra 或 SPFA 实现，$f_{x,S}=\min\{f_{x,S},f_{y,S}+w(x,y)\}$
- 总时间复杂度 $\mathcal{O}(2^k(n+m)\log(n+m)+3^kn)$。

竞赛图

- **定义** 将 n 个点的无向完全图任意定向即可得到竞赛图。
- **性质1** 竞赛图缩点后呈链状，即求出竞赛图缩点后的拓扑序后，任意两个强连通分量之间的连边一定都是从拓扑序小的连向拓扑序大的。

> **证明** 对强连通分量个数归纳，若新增的强连通分量拓扑序最大，则所有连向其的边同向。

- **性质2** 竞赛图中每个强连通分量中存在哈密顿回路。

> **证明** 对点数 $n(n\geq 3)$ 归纳，若新增的点连向原有 n 个点的所有边均同向，则这 $n+1$ 个点不构成强连通分量，否则总可以找到 n 个点中的两个点，使得它们在哈密顿回路上相邻且连向新增点的方向相反。

- **性质3** 竞赛图中存在一条哈密顿路径。

> **证明** 因为属于不同强连通分量间的点均有连边，由 **性质2** 取每个强连通分量中的哈密顿回路相连即可。

- **性质4** 对于点数为 n 的强连通竞赛图，$\forall 3\leq l\leq n$，其一定存在长度为 l 的简单环。

> **证明** 考虑对点数 n 归纳，由 **性质1**，删除第 n 个点后原图变为若干个强连通分量的链状图，由归纳条件，每个强连通分量均可以构造出不超过点数的简单环，且第 n 个点一定有指向拓扑序最小的强连通分量的边，拓扑序最大的强连通分量一定有指向 n 的边，不难得到构造长度不超过 n 的简单环的方案。

- **性质5** 在竞赛图中若点 u 的出度大于等于点 v 的出度，则 u 一定可以到达 v。

> **证明** 考虑证明其逆否命题，由 **性质1**，仅需证明 u 所在强连通分量拓扑序大于 v 所在强连通分量拓扑序的情况，此时显然 v 的出度大于 u 的出度。

- **兰道定理** 将竞赛图的出度序列排序后得到 $s_1,s_2,\ldots,s_n$，该序列合法当且仅当 $\forall 2\leq k\leq n,\sum\limits_{i=1}^{k}s_i\geq\binom{k}{2}$ 且 $\sum\limits_{i=1}^{n}s_i=\binom{n}{2}$。

> **证明** 其必要性显然，因为任取点数为 k 的导出子图都满足该条件。考虑证明其充分性，构造一个所有边均是大点连向小点的竞赛图，则其 $s'_i=i-1$，上述不等式可变形为 $\sum\limits_{i=1}^{k}s_i\geq\sum\limits_{i=1}^{k}s'_i$，现在尝试通过不断调整该图，使得每个不等式均能取等号，每次操作如下：
>
> - 找到第一个 x 使得 $s_x>s'_x$。
> - 在 x 后找到第一个 z 使得 $s_z<s'_z$
> - 因而有 $s'_z>s_z\geq s_x>s'_x$，即 $s'_z-s'_x\geq 2$，此时必存在点 y，使得边 $z\to y$ 和 $y\to x$ 存在，将这两条边反向，则 s'_z 减少 1，s'_x 增加 1，s'_y 不变，原不等式仍成立。

强连通竞赛图计数

- 设 f_n 表示 n 个点的强连通竞赛图个数，考虑通过容斥原理计算 f_n，枚举拓扑序最靠前的强连通分量的大小，则剩余点连向该强连通分量的边的方向固定，而剩余点内部的连边是任意的，则对于 $n\geq 1$，有：

$$f_n=2^{\binom{n}{2}}-\sum_{i=1}^{n-1}\binom{n}{i}f_i2^{\binom{n-i}{2}}$$

$$\frac{2^{\binom{n}{2}}}{n!}=\sum_{i=1}^{n}\frac{f_i}{i!}\frac{2^{\binom{n-i}{2}}}{(n-i)!}$$

- 设 $F(x)=\sum\limits_{n\geq 1}\dfrac{f_n}{n!}x^n$，$G(x)=\sum\limits_{n\geq 0}\dfrac{2^{\binom{n}{2}}}{n!}x^n$，则上式可表达为：

$$G=FG+1$$
$$F=1-\frac{1}{G}$$

- 通过多项式求逆计算出 F 即可。