连通性

前置芝士

无向图

若图 G 为无向图，则边集合 E 中的每个元素为无序二元组 (u,v)，为无向边，u,v 一定可互达。

有向图

若图 G 为有向图，则边集合 E 中的每个元素为有序二元组 (u,v)，写作 $u\to v$，为有向边，u 指向 v，u 可达 v，v 不一定可达 u。

带权图

如果对于图 G 内每条边 e_i，都被赋予权值（边权）w_i，则称 G 为带权图。

> 如果这些权值都是正实数，就称 G 为正权图。

强连通分量（有向图）

强连通定义

有向图 G 强连通是指，G 中任意两个结点连通。

强连通分量

根据某度解释，q有向图中最大的强连通子图。

Tarjan 算法

在 Tarjan 算法中为每个结点 u 维护了以下几个变量：

dfn_u：深度优先搜索遍历时结点 u 被搜索的次序。

low_u：在 u 的子树中能够回溯到的最早的已经在栈中的结点。设以 u 为根的子树为 $Subtree_u$。low_u 定义为以下结点的 dfn $Subtree_u$ 中的结点；从 $Subtree_u$ 通过一条不在搜索树上的边能到达的结点。

根据深度优先搜索的性质，一个结点的子树内结点的 dfn 都大于该结点的 dfn。所以从根开始的一条路径上的 dfn 严格递增，low 严格非降。

按照深度优先搜索算法搜索的次序对图中所有的结点进行搜索，维护每个结点的 dfn 与 low 变量，且让搜索到的结点入栈。每当找到一个强连通元素，就按照该元素包含结点数目让栈中元素出栈。在搜索过程中，对于结点 u 和其相邻结点 v 满足 $fa_u \neq v$，有以下三种情况

1. v 未被访问：继续往下递归 v。在回溯过程中，用 low_v 更新 low_u。
2. v 被访问过，已经在栈中:根据 low 值的定义，用 dfn_v 更新 low_u。
3. v 被访问过，已不在栈中：连通分量已处理，不做任何操作。

关键代码：

```
int dfn[N], low[N], dfncnt, s[N], in_stack[N], tp;
int scc[N], sc;  // 结点 i 所在 SCC 的编号
int sz[N];       // 强连通 i 的大小

void tarjan(int u) {
  low[u] = dfn[u] = ++dfncnt, s[++tp] = u, in_stack[u] = 1;
  for (int i = h[u]; i; i = e[i].nex) {
    const int &v = e[i].t;
    if (!dfn[v]) {
      tarjan(v);
      low[u] = min(low[u], low[v]);
    } else if (in_stack[v]) {
      low[u] = min(low[u], dfn[v]);
    }
  }
  if (dfn[u] == low[u]) {
    ++sc;
    do {
      scc[s[tp]] = sc;
```

```
            sz[sc]++;
            in_stack[s[tp]] = 0;
        } while (s[tp--] != u);
    }
}
```

例题：

洛谷 P3916 图的遍历

题解：

洛谷 P3916 图的遍历

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

const int N = 1e5 + 10;

int n, m, x, y, f[N];
vector<int> a[N];

void dfs(int i, int x){
    if(f[i]){
        return;
    }
    f[i] = x;
    for(int j = 0; j < a[i].size(); j++){
        dfs(a[i][j], x);
    }
    return;
}

int main(){
    cin >> n >> m;
    for(int i = 1; i <= m; i++){
        cin >> x >> y;
        a[y].push_back(x);
    }
    for(int i = n; i >= 1; i--){
        dfs(i, i);
```

```
    }
    for(int i = 1; i <= n; i++){
        cout << f[i] << " ";
    }
    return 0;
}
```

dfs（无向图）

相较于有向图，简单许多。

从任意结点开始做 dfs，如果能一次就将所有的点遍历到，就说明此无向图连同；否则，图不连通，且未被访问到的节点属于其他连通分量。

拓扑排序

定义

在有向无环图上，对于任意一条有向边 $u \to v$，u 在拓扑序中

求解拓扑序

Kahn 算法

每次寻找一个入度为 0 的点，就将其插入到拓扑序中，同时删除此点和其出边。这个过程可以使用队列实现。

算法步骤如下：

- 一开始存储所有入度为 0 的结点到容器中。
- 重复以下操作直到 q 容器为空：
  - 从 q 中取出一个结点，加入到拓扑序中。
  - 删除结点 u 以及其出边 $u \to v$：
    - v 的入度减一。
    - 如果 v 的入度为 0，将其加入 q 中。

关键代码：

```
int n, m;
bool flag;  // 有向图是否有环
vector<int> ans, g[MAXN];
int ind[MAXN];  // 入度
queue<int> que;

void topo_sort() {
  for (int i = 1; i <= n; i++) {
    if (!ind[i]) {
      que.push(i);
    }
  }
  while (!que.empty()) {
    int u = que.front();
    que.pop();
    ans.push_back(u);
    for (int v : g[u]) {
      ind[v]--;
      if (!ind[v]) {
        que.push(v);
      }
    }
  }
  for (int x : ans) {
    cout << x << ' ';
  }
}
```

例题：

注意一下，以下题目需要在拓扑序的基础上跑dp

CSES 1681 Game Routes

洛谷 P1807 最长路

题解：

CSES 1681 Game Routes

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

const int N = 1e5 + 10, mod = 1e9 + 7;

int n, m;
vector<int> g[N];
int ind[N], dp[N];
queue<int> q;

void kahn(){
  for(int i = 1; i <= n; i++){
    if(!ind[i]){
      q.push(i);
    }
  }
  while(!q.empty()){
    int now = q.front();
    q.pop();
    for(int i : g[now]){
      dp[i] += dp[now];
      dp[i] %= mod;
      ind[i]--;
      if(!ind[i]){
        q.push(i);
      }
    }
  }
}

int main(){
  cin >> n >> m;
  for(int x, y, i = 1; i <= m; i++){
    cin >> x >> y;
    ind[y]++;
    g[x].push_back(y);
  }
  dp[1] = 1;
  kahn();
  cout << dp[n];
  return 0;
}
```

洛谷 P1807 最长路

```cpp
#include<bits/stdc++.h>
using namespace std;
using ll = long long;

const ll N = 1e3 + 510, inf = 5e12;

struct P{
  int v, w;
};

int n, m;
ll dis[N];
vector<P> g[N];

ll dfs(int x){
  if(dis[x] == inf){
    dis[x] = -inf;
    for(P i : g[x]){
      ll t = dfs(i.v);
      if(t != -inf){
        dis[x] = max(dis[x], t + i.w);
      }
    }
  }
  return dis[x];
}

int main(){
  cin >> n >> m;
  for(int u, v, w, i = 1; i <= m; i++){
    cin >> u >> v >> w;
    g[u].push_back({v, w});
  }
  fill(dis + 1, dis + n, inf);
  ll ans = dfs(1);
  cout << (ans == -inf ? -1 : ans);
  return 0;
```

```
}
```

欧拉路

定义

欧拉路径（Eulerian path）是经过图中每条边恰好一次的路径；

欧拉回路（Eulerian circuit）是经过图中每条边恰好一次的回路；

如果一个图中存在欧拉回路，则这个图被称为欧拉图（Eulerian graph）。

欧拉路径（回路）判定（是否存在）：

- 有向图欧拉路径：图中恰好存在 1 个点出度比入度多 1（这个点即为 起点 S），1 个点入度比出度多 1（这个点即为 终点 T），其余节点出度=入度。
- 有向图欧拉回路：所有点的入度=出度（起点 S 和终点 T 可以为任意点）。
- 无向图欧拉路径：图中恰好存在 2 个点的度数是奇数，其余节点的度数为偶数，这两个度数为奇数的点即为欧拉路径的 起点 S 和 终点 T。
- 无向图欧拉回路：所有点的度数都是偶数（起点 S 和终点 T 可以为任意点）。

寻找具体欧拉路径（回路）

在确定存在欧拉路径（回路）之后，按照题目要求跑 dfs 最后输出即可。

例题：

洛谷 P7771 【模板】欧拉路径

洛谷 P2731 [USACO3.3] 骑马修栅栏 Riding the Fences

CF 1981D Turtle and Multiplication

题解：

洛谷 P7771 【模板】欧拉路径

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

const int N = 2e5 + 10;
int n, m, nxt[N], oud[N], ind[N];
vector<int> st, g[N];

void dfs(int x){
  for(int i = nxt[x]; i < g[x].size(); i = nxt[x]){
    nxt[x] = i + 1;
    dfs(g[x][i]);
  }
  st.push_back(x);
}

int main(){
  cin >> n >> m;
  for(int i = 1, u, v; i <= m; i++){
    cin >> u >> v;
    g[u].push_back(v);
    ind[v]++;
    oud[u]++;
  }
  for(int i = 1; i <= n; i++){
    sort(g[i].begin(), g[i].end());
  }
  bool f = 1;
  for(int i = 1; i <= n; i++){
    if(ind[i] != oud[i]){
      f = 0;
    }
  }
  if(f){
    dfs(1);
    reverse(st.begin(), st.end());
    for(int i : st){
      cout << i << " ";
    }
    return 0;
  }
  int start = -1, fini = -1;
  for(int j = 1; j <= n; j++){
```

```
    if(oud[j] == ind[j] + 1){
      start = j;
      break;
    }
  }
  for(int j = 1; j <= n; j++){
    if(oud[j] == ind[j] - 1){
      fini = j;
      break;
    }
  }
  if(start == -1 || fini == -1){
    cout << "No";
    return 0;
  }
  for(int i = 1; i <= n; i++){
    if(i != start && i != fini){
      if(oud[i] != ind[i]){
        cout << "No";
        return 0;
      }
    }
  }
  dfs(start);
  reverse(st.begin(), st.end());
  for(int i : st){
    cout << i << " ";
  }
  return 0;
}
```

洛谷 P2731 [USACO3.3] 骑马修栅栏 Riding the Fences

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

const int N = 3000;

struct P{
  int o, v;
};
```

```
int cnt, edge[N], vis[N], to[N], d[N];
vector<P> g[N];
int m, n = 500, fl[N], mn = 510;

void addedge(int u, int v){
  edge[cnt] = v;
  g[u].push_back({cnt, v});
  cnt++;
}

vector<int> st;

void dfs(int x){
  for(int i = to[x]; i < g[x].size(); i = to[x]){
    to[x] = i + 1;
    int id = g[x][i].o;
    if(vis[id]) continue;
    int nxt = edge[id];
    vis[id] = 1;
    vis[id ^ 1] = 1;
    dfs(nxt);
  }
  st.push_back(x);
}

bool cmp(const P &a, const P &b){
  return a.v < b.v;
}

int main(){
  cin >> m;
  for(int i = 1, u, v; i <= m; i++){
    cin >> u >> v;
    mn = min({mn, u, v});
    fl[u] = fl[v] = 1;
    d[u]++, d[v]++;
    addedge(u, v);
    addedge(v, u);
  }
  for(int i = 1; i <= n; i++){
    if(fl[i])
```

```
        sort(g[i].begin(), g[i].end(), cmp);
    }
    bool f = 1;
    for(int i = 1; i <= n; i++){
        if(!fl[i]) continue;
        if(d[i] & 1){
            f = 0;
            break;
        }
    }
    if(f){
        dfs(mn);
        reverse(st.begin(), st.end());
        for(int i : st){
            cout << i << "\n";
        }
        return 0;
    }
    int start = -1;
    for(int i = 1; i <= n; i++){
        if(!fl[i]) continue;
        if(d[i] & 1){
            start = i;
            break;
        }
    }
    dfs(start);
    reverse(st.begin(), st.end());
    for(int i : st){
        cout << i << "\n";
    }
    return 0;
}
```

CF 1981D Turtle and Multiplication

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

const int N = 4e6 + 10;
```

```
int T, n, to[N], vis[N];

int v[N], prm[N], tot, ttt = 2;

struct Graph{
  int cnt = 0, edge[N];
  vector<int> g[N];
  void init(int x){
    fill(edge, edge + cnt, 0);
    for(int i = 0; i <= x; i++){
      g[i].clear();
    }
    fill(to + 1, to + 1 + x, 0);
    fill(vis, vis + cnt, 0);
    cnt = 0;
  }
  void addedge(int u, int v){
    edge[cnt] = v;
    g[u].push_back(cnt);
    cnt++;
  }
}t;

void get(int x) {
  for (; tot < x; ttt++){
    if (!v[ttt]) {
      prm[++tot] = ttt;
    }
    for (int j = 1; j <= tot && ttt * prm[j] <= 1e6; j++) {
      v[ttt * prm[j]] = 1;
      if (ttt % prm[j] == 0) {
        break;
      }
    }
  }
  return ;
}

vector<int> st;

void dfs(int x){
  for(int i = to[x]; i < t.g[x].size(); i = to[x]){
```

```
    to[x] = i + 1;
    int id = t.g[x][i];
    if(vis[id]) continue;
    int nxt = t.edge[id];
    vis[id] = 1;
    vis[id ^ 1] = 1;
    dfs(nxt);
  }
  st.push_back(x);
}

void solve(){
  cin >> n;
  int x = 1;
  for(; x * (x + 1) / 2 - (x / 2 - 1) * (x % 2 == 0) + 1 < n; x++);
  get(x);
  if(x % 2 == 0){
    for(int i = 1; i <= x; i++){
      for(int j = i; j <= x; j++){
        if(i % 2 == 0 && j == i + 1) continue;
        t.addedge(i, j);
        t.addedge(j, i);
      }
    }
  }
  else{
    for(int i = 1; i <= x; i++){
      for(int j = i; j <= x; j++){
        t.addedge(i, j);
        t.addedge(j, i);
      }
    }
  }
  dfs(1);
  t.init(x);
  reverse(st.begin(), st.end());
  for(int i = 0; i < n; i++){
    cout << prm[st[i]] << " ";
  }
  st.clear();
  cout << "\n";
}
```

```
int main(){
  cin >> T;
  while(T--){
    solve();
  }
  return 0;
}
```

最短路

类型

单源最短路

给定图 $G = (V, E)$ 和一个点 S，单源最短路径的目标是求得从源点 S 到其他每个顶点的最短路径长度 $D(u)$。

多源最短路

从多个起始点出发，求到达图中其他点的最短路径。

正题

前置芝士

对于一条 $u \to v$ 的边，定义：

- 三角形不等式：如果 $D(v) \le D(u) + w(u, v)$，则称改变满足三角形不等式。
- 松弛操作：如果 $D(v) > D(u) + w(u, v)$，则 $D(v) \leftarrow D(u) + w(u, v)$

从转移的角度看，状态为 (u, l)，表示路径终点为 u、长度为 l，转移就是通过一条 $u \to v$ 的边移动到另一个点 v 上的。

Dijkstra

dijkstra 算法适用于边权都是非负数的图，步骤如下：

1. 对于源点 S，$D(S)=0$。对于其他点 u，$D(u)=+\infty$。其中 $D(u)$ 为上文所示。
2. 不断重复以下操作 n 次（$n-1$）次，直到所有点都被标记过。
3. 找出一个点 u，u 是在所有未被标记过的点中 $D(u)$ 最小的，然后标记点 u 为已求解过最短路径的结点。即距离所有未标记的点中距离源点最近的点。
4. 对于起点为 u 的所有边 $u \to v$，如果该边不满足三角不等式，则执行松弛操作。

对于第 3 步，可以暴力寻找 u，但这会使时间复杂度变为 $\mathcal{O}(n^2)$，不符和我们的~~要求~~审美。所以可以使用堆（优先队列）优化，将时间复杂度降为 $\mathcal{O}((n+m)\log n)$，可以通过大部分题目。

floyd

例题：

洛谷 P4779 【模板】单源最短路径（标准版）

洛谷 B3647 【模板】Floyd

题解：

洛谷 P4779 【模板】单源最短路径（标准版）

```
#include<bits/stdc++.h>
using namespace std;
using pii = pair<int, int>;
#define w first
#define u second

const int N = 1e5 + 10;

struct P{
  int v, w;
};

int n, m, s;
vector<P> g[N];
```

```
int dis[N], vis[N];

priority_queue<pii, vector<pii>, greater<pii> > q;

void dij() {
  fill(dis, dis + n + 1, INT_MAX);
  dis[s] = 0;
  q.push({0, s});
  while(!q.empty()){
    pii now = q.top();
    q.pop();
    if(vis[now.u]){
      continue;
    }
    vis[now.u] = 1;
    for(P e : g[now.u]){
      if(dis[e.v] > now.w + e.w){
        dis[e.v] = now.w + e.w;
        q.push({dis[e.v], e.v});
      }
    }
  }
}

int main(){
  cin >> n >> m >> s;
  for(int i = 1, u, v, w; i <= m; i++){
    cin >> u >> v >> w;
    g[u].push_back({v, w});
  }
  dij();
  for(int i = 1; i <= n; i++){
    cout << dis[i] << " ";
  }
  return 0;
}
```

洛谷 B3647 【模板】Floyd

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

```

```cpp
using ll = long long;

const int N = 110;
ll D[N][N];

int main(){
  int n, m;
  cin >> n >> m;
  for(int i = 1; i <= n; i++){
    for(int j = 1; j <= n; j++){
      D[i][j] = INT_MAX;
      if(i == j){
        D[i][j] = 0;
      }
    }
  }
  for(int i = 1, u, v, w; i <= m; i++){
    cin >> u >> v >> w;
    D[u][v] = min(w * 1ll, D[u][v]);
    D[v][u] = min(w * 1ll, D[v][u]);
  }
  for(int k = 1; k <= n; k++){
    for(int i = 1; i <= n; i++){
      for(int j = 1; j <= n; j++){
        D[i][j] = min(D[i][j], D[i][k] + D[k][j]);
      }
    }
  }
  for(int i = 1; i <= n; i++){
    for(int j = 1; j <= n; j++){
      cout << D[i][j] << " ";
    }
    cout << "\n";
  }
  return 0;
}
```

生成树(最小生成树)

定义

生成子图：子图中的点集为原图中的点集。

生成树：无向连通图的生成子图，且该子图为树。也就是在原图中选择 $n-1$ 条边和互相连通的 n 个点构成的子图。类似地有生成森林的概念。

最小生成树：边权之和最小的生成树。不同于最短路，最小生成树仅要求图是简单图。

如果是非连通图，那就是生成森林。

对于非简单图，去掉自环，保留重边中最小边权的边，即可转换为简单图。

在生成树中和不在生成树中的边称为树边、非树边。

Kruskal 算法

Kruskal 算法是一种常见并且好写的最小生成树算法，由 Kruskal 发明。该算法的基本思想是从小到大加入边，是个贪心算法。

步骤如下：

- 初始时有一个包含 N 个孤立点的图 G。
- 按边权从小到大的顺序处理每条边，如果当前边加入 G 后不成环则加入到 $G 中，否则不加入。换句话说，就是当前边连接的两点如果处于同一连通块，则不加边。
- 如果加入了 $n-1$ 条边，说明最小生成树存在。
- 使用并查集判断是否成环（判断两个端点是否在同一连通块当中）。

Prim 算法

从点的角度构造最小生成树

- 随便选一个点放入生成树中。
- 不断选取离当前生成树距离最近的且不在生成树中的点加入树中。
    - 距离一般为最短距离的简称。

    - 点到生成树的距离，就是点到生成树中每个点的最短距离的最小值。
    - 这个距离会对应一条边，同时将点和边加入到树中。
- 每当新的点加入最小生成树，用它来更新不在生成树中的其他点到最小生成树的距离。
    - 其他点到最小生成树的距离需要考虑多一个点，即当前新加入的点。
    - 不应该更新已在生成树中的点。

例题：

洛谷 P4047 [JSOI2010] 部落划分

洛谷 P1396 营救

题解：

洛谷 P4047 [JSOI2010] 部落划分

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

const int N = 1e3 + 10, M = 1e3 + 10;

int n, k, fa[N], f[N], sz[N], cnt;

struct Node{
  int x, y;
}a[M];

struct P{
  int x, y;
  long double w;
}p[N * N];

int Find(int x){
  return fa[x] == x ? x : fa[x] = Find(fa[x]);
}

void Merge(int i){
  int A = Find(p[i].x), B = Find(p[i].y);
  if(A != B){
```

```
    if(sz[A] > sz[B]){
      swap(A, B);
    }
    fa[A] = B, sz[B] += sz[A];
    cnt--;
  }
}

long double gg(int u, int v){
  return sqrt((long double)((a[u].x - a[v].x) * (a[u].x - a[v].x) + 
(a[u].y - a[v].y) * (a[u].y - a[v].y)));
}

bool cmp(const P &a, const P &b){
  return a.w < b.w;
}

int main(){
  cin >> n >> k;
  for(int i = 1; i <= n; i++){
    cin >> a[i].x >> a[i].y;
  }
  for(int i = 1; i <= n; i++){
    for(int j = 1; j <= n; j++){
      p[(i - 1) * n + j] = {i, j, gg(i, j)};
    }
  }
  sort(p + 1, p + 1 + n * n, cmp);
  fill(sz + 1,sz + 1 + n, 1);
  iota(fa + 1, fa + 1 + n, 1);
  cnt = n;
  for(int i = 1; i <= n * n; i++){
    Merge(i);
    if(cnt < k){
      cout << fixed << setprecision(2) << p[i].w;
      return 0;
    }
  }
  return 0;
}
```

洛谷 P1396 营救

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

const int N = 1e4 + 10, M = 2e4 + 10;

int n, m, fa[N], s, t, f[N], sz[N];

struct P{
  int x, y, w;
}a[M];



int Find(int x){
  return fa[x] == x ? x : fa[x] = Find(fa[x]);
}

bool cmp(const P &a, const P &b){
  return a.w < b.w;
}

void Merge(int A, int B){
  if(A != B){
    if(sz[A] > sz[B]){
      swap(A, B);
    }
    fa[A] = B, sz[B] += sz[A];
  }
}

int main(){
  cin >> n >> m >> s >> t;
  for(int i = 1; i <= m; i++){
    cin >> a[i].x >> a[i].y >> a[i].w;
  }
  if(s == t){
    cout << 0;
    return 0;
  }
  sort(a + 1, a + 1 + m, cmp);
```

```
    iota(fa + 1, fa + 1 + n, 1);
    fill(sz + 1, sz + 1 + n, 1);
    for(int i = 1; i <= m; i++){
        int FA = Find(a[i].x), FB = Find(a[i].y);
        Merge(FA, FB);
        if(Find(s) == Find(t)){
            cout << a[i].w;
            return 0;
        }
    }
    return 0;
}
```