```python

################################
#           グラフ関係
################################
N # 頂点数
lines = defaultdict(set)
lines[s].add((t,c)) # s->t コストc の辺 0-indexed に直す

# ベルマンフォード
# 最短経路（負辺有り）and 負コスト回路検出
def BellmanFord(Start,Goal,lines,N):
    Costs=[INF]*N
    Costs[Start] = 0
    upd8s = [True]*N
    for i in range(2*N): #2N回ループ(負回路の検出までみる)
        will_upd8s = [False]*N
        upd8 = False
        for s in range(N):
            if not upd8s[s]: continue   #前回更新してないので見ない
            for t,c in lines[s]:
                if c + Costs[s] < Costs[t]:
                    Costs[t] = Costs[s]+c
                    upd8 = True
                    will_upd8s[t] = True #更新した点だけ次に見る
        if not upd8: #なにも更新しなかったら終わり
            return Costs[Goal]
        if i == N-1: #Nループ目のGoalのCostを記録
            tmp = Costs[Goal]
        upd8s = will_upd8s[:]

    if tmp != Costs[Goal]: return -INF
    else: return Costs[Goal]


# ダイクストラ
# 最短距離（負辺無し）
def Dijkustra(s,lines,N):
    weight = [INF]*N
    weight[s] = 0

    def search (s,w_0,q,weight): #s->t
        for line in list(lines[s]):
            t = line[0]
            w = w_0 + line[1]
            if weight[t] > w:
                heapq.heappush(q, [w,t])
                weight[t] = w

    q = [[0,s]]
    heapq.heapify(q)
    while q:
        w,n = heapq.heappop(q)
        search(n,w,q,weight)

    return weight


# ワーシャルフロイド
# 全点間 最短距離
N,M = inpl()
cost = [[INF for i in range(N)] for j in range(N)]

for _ in range(M):
    a,b,c = inpl()
    a,b = a-1,b-1
    cost[a][b],cost[b][a] = c

for k in range(N):
    for i in range(N):
        for j in range(N):
            cost[i][j]=min(cost[i][j],cost[i][k]+cost[k][j])

```

```python
# Ford_Fulkerson
# 最小カット最大フローやつ
N,E = inpl()
Start = 0
Goal = N-1
ans = 0

lines = defaultdict(set)
cost = [[0]*N for i in range(N)]
for i in range(E):
    a,b,c = inpl()
    if c != 0:
        lines[a].add(b)
        cost[a][b] += c

def Ford_Fulkerson(s): #sからFord-Fulkerson
    global lines
    global cost
    global ans

    queue = deque()     #BFS用のdeque
    queue.append([s,INF])
    ed = [True]*N    #到達済み
    ed[s] = False
    route = [0 for i in range(N)]   #ルート
    route[s] = -1

    #BFS
    while queue:
        s,flow = queue.pop()
        for t in lines[s]:  #s->t
            if ed[t]:
                flow = min(cost[s][t],flow)  #flow = min(直前のflow,line容量)
                route[t] = s
                queue.append([t,flow])
                ed[t] = False
                if t == Goal: #ゴール到達
                    ans += flow
                    break
        else:
            continue
        break
    else:
        return False

    #ラインの更新
    t = Goal
    s = route[t]
    while s != -1:

        #s->tのコスト減少，ゼロになるなら辺を削除
        cost[s][t] -= flow
        if cost[s][t] == 0:
            lines[s].remove(t)

        #t->s(逆順)のコスト増加，元がゼロなら辺を作成
        if cost[t][s] == 0:
            lines[t].add(s)
        cost[t][s] += flow

        t = s
        s = route[t]

    return True

while True:
    if Ford_Fulkerson(Start):
        continue
    else:
        break
```

```python


# クラスカル
# 最小全域木
class UnionFind:
    # 貼る

N,M = inpl()
UF = UnionFind(N)
q = []
for _ in range(M):
    a,b,w = inpl()
    a,b = a-1,b-1
    q.append([w,a,b])
q.sort()
weight = 0
for w,a,b in q:
    if not UF.Check(a,b):
        weight += w
        UF.Unite(a,b)



################################
#          数学系
################################


# Combination
class Combination:
    def __init__(self,N):
        self.fac = [1]*(N+1)
        for i in range(1,N+1):
            self.fac[i] = (self.fac[i-1]*i)%mod
        self.invmod = [1]*(N+1)
        self.invmod[N] = pow(self.fac[N],mod-2,mod)
        for i in range(N,0,-1):
            self.invmod[i-1] = (self.invmod[i]*i)%mod

    def calc(self,n,k):#nCk
        return self.fac[n]*self.invmod[k]%mod *self.invmod[n-k] %mod


#最大公約数
def gcd(a,b):
    while b:
        a,b = b, a%b
    return a

#最小公倍数
def lcm(a,b):
    return a*b // gcd(a,b)


# なんか早い素数判定
def is_prime(x):
    if x < 2: return False # 2未満に素数はない
    if x == 2 or x == 3 or x == 5: return True # 2,3,5は素数
    if x % 2 == 0 or x % 3 == 0 or x % 5 == 0: return False # 2,3,5の倍数は合成数

    # 疑似素数で割る
    prime = 7
    step = 4
    while prime <= math.sqrt(x):
        if x % prime == 0: return False
        prime += step
        step = 6 - step

    return True


```