```python
import sys
import math
import numpy as np

class factory: 
    def __init__(self, id):
        #nodes about 
        self.id = id
        self.con = {}
        self.dist = {}
        self.avgd = -1
        self.troops = {1: [], -1: []}

    def add_connection(self, id, node, dist):
        #connections between nodes
        self.con[id] = node
        self.dist[id] = dist
        
    def update_data(self, player, population, production):
        #reset node info
        self.player = player
        self.pop = population
        self.prod = production
        
    def new_turn(self):
        #reset vars
        self.avgd = -1
        self.troops = {1: [], -1: []}
    
    def track_troops(self, player, size, turns):
        #track any troops heading for this one
        self.troops[player].append([size, turns]) 
        
        
    def calculate_survival(self):
        self.status = []
        #itterate over all turns we know about (max of all movements to node)
        turns = max(([t[1] for t in self.troops[1]] + [t[1] for t in self.troops[-1]]) or [0])+1
        if(self.player == 1): #our territory
            #our current population
            self.future_pop = self.pop
            for turn in range(turns):
                #subtract the enemy and add our support
                self.future_pop -= sum([a for a,b in self.troops[-1] if b == turn])
                self.future_pop += sum([a for a,b in self.troops[1] if b == turn])
                #We can't survive the coming attack yet at any point
                if(self.future_pop <= 0): 
                    self.status = [-1, self.future_pop, turn]
            #We will survive 
            if(self.future_pop > 0):
                self.status = [1, self.future_pop, turns]

        elif(self.player == -1): #enemy territory
            #enemy current population
            self.future_pop = -self.pop
            #subtract the enemy and add our support
            self.future_pop -= sum([a for a,b in self.troops[-1]])
            self.future_pop += sum([a for a,b in self.troops[1]])

            #we want to know the overal victor
            if(self.future_pop <= 0): #if we need more to win overall 
                self.status = [-1, -1*self.future_pop, turns] 
            else: #we won the node
                self.status = [1, self.future_pop, turns]
        else: #netural node
            #barrier to entry
```

```
            self.future_pop = self.pop
            #has the barrier to entry been overcome?
            turnover = False
            for turn in range(turns):
                net_attack = sum([a for a,b in self.troops[1] if b == turn]) \
                           - sum([a for a,b in self.troops[-1] if b == turn])
                if(not turnover):
                    self.future_pop -= net_attack
                    #if node was taken over
                    if(self.future_pop < 0):
                        turnover = True
                        #if we won, make the pop +
                        if(net_attack > 0):
                            self.future_pop *= -1
                else:
                    self.future_pop += net_attack
            #we want to know the overal victor
            if(turnover): #territory is claimed
                if(self.future_pop < 0): #enemy claims it
                    self.status = [-1, -self.future_pop, turns]
                else: # we claim it
                    self.status = [1, self.future_pop, turns]
            else: #if we need more to win overall
                self.status = [0, self.future_pop, turns]

    #vulnerable enemy node unweighted selection
    def venus(self):

        #population density
        #self.protection = self.pop + sum([self.con[id].pop / self.dist[id] for id in self.con.keys() if self.con[id].player == self.player])
        #self.exposure = sum([self.con[id].pop / self.dist[id] for id in self.con.keys() if self.con[id].player == -1*self.player])

        #if vulnerable enemy, this will be negative
        #if vulnerable friend, this will be negative
        #if secure enemy, this will be positive
        #if secure friend, this will be positive
        #self.rank = self.protection - self.exposure

        #connections to enemy nodes
        self.exposure = [n for n in self.con.keys() if self.con[n].player == -1]
        #connection to friendly nodes
        self.protection = [n for n in self.con.keys() if self.con[n].player == 1]

        #rank = number of connections to node
        if(len(self.exposure) > len(self.protection)):
            self.RANK = 0 #vulnerable
        elif (len(self.exposure) < len(self.protection)):
            self.RANK = 2 #secure
        else:
            self.RANK = 1 #neutral

        #enemy pop in case of blitz
        self.threat = sum(self.con[n].pop for n in self.exposure)
        #friendly pop in case of blitz
        self.defense = sum(self.con[n].pop for n in self.protection)
        #net population after attack
        self.net_pop = self.defense - self.threat
        #depending on the player update the net population
        if(self.player == 1):
            self.net_pop += self.pop
        else:
            self.net_pop -= self.pop

        #set blitz
        if(self.net_pop < 0):
```

```python
            self.CLASS = 0 #vulnerable
        elif (self.net_pop > 0):
            self.CLASS = 2 #secure
        else:
            self.CLASS = 1 #neutral

        #set rank of engagement (0-8)
        self.ROE = 3*self.RANK + self.CLASS
#---------------------------------------------ROUTING----------------------------------------------
def compute_aim():
    global moves, bombs, wait
    #bombs
    total_cyborgs = sum([n.pop for n in nodes]) #This only takes in acount those in facotries
    factory_rank = sorted([n for n in nodes if n.player != 1], key=lambda n: n.pop / total_cyborgs, reverse=True)

    if((factory_rank[0].pop > total_cyborgs * .33) and (bombs > 0) and (wait == 0)):
        #find closest node to our target
        closest_node = sorted(factory_rank[0].dist, key=lambda e: factory_rank[0].dist[e])
        closest_node = [n for n in closest_node if nodes[n].player == 1]
        if(closest_node):
            wait = factory_rank[0].dist[closest_node[0]]
            moves += 'BOMB %s %s;' % (closest_node[0], factory_rank[0].id)
            bombs -= 1
    else:
        if(wait > 0):
            wait -=1

    #Get proximity values, the others we have
    average_distance = {}
    for n in nodes:
        distances = [d for k,d in n.dist.items() if nodes[k].player == 1]
        if(distances):
            n.avgd = sum(distances)/len(distances)
        else:
            n.avgd = -1
        n.calculate_survival()
        n.venus()

        #if we control more than half of the board
        if(sum([n.pop for n in nodes if n.player == 1]) >= total_cyborgs*.5):
            #if we will maintain control over this node and we have a pop 10 and 15
            if((n.status[0] == 1) and (n.player == 1) and (abs(n.pop - 12.5) < 2.5)):
                moves += 'INC %s;' % n.id
                n.pop = math.floor(n.pop/2)

    #get only nodes that are in danger or not ours
    aim_order = [n for n in nodes if n.player == 0]

    aim_order += [n for n in nodes if (n.status[0] != 1) and n not in aim_order]

    #choose most vulnerable nodes
    aim_order.sort(key = lambda n: n.ROE)

    #sort by how many turns I have to respond
    #aim_order.sort(key = lambda n: n.status[2])

    #if two nodes have the same pop and prox, chose higher prod
    #aim_order.sort(key = lambda n: n.prod, reverse=True)

    #first by
    aim_order = [n for n in aim_order if n.avgd > 0]
    aim_order.sort(key = lambda n: n.avgd)

```

```
    print([[n.id, n.ROE, n.pop, n.prod] for n in aim_order], file=sys.stderr) 
    
    return aim_order

target_prod = lambda node,turns: (node.prod*turns)+2 if (node.player==-1) else 2 if (node.player==0) else 0

def dijkstra_routing(src, target_pop):
    #1 more than the farthest possible solution 
    max_distance = 301 
    #list to track if nodes have been visited (that we own)
    unvisited = [n for n in nodes if n.player == 1]
    #Dijkstra's shortest path
    dsp = {src: [0, None, 0, 0]}
    
    #if we have no nodes, we lost
    if(not unvisited):
        return dsp

    if(len(unvisited) == 1):
        #is there a direct connection between the one node and our goal?
        if(src.id in unvisited[0].dist):
            #get the target factory production
            pop_inc = target_prod(src, unvisited[0].dist[src.id])
            #do we have enough to over come it (and its inc in pop)
            if(unvisited[0].pop >= target_pop + pop_inc):
                dsp[unvisited[0]] = [unvisited[0].dist[src.id], src, target_pop, target_pop + pop_inc]
            else:
                dsp[unvisited[0]] = [unvisited[0].dist[src.id], src, unvisited[0].pop-2, unvisited[0].pop-2]
        else:
            #no solution, return
            return dsp
        return dsp
        
    #add the source node as a starting point
    unvisited.insert(0, src)

    for node in unvisited:
        if(node != src):
            #initialize with no distance, no previous vertex, no cum. population, and no used pop
            dsp[node] = [max_distance, None, 0, 0] 
    
    #while there are still nodes to visit
    while(unvisited): 
        #sort by distance and get the first element 
        current_node = sorted(unvisited, key = lambda e: dsp[e][0])[0]
        #loop over each connection 
        for id, connected_node in [n for n in current_node.con.items() if n[1].player == 1]:
            #calculate the shortest new shortest path to the connected node
            node_distance = dsp[current_node][0] + current_node.dist[id]
            #get the target factory production
            pop_inc = target_prod(src, node_distance)
            #if it is shorter than what is listed or the distance is None
            if(node_distance < dsp[connected_node][0]):
                #see if this node can contribute
                if((dsp[current_node][2] >= target_pop + pop_inc) or (connected_node.pop <= 2)):
                    #we don't need anything from this connection
                    dsp[connected_node] = [node_distance, current_node, dsp[current_node][2], 0]
                else:
                    #calculate how many from this node will be needed (leave 2 for production)
```

```
                    if((target_pop + pop_inc) - dsp[current_node][2] > connected_node.pop-2):
                        #we need everything we can get from this node
                        dsp[connected_node] = [node_distance, current_node, dsp[current_node][2]+connected_node.pop-2, connected_node.pop-2]
                    else:
                        #we need a portion but not all
                        dsp[connected_node] = [node_distance, current_node, target_pop, (target_pop + pop_inc) - dsp[current_node][2]]
        #we have visited this node, remove it from the unvisited array
        unvisited.remove(current_node)
    return dsp

def route_move(target):
    #output variable
    move = ''
    #get the shortest path map
    dsp = dijkstra_routing(target, target.pop)
    #isolate paths that cumulatively send enough troops and have something to send
    paths = [[k,v] for k,v in dsp.items() if (v[2] >= target.pop) and (v[3] > 0)]
    #if we have a potential route
    
    if(paths):
        #sort paths by distance
        paths.sort(key = lambda e: e[1][0])
        #begin the path with the shortest distance
        start = paths[0]
        #while we have a parent node
        while(start[1][1] != None):
            #update move commands
            move += 'MOVE %s %s %s;' % (start[0].id, start[1][1].id, start[1][3])
            #update the population of the sending node so we don't oversend in the future
            start[0].pop -= start[1][3]
            #update the start node
            start = [start[1][1], dsp[start[1][1]]]
    return move

#--------------------------------------------INIT--------------------------------------------
#bomb tracking
bombs = 2
wait = 0 #turns for bomb to detonate
#move tacking
moves = ''
#make array of factory nodes
node_count = int(input()) # the number of nodes
nodes = []
for id in range(node_count): 
    nodes.append(factory(id))

#make a grid of bidirectional connections
for i in range(int(input())): # the number of links between nodes
    node_1, node_2, dist = [int(j) for j in input().split()]
    nodes[node_1].add_connection(node_2, nodes[node_2], dist)
    nodes[node_2].add_connection(node_1, nodes[node_1], dist)

# game loop
while True:
    #reset essential variables for the new round
    for id in range(node_count): 
        nodes[id].new_turn()

    # the number of entities (e.g. factories and troops)
    entity_count = int(input())  
    for i in range(entity_count):
        entity_id, entity_type, arg_1, arg_2, arg_3, arg_4, arg_5 = input().split()
        #update information for each turn
```

```
        if(entity_type == 'FACTORY'): #factory
            nodes[int(entity_id)].update_data(int(arg_1), int(arg_2), int(arg_3))
        elif(entity_type == 'TROOP'): #troop
            nodes[int(arg_3)].track_troops(int(arg_1), int(arg_4), int(arg_5))
    
    moves = ''
    #get all actions
    targets = compute_aim()
    for n in targets:
        #accomplish as many as possible
        moves += route_move(n)
    
    #if we can make a move, do so, otherwise wait
    if(moves != ''):
        print(moves[:-1])
    else:
        print("WAIT")
```