```

###### いつもの ######

from collections import defaultdict,deque
import sys,heapq,bisect,math,itertools,string,queue
sys.setrecursionlimit(10**8)
INF = float('inf')
mod = 10**9+7
eps = 10**-7
def inpl(): return list(map(int, input().split()))
def inpl_str(): return list(input().split())




##### bit 関係 ######

#bit shift
a << b, a >> b
> a = 101, b=2
> a<<b = 10100
> a<<b =     1

#bitwiseOR AND XOR
a|b , a&b , a^b
> a = 10010, b=00110
> a|b = 10110
> a&b = 00010
> a^b = 10100

#bit長
b = 10101
> b.bit_length() = 5

#一番最初のbitが立ってるものをとる
b&-b
> b    = 10110
> b&-b =    10

#ビットマスク(特定の桁だけ)
a = 10100
> (a>>0) & 1 = 0
> (a>>1) & 1 = 0
> (a>>2) & 1 = 1
> (a>>3) & 1 = 0
> (a>>4) & 1 = 1

#next_combination (n桁でk箇所bitが立ってる物を全探索)
def next_com(bit):
    x = bit & -bit
    y = bit + x
    return (((bit & ~y) // x) >> 1) | y
n,k = 5,3
bit = (1<<k)-1
ans = 0
while bit < (1<<n):
    for i in range(n):
        if (bit>>i) & 1:
            # 処理
    bit = next_com(bit)


#bitが立ってる数をカウント
def bitcount(bits):
    bits = (bits & 0x55555555) + (bits >> 1 & 0x55555555)
    bits = (bits & 0x33333333) + (bits >> 2 & 0x33333333)
    bits = (bits & 0x0f0f0f0f) + (bits >> 4 & 0x0f0f0f0f)
    bits = (bits & 0x00ff00ff) + (bits >> 8 & 0x00ff00ff)
    return (bits & 0x0000ffff) + (bits >>16 & 0x0000ffff)


```

```
##### itertools #####
seq = ('a', 'b', 'c', 'd', 'e')

# 並べ方
list(itertools.permutations(seq))
>[('a', 'b', 'c', 'd', 'e'),
  ('a', 'b', 'c', 'e', 'd'),
  ('a', 'b', 'd', 'c', 'e'),
  ('a', 'b', 'd', 'e', 'c'),
           中略
  ('e', 'd', 'c', 'a', 'b'),
  ('e', 'd', 'c', 'b', 'a')]

# 何個かを選ぶ並べ方
list(itertools.permutations(seq, 3))
> [('a', 'b', 'c'),
  ('a', 'b', 'd'),
  ('a', 'b', 'e'),
  ('a', 'c', 'b'),
       中略
  ('e', 'd', 'a'),
  ('e', 'd', 'b'),
  ('e', 'd', 'c')]

# 重複を許す順列 ([True,False]でやればbit全探索ができる)
list(itertools.product(A, repeat=3))
>[('a', 'a', 'a'),
  ('a', 'a', 'b'),
  ('a', 'a', 'c'),
  ('a', 'b', 'a'),
  ('a', 'b', 'b'), …

# 組み合わせ
list(itertools.combinations(seq,5))
> [('a', 'b', 'c', 'd', 'e')]

# 何個かを選ぶ組み合わせ
list(itertools.combinations(seq,3))
[('a', 'b', 'c'),
 ('a', 'b', 'd'),
 ('a', 'b', 'e'),
 ('a', 'c', 'd'),
 ('a', 'c', 'e'), …

# 重複を許す組み合わせ
list(itertools.combinations_with_replacement(A, 3))
[('a', 'a', 'a'),
 ('a', 'a', 'b'),
 ('a', 'a', 'c'),
 ('a', 'b', 'b'),
 ('a', 'b', 'c'), …


##### 編集距離 #####
S1 = 'yafo'
S2 = 'yahoo'
def levenshtein(s1,s2):
    n, m = len(s1), len(s2)
    dp = [[0]*(m+1) for _ in range(n + 1)]

    for i in range(n+1):dp[i][0] = i
    for j in range(m+1):dp[0][j] = j

    for i in range(1, n + 1):
        for j in range(1, m + 1):
            if s1[i-1] == s2[j-1]:cost = 0
            else:cost = 1
            dp[i][j] = min(dp[i - 1][j] + 1,         # insertion
                           dp[i][j - 1] + 1,         # deletion
                           dp[i - 1][j - 1] + cost)  # replacement
    return dp[n][m]
print(levenshtein(S1,S2))

```

```
##########################################
#
#                データ構造
#
##########################################

class UnionFind:
    def __init__(self,N): # 頂点数 N
        self.table = [i for i in range(N)]    # 親 table[x] == x で根
        self.rank  = [1 for i in range(N)]    # 木の長さ
        self.size  = [1 for i in range(N)]    # 集合のサイズ

    def Find(self,x):    #xの根を返す
        if self.table[x] == x:
            return x
        else:
            self.table[x] = self.Find(self.table[x]) #親の更新
            self.size[x] = self.size[self.table[x]]
            return table[x]

    def Unite(self,x,y,w): #xとyをdiff(x,y)=W で繋げる
        w = w - self.weight(y) + self.weight(x)
        x,y = self.Find(x), self.Find(y)
        sx,sy = self.Size(x), self.Size(y)
        if x == y: return
        if self.rank[x] > self.rank[y]:
            self.table[y] = x
            self.size[x] = sx + sy
        else:
            self.table[x] = y
            self.size[y] = sx + sy
            if self.rank[x] == self.rank[y]:
                self.rank[y] += 1

    def Check(self,x,y):
        return self.Find(x) == self.Find(y)

    def Size(self,x):
        return self.size[self.Find(x)]



class BinaryIndexedTree():
    def __init__(self,N):
        self.N = N
        self.bit = [0]*(self.N+1)

    def add(self,a,w):
        x = a
        while x <= self.N:
            self.bit[x] += w
            x += x & -x

    def sum(self,a):
        tmp = 0
        x = a
        while x > 0:
            tmp += self.bit[x]
            x -= x & -x
        return tmp



class SegmentTree:
    def __init__(self,N,d):
        self.NN = 1
        while self.NN < N:
            self.NN *= 2
        self.SegTree = [d]*(self.NN*2-1)

    def update(self,i,x): #iの値をxに更新
        i += self.NN - 1
```

```
        self.SegTree[i] = x
        while i>0:
            i = (i-1)//2
            self.SegTree[i] = self.process(self.SegTree[i*2+1],self.SegTree[i*2+2])

    def query(self,a,b,k=0,l=0,r=None): #[A,B)の値，呼ぶときはquery(a,b)
        if r == None: r = self.NN
        if r <= a or b <= l: #完全に含まない
            return INF
        elif a <= l and r <= b : #完全に含む
            return self.SegTree[k]
        else: #交差する
            vl = self.query(a,b,k*2+1,l,(l+r)//2)
            vr = self.query(a,b,k*2+2,(l+r)//2,r)
            return(self.process(vl,vr))

    def process(self,x,y): #x,yが子の時，親に返る値
        return min(x,y)



class PotentialUnionFind:
    def __init__(self,N): # 頂点数 N
        self.table = [i for i in range(N)]    # 親 table[x] == x で根
        self.rank  = [1 for i in range(N)]    # 木の長さ
        self.size  = [1 for i in range(N)]    # 集合のサイズ
        self.diffweight = [0 for i in range(N)]

    def Find(self,x):    #xの根を返す
        if self.table[x] == x:
            return x
        else:
            root = self.Find(self.table[x]) #親の更新
            self.size[x] = self.size[self.table[x]]
            self.diffweight[x] += self.diffweight[self.table[x]]
            self.table[x] = root
            return root

    def Unite(self,x,y,w): #xとyをDiff(x,y)=W で繋げる
        w = w - self.Weight(y) + self.Weight(x)
        x,y = self.Find(x), self.Find(y)
        sx,sy = self.Size(x), self.Size(y)
        if x == y: return
        if self.rank[x] > self.rank[y]:
            self.table[y] = x
            self.size[x] = sx + sy
            self.diffweight[y] = w
        else:
            self.table[x] = y
            self.size[y] = sx + sy
            self.diffweight[x] = -w
            if self.rank[x] == self.rank[y]:
                self.rank[y] += 1

    def Check(self,x,y):
        return self.Find(x) == self.Find(y)

    def Size(self,x):
        return self.size[self.Find(x)]

    def Weight(self,x): # 重さ(根からの距離)
        self.Find(x)
        return self.diffweight[x]

    def Diff(self,x,y): # 繋がってる二点間距離
        return self.Weight(y) - self.Weight(x)

```

```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


```