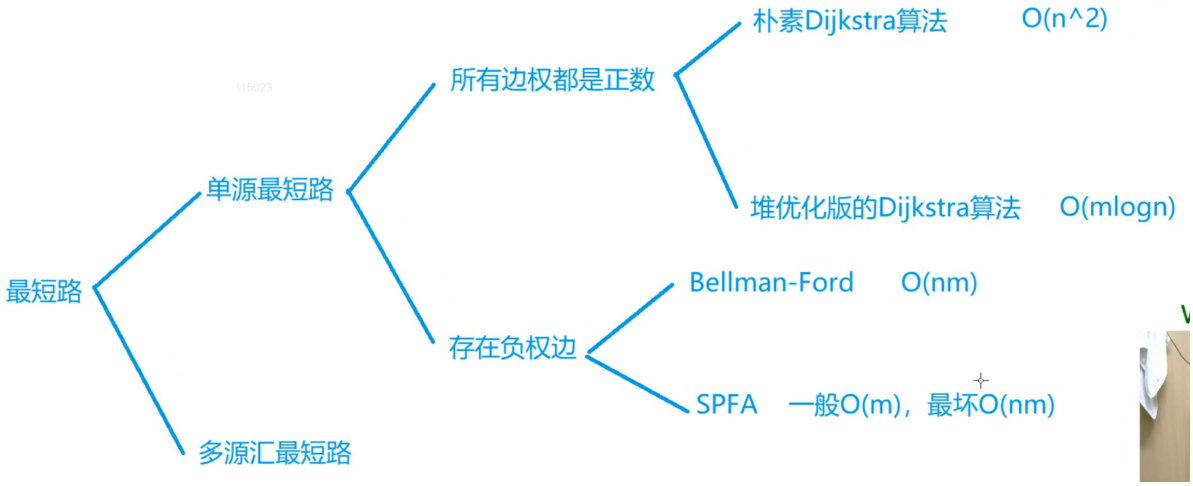
# 图论最短路




## Dijkstra(朴素版)

### 算法作用

求稠密图的单源最短路

使用邻接矩阵存图

### 算法思路

1. 初始化距离为正无穷，dis[1] = 0;
2. 循环迭代n次，每次可以确定一个点
    1. 遍历该点的每一个节点，找到不在close_set中距离集合最近的点
    2. 标记选中该点
    3. 用该点更新其他点的距离 `dis[j] = min(dis[j], dis[t] + g[t][j]);`

### 代码

```cpp
#include<bits/stdc++.h>
using namespace std;
const int maxn = 500 + 10;

int g[maxn][maxn];
int dis[maxn];
int n, m;
bool st[maxn];

int dijkstra(){
    memset(dis, 0x3f, sizeof dis);
    dis[1] = 0;
    for(int i = 0; i < n; i++){
        int t = -1;
        for(int j = 1; j <= n; j++){
            if(!st[j] && (t == -1 || dis[j] < dis[t])){
                t = j;
            }
```

```
        }
        st[t] = true;
        for(int j = 1; j <= n; j++){
            dis[j] = min(dis[j], dis[t] + g[t][j]);
        }
    }

    if(dis[n] == 0x3f3f3f3f) return -1;
    return dis[n];
}

int main(){
    cin >> n >> m;
    memset(g, 0x3f, sizeof g);
    for(int i = 0; i < m; i++){
        int x, y, z; cin >> x >> y >> z;
        g[x][y] = min(g[x][y], z);
    }

    cout << dijkstra() << endl;

    return 0;
}
```

# Dijkstra(堆优化版)

无法求最长路

## 算法作用

不含负权边的单源最短路

稀疏图使用邻接表存储

## 算法思路

将枚举过程替换为优先队列

## 代码

```
#include<iostream>
#include<cstring>
#include<algorithm>
#include<queue>

using namespace std;
typedef pair<int, int> PII; //<离起点的距离,  节点编号>

const int N = 150010;
int h[N], e[N], ne[N], idx, w[N];
int dist[N];
bool st[N];
int n, m;

//在a节点之后插入一个b节点，权重为c
void add(int a, int b, int c) {
    e[idx] = b;
```

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

int dijkstra() {
//    所有距离初始化为无穷大
    memset(dist, 0x3f, sizeof dist);
//    1号节点距离为0
    dist[1] = 0;
//    建立一个小根堆
    priority_queue<PII, vector<PII>, greater<PII>> heap;
//    1号节点插入堆
    heap.push({0, 1});
    while (heap.size()) {
//        取出堆顶顶点
        auto t = heap.top();
//        并删除
        heap.pop();
//        取出节点编号和节点距离
        int ver = t.second, distance = t.first;
//        如果节点被访问过，则跳过
        if (st[ver])continue;
        st[ver] = true;
        for (int i = h[ver]; i != -1; i = ne[i]) {
//            取出节点编号
            int j = e[i];
//            dist[j] 大于从t过来的距离
            if (dist[j] > distance + w[i]) {

                dist[j] = distance + w[i];
                heap.push({dist[j], j});
            }
        }
    }
    if (dist[n] == 0x3f3f3f3f)return -1;
    return dist[n];

}

int main() {
    memset(h, -1, sizeof h);
    cin >> n >> m;
    while (m--) {
        int a, b, c;
        cin >> a >> b >> c;
        add(a, b, c);
    }
    cout << dijkstra() << endl;
    return 0;
}

```

## bellman_ford

## 算法作用

- 含有负权边图的最短路
- 一定步数以内的最短路

## 算法思路

`dis[], bk[]`

1. 初始化距离为无穷
2. 将起点的距离设置为1
3. 循环k-1次（最多点的个数）
    1. 将距离数组进行备份，防止串联
    2. 遍历所有的边
        1. 遍历到边目前的最短距离=min(本来到该点的距离，前置点到该点的距离+边权)
4. 如果到目标点的距离小于INF>>1, 没有路径。否则有路径。

## 代码

# SPFA

## 算法作用

- 类地杰斯特拉，优化版Ford（使用邻接表存图）
- 求负环

## 算法思路（最短路）

`st[], dis[], queue, 邻接表`

1. 初始化距离为正无穷，**起点距离为1**；
2. 新建队列保存要遍历的点。起点入队。
3. 将起点st标记，已经在队列中。
4. 如果队列不空
    1. 取出队头，弹出队头，取消标记
    2. 遍历所有临边
        1. 如果目标点距离>起始点+边权： 更新距离
            1. 如果该点不在队列中则入队，标记
5. 如果到目标点的距离等于原始距离，则无路径。

## 代码

```
#include<bits/stdc++.h>
using namespace std;
const int maxn = 1e6 + 10;

int h[maxn], e[maxn], ne[maxn], w[maxn], idx;
int dis[maxn];
```

```cpp
bool st[maxn];
int n, m;

void add(int a, int b, int c){
    e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx ++;
}

int spfa(){
    memset(dis, 0x3f, sizeof dis);
    dis[1] = 0;

    queue<int> q;
    q.push(1);
    st[1] = true;
    while(!q.empty()){
        int t = q.front(); q.pop(); st[t] = false;
        for(int i = h[t]; i != -1; i = ne[i]){
            int j = e[i];
            if(dis[j] > dis[t] + w[i]){ //注意w的idx为i！！！
                dis[j] = dis[t] + w[i];
                if(!st[j]){
                    st[j] = true;
                    q.push(j);
                }
            }
        }
    }

    return dis[n];
}

int main(){
    cin >> n >> m;
    memset(h, -1, sizeof h);
    for(int i = 0; i < m; i++)
        int x, y, z; cin >> x >> y >> z; add(x, y, z);

    //一定要先复制再判断！！因为最短路径可能是负数！！
    int ans = spfa();
    if(ans == 0x3f3f3f3f) cout << "impossible";
    else cout << ans;
    return 0;
}
```

## 算法思路（求负环）

cnt[]//记录路径长度

维护cnt数组，只要大于节点数，必有负环。

开始时需要把所有的点全部放在队列中。

不必初始化`dis[]`的距离为正无穷

！！！**使用stack可以优化效率效率**

## 代码

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

const int maxn = 1e4 + 10, N = 2010;
int e[maxn], ne[maxn], h[N], w[maxn], idx;
int n, m;
int dis[N], cnt[N];
bool st[N];

void add(int a, int b, int c){
    e[idx] = b, w[idx] = c, ne[idx] = h[a], h[a] = idx ++;
}

bool spfa(){
    queue<int> q;
    for(int i = 1; i <= n; i ++){
        q.push(i); st[i] = true;
    }

    while(q.size()){
        int t = q.front(); q.pop(); st[t] = false;
        for(int i = h[t]; i != -1; i = ne[i]){
            int j = e[i];

            if(dis[j] > dis[t] + w[i]){
                dis[j] = dis[t] + w[i];
                cnt[j] = cnt[t] + 1; //抽屉原理
                if(cnt[j] >= n) return true;
                if(!st[j])
                    q.push(j), st[j] = true;
            }
        }
    }
    return false;
}

int main(){
    cin >> n >> m;
    memset(h, -1, sizeof h);
    for(int i = 0; i < m; i ++){
        int a, b, c; cin >> a >> b >> c;
        add(a, b, c);
    }

    bool ans = spfa();
    if(ans) puts("Yes");
    else puts("No");

    return 0;
}
```

# Floyd

## 算法作用

用邻接矩阵存储图，求**多元汇**最短路。复杂度：$O(N^3)$

求任意两点之间的最短路径

传递闭包，判断两点

找最小环（SPFA找负环）

恰好经过k条边的最短路 倍增思想

## 算法思路

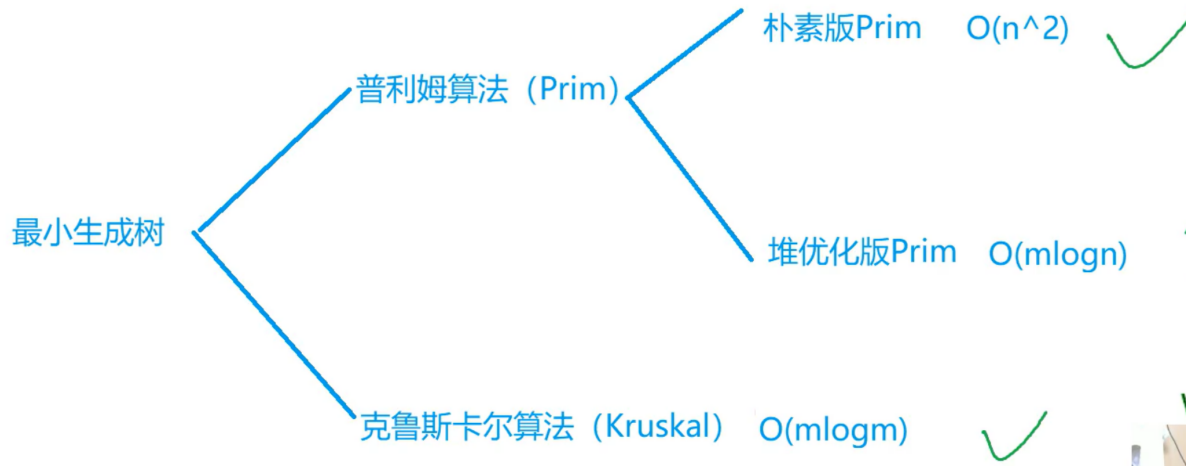
先打表，DP思想。

## 代码

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

using namespace std;

const int N = 210, INF = 1e9;

int n, m, Q;
int d[N][N];

void floyd(){
    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]);
            }
        }
    }
}
int main()
{
    ios::sync_with_stdio(false); cin.tie(0); cout.tie(0);
    cin >> n >> m >> Q;

    for (int i = 1; i <= n; i ++ ) //邻接表初始化，最短路问题故无视自环
        for (int j = 1; j <= n; j ++ )
            if (i == j) d[i][j] = 0;
            else d[i][j] = INF;

    while(m--){
        int a, b, c; cin >> a >> b >> c;
        d[a][b] = min(d[a][b], c);
    }

    floyd();

    while(Q--){
        int a, b;
        cin >> a >> b;
```

```
        int t = d[a][b];
        if(t > INF / 2) cout << "impossible" << endl;
        else cout << t << endl;
    }

    return 0;
}
```

# 最小生成树




## Prim

### 算法作用

稠密图的最小生成树：找到总代价最小的树，使图中的任意两点在同一树中。$O(n^2)$

### 算法思路

1. 将所有距离初始化为正无穷
2. 循环迭代n次，每次可以确定一个点
    1. 遍历每个节点，找到不在生成树的剩下点中，到树距离最小的点
    2. 标记选中该点
    3. 用该点更新其他点到生成树的距离（如果小于，则更新）

### 代码

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

const int maxn = 510, INF = 0x3f3f3f3f;
int g[maxn][maxn];
int n, m;
int dis[maxn];
bool st[maxn];

int prim(){
    memset(dis, 0x3f, sizeof dis);
    dis[1] = 0;
    int res = 0;
    for(int i = 0; i < n; i ++){
```

```
        int t = -1;
        for(int j = 1; j <= n; j ++)
            if(!st[j] && (t == -1 || dis[t] > dis[j])) t = j;

        if(dis[t] == INF) return INF;

        st[t] = true;
        res += dis[t];
        for(int j = 1; j <= n; j ++)
            dis[j] = min(dis[j], g[t][j]);

    }
    return res;
}

int main(){
    cin >> n >> m;
    memset(g, 0x3f, sizeof g); //重边和自环不影响最小生成树
    for(int i = 0; i < m; i ++){
        int u, v, w; cin >> u >> v >> w;
        g[u][v] = g[v][u] = min(g[u][v], w);
    }

    int t = prim();

    if(t == 0x3f3f3f3f) cout << "impossible" << endl;
    else cout << t << endl;

    return 0;
}

```

# Kruskal

## 算法作用

用于稀疏图的最小生成树算法。$O(mlogm)$

不需要使用邻接表或邻接矩阵存图

求**边权和**最小的最小生成树&&**最大边权最小**的最小生成树

## 算法思路

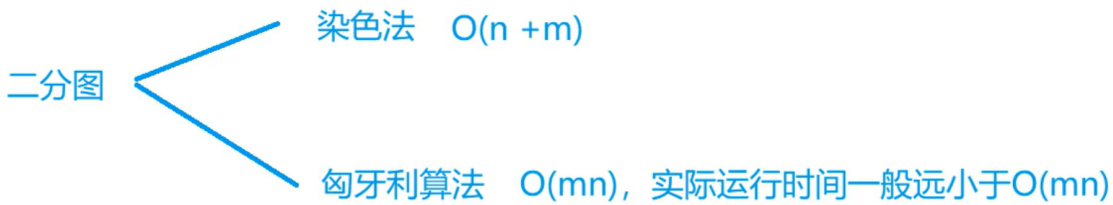
1. 将所有的边按边权从小到大排序
2. 枚举所有的边，对$u,v$做并查集，如不属于一个集合则合并。

## 代码

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

int n, m;
const int maxn = 1e5 +10;
int p[maxn];
struct Edge{
    int u, v, w;
```

```
    bool operator<(const Edge&e) const{
        return w < e.w;
    }
} edge[maxn];

int find(int x){
    if(x != p[x]) p[x] = find(p[x]);
    return p[x];
}

int main(){
    cin >> n >> m;
    for(int i = 1; i <= n; i ++) p[i] = i;

    for(int i = 0; i < m; i ++)
        cin >> edge[i].u >> edge[i].v >> edge[i].w;

    sort(edge, edge + m);

    int res = 0, cnt = 0;
    for(int i = 0; i < m; i ++){
        int u = edge[i].u, v = edge[i].v, w = edge[i].w;
        int a = find(u), b = find(v);
        if(a != b){
            res += w; cnt ++;
            p[a] = b;
        }
    }
    if(cnt < n - 1) cout << "impossible" << endl;
    else cout << res;

    return 0;
}
```

# 二分图




## 染色法

### 算法作用

判断一个图是不是二分图

一个图是二分图，当且仅当图中不含有奇数环（边的数量为奇数个的环）。

## 算法思路

遍历每一个点，如果没有被染色，则对它进行染色

- pii存储，第一个存储标号，第二个存储颜色
- 对于每个点，搜索与其相邻的点。如未染色，则染色入队。否则，判断其颜色状态是否合法，不合法返回false。

## 代码

```cpp
#include <bits/stdc++.h>
#define pii pair<int, int>
using namespace std;

int n, m;
const int maxn = 2e5 + 10;
int e[maxn], ne[maxn], h[maxn], idx;
int st[maxn];

void add(int a, int b){
    e[idx] = b, ne[idx] = h[a], h[a] = idx ++;
}

bool bfs(int x){
    queue<pii> q;
    q.push({x, 1}); //第一个是编号，第二个是颜色
    st[x] = 1;

    while(q.size()){
        int ver = q.front().first, color = q.front().second; q.pop();
        for(int i = h[ver]; i != -1; i = ne[i]){
            int j = e[i];

            if(!st[j]){ //未被染色，则染色
                st[j] = 3 - color;
                q.push({j, 3 - color});
            }
            else{ //已被染色则判断
                if(st[j] == color)  return false;
            }
        }
    }
    return true;
}

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

    cin >> n >> m;
    memset(h, -1, sizeof h);
    for(int i = 1; i <= m; i ++){
        int a, b; cin >> a >> b;
        add(a, b), add(b, a);
    }

    for(int i = 1; i <= m; i ++){
        if(!st[i])
```

```
            if(!bfs(i)){
                cout << "No" << endl;
                return 0;
            }
    }

    cout << "Yes" << endl;
    return 0;
}
```

# 匈牙利算法

## 算法作用

求二分图的最大匹配

## 算法思路

match[女生] = 男生 //表示女生对应的男生

st[女生] = true // 表示当前女生是否可以被选择（判重和回溯）

find(x) //找st状态下x是否有匹配的女生

1. 枚举每个男生，遍历对应的女生
2. 如果该女生已经被预定，跳过。否则，预定。
3. 如果女生没有对象||在st状态下可以换对象。则换对象，return true。

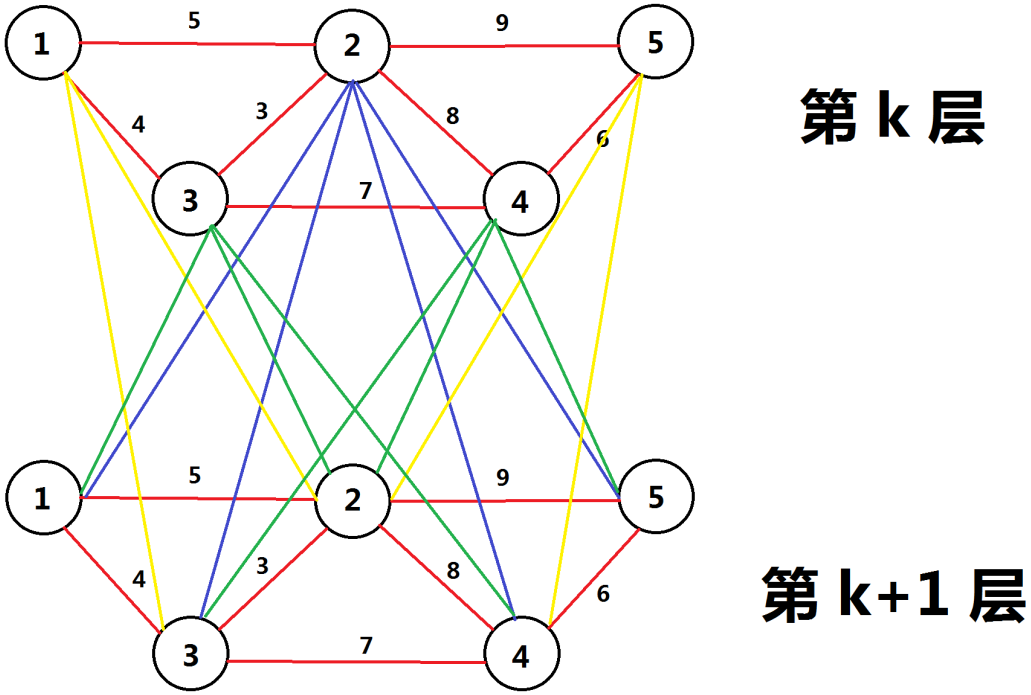
## 代码

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

int m, n1, n2;
const int M = 1e5 + 10, N = 510;
int e[M], ne[M], h[N], idx;
bool st[N];// st[a] = true 说明女生 a 目前被一个男生预定了
int match[N];//  match[a] = b: 女生 a 目前匹配了男生 b

void add(int a, int b){
    e[idx] = b, ne[idx] = h[a], h[a] = idx ++;
}

bool find(int x){// 为单身狗 x 找一个对象， (或) x的女朋友被别人预定，给x换一个对象
如果成功，返回true
    for(int i = h[x]; i != -1; i = ne[i]){// j 是可以与男生 x 匹配的女生之一
        int j = e[i];
        if(st[j]) continue;// 女生 j 目前被一个男生预定了，跳过它
        st[j] = true;// 将女生 j 预定给男生 x

         // 如果女生 j 没有对象， 或者
        // 女生 j 在前几轮深搜中已预定有对象，但我们成功给她的对象换了个新对象
        if(match[j] == 0 || find(match[j])){
            match[j] = x;
            return true;
        }
    }
```

```
    return false;
}

int main(){
    cin >> n1 >> n2 >> m;
    memset(h, -1, sizeof h);
    for(int i = 0; i < m; i ++){
        int u, v; cin >> u >> v;
        add(u, v);
    }

    int cnt = 0;
    for(int i = 1; i <= n1; i ++){
        memset(st, false, sizeof st);
        if(find(i)) cnt ++;
    }

    cout << cnt;

    return 0;
}
```

## 分层图

在图上，有k次机会可以直接通过一条边，问起点与终点之间的最短路径。




分层图的作用

经过分层后，我们得到了新图

我们可以发现，原本题目中选 k 条边免费的操作被我们等价了：

在从一个点到另一个点时，如果选择免费，就进入下一层，相当于进行一次免费操作

因为可以免费 k 次，所以我们要建 k+1 层图

在 k+1 层图上我们已经不能再往下了，即免费操作已用完

**k个免费操作即建k层**

## 建图

```
for(int i=1;i<=p;i++){
    int a,b,l; cin>>a>>b>>l;
    add(a,b,l), add(b,a,l);
    for(int j=1;j<=k;j++){
        add(a+(j-1)*n,b+j*n,0), add(b+(j-1)*n,a+j*n,0);//上层到下层的双向边
        add(a+j*n,b+j*n,l), add(b+j*n,a+j*n,l);//下层的双向边
    }
}
for(int i=1;i<=k;i++){
    add(i*n,(i+1)*n,0);
}
```

## 模板-dij

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

const int N=100100*42+10,M=500500*42+10; //容易爆空间
int e[M],ne[M],h[N],w[M],idx;
bool st[N];
int dis[N];
int n,p,k;

void add(int a,int b,int c){
    e[idx]=b,w[idx]=c,ne[idx]=h[a],h[a]=idx++;
}

void dij(){
    memset(dis,0x3f,sizeof dis);
    dis[1]=0;
    priority_queue<pii,vector<pii>,greater<pii>> pq;
    pq.push({0,1});

    while(pq.size()){
        auto t=pq.top(); pq.pop();
        int ver=t.second, d=t.first;
        if(st[ver]) continue;
        st[ver]=true;
        for(int i=h[ver];~i;i=ne[i]){
            int j=e[i];
            if(dis[j]>dis[ver]+w[i]){
                dis[j]=dis[ver]+w[i];
                pq.push({dis[j],j});
            }
        }
    }
}

int main(){
```

```
    cin>>n>>p>>k;
    memset(h,-1,sizeof h);
    for(int i=1;i<=p;i++){
        int a,b,l; cin>>a>>b>>l;
        add(a,b,l), add(b,a,l);
        for(int j=1;j<=k;j++){
            add(a+(j-1)*n,b+j*n,l/2), add(b+(j-1)*n,a+j*n,l/2);//上层到下层的双向边
            add(a+j*n,b+j*n,l), add(b+j*n,a+j*n,l);//下层的双向边
        }
    }
    for(int i=1;i<=k;i++){
        add(i*n,(i+1)*n,0);
    }

    dij();

    // cout<<dis[(k+1)*n]<<endl;
    int ans=dis[n];
    for(int i=1;i<=k;i++){
        // cout<<"### dis["<<i*n+n<<"]: "<<dis[i*n+n]<<endl;
        ans=min(dis[i*n+n],ans);
    }
    cout<<ans;

    return 0;
}
```

## 模板-SPFA

```
//地杰斯特拉无法求最长路
//分层图
#include<bits/stdc++.h>
#define pii pair<int,int>
using namespace std;

const int N=100000*3+10, M=100000*2*10;
int h[N],e[M],ne[M],w[M],idx;
int dis[N];
bool st[N];
int n,m;

void add(int a,int b,int c=0){
    e[idx]=b,w[idx]=c,ne[idx]=h[a],h[a]=idx++;
}

void spfa(){
    memset(dis,-0x3f,sizeof dis);
    queue<int>q;
    q.push(1);
    st[1]=true;
    dis[1]=0;

    while(q.size()){
        int t=q.front(); q.pop(); st[t]=false;
        for(int i=h[t];~i;i=ne[i]){
            int j=e[i];
```

```
            if(dis[j]<dis[t]+w[i]){
                dis[j]=dis[t]+w[i];
                if(!st[j]){
                    st[j]=true;
                    q.push(j);
                }
            }
        }
    }
}

int main(){
    cin>>n>>m;
    memset(h,-1,sizeof h);
    for(int i=1;i<=n;i++){
        int c; cin>>c;
        add(i,i+n,-c), add(i+n,i+2*n,c);
    }
    for(int i=1;i<=m;i++){
        int x,y,z; cin>>x>>y>>z;
        if(z==1){
            add(x,y), add(x+n,y+n), add(x+2*n,y+2*n);
        }
        else{
            add(x,y), add(x+n,y+n), add(x+2*n,y+2*n);
            add(y,x), add(y+n,x+n), add(y+2*n,x+2*n);
        }
    }

    spfa();

    cout<<dis[3*n]<<endl;

    return 0;
}
```