A. 璃月的桥梁

令山峰为点，桥梁为边，这是一个连通图，这道题询问是否存在从传送锚点出发的欧拉通路。

解法一：检查每个点的度数

我们知道连通图存在欧拉通路的充分必要条件是所有点的度都为偶数，此时可以从任意点出发，或者有且只有两个点的度数为奇数，这样必须从两个点之一出发。

我们可以对两种情况分类讨论，或者这两种情况可以概括为：度为奇数的非起点的点有至多1个。（如果有，那这个点就是终点，不然终点和起点重合）

标准程序：

```python
n, m = map(int, input().split(' '))

degree = [0 for _ in range(n + 1)]

for _ in range(m):
    u, v = map(int, input().split(' '))
    degree[u] += 1
    degree[v] += 1

print('YES' if sum(i % 2 for i in degree[2:]) <= 1 else 'NO')
```

B. 枫丹的预言

这道题非常的水，基本横着倒着都能对，题目讲的什么不重要，关于枫丹的预言可以进游戏里面看看，这里就不多说了。总之题目要求我们求解$f(t)=k$，换句话说求连续可导$F(t)=f(t)-k$的零点。

解法一：二分法

也就是每次取中间，然后判断是在左边还是右边，然后不断缩小范围，直到找到答案。

标准程序：

```
k = float(input())

def f(x):
    print("? %.10f" % x)
    y = float(input())
    return y - k

def biselect(l, r, eps):
    while r - l >= eps:
        m = (r + l) / 2
        if f(m) >= 0:
            r = m
        else:
            l = m
    return (r + l) / 2

ans = biselect(0, 100, 1e-6)

print("! %.10f" % ans)
```

解法二：弦截法

也就是用最近两个点的连线来斜率来代替牛顿迭代中的导数，然后求出零点。

标准程序：

```
k = float(input())

def f(x):
    print("? %.10f" % x)
    y = float(input())
    return y - k

def secant(g1, g2, backwardeps):
    lbound, rbound = g1, g2
    be1, be2 = f(g1), f(g2)
    while abs(be2) >= backwardeps:
        l = 1
        g3 = g2 - be2 * (g2 - g1) / (be2 - be1)
        while g3 < lbound or g3 > rbound:
            l /= 2
            g3 = g2 - l * be2 * (g2 - g1) / (be2 - be1)
        g1, g2 = g2, g3
        be1, be2 = be2, f(g2)
        if be2 > 0:
            rbound = g2
        else:
            lbound = g2
    return g2
```

```
24
25 ans = secant(0, 100, 1e-6)
26
27 print("! %.10f" % ans)
```

C. 稻妻的解谜

这道题要求你对整个图进行分层标号，并保证每一层与相邻两层完全连接。

这里可以参考上课讲的判定完全二部图做法。

判定一个简单连通图是完全二部图是可以任选一个点出发，先将它染成红色，然后将与它相邻的点染成蓝色，接着类似地搜索下去，出现矛盾就返回不是完全二部图，最后检查两部分连接数是否恰好为两种颜色点的个数的乘积。

回到这题，检查一个解是否合法是比较容易的，就是遍历每个点检查它和前后两层每一个点都连接，且不与不相邻的层连接。

我们类比二部图发现节点的层数等于它到1号节点的距离，这点可以通过反证法证明。

这题不一样的是，它有多个层，并且是相邻的层完全连接，但基本思路是一样的，染色要换成标数，和1 相邻的标2，和2相邻的非1标3，以此类推。

解法一：广度优先搜索

广度优先搜索是确定不带权图最短路常见做法，当然在本题也可以使用。

```python
import collections

n, m = map(int, input().split(' '))
edges = [[] for _ in range(n + 1)] # layer
number = [None for _ in range(n + 1)] # number of nodes in each layer
for _ in range(m):
    u, v = map(int, input().split(' '))
    edges[u].append(v)
    edges[v].append(u)

stat = [0 for _ in range(n + 2)]
number[1] = 1
stat[1] = 1
qu = collections.deque([1])
while qu:
    u = qu.popleft()
    for v in edges[u]:
        if number[v] is None:
            number[v] = number[u] + 1
            stat[number[v]] += 1
            qu.append(v)
        elif abs(number[v] - number[u]) != 1:
            print('NO')
            exit()
for i in range(2, n):
    if stat[number[i] + 1] + stat[number[i] - 1] != len(edges[i]): # not fully connected
        print('NO')
        exit()
print('YES')
print(*number[2:])
```

解法二：一层层遍历

不过仔细想想，实际上是合法的图从每一层任意一个节点都可以遍历到下一层的所有节点，所以只需要在每一层任选一个节点就可以了。

根本没必要维护什么队列。

出于方便考虑，程序编写时选择了每轮遍历的最后一个节点作为本层选择的节点。

```python
n, m = map(int, input().split(' '))
```

```
edges = [[] for _ in range(n + 1)]
number = [0 for _ in range(n + 1)]
for _ in range(m):
    u, v = map(int, input().split(' '))
    edges[u].append(v)
    edges[v].append(u)

number[1] = 1
p = 1 # vertex selected in the previous layer
d = 2 # the layer being iterated
while p:
    e = edges[p]; p = None
    for v in e:
        if number[v] == 0:
            number[v] = d
            p = v
    d += 1
stat = [0 for _ in range(n + 2)]
for i in range(1, n + 1):
    stat[number[i]] += 1
for i in range(1, n + 1):
    if number[i] == 0 or stat[number[i] - 1] + stat[number[i] + 1] != len(edges[i]) or \
            not all(abs(number[i] - number[j]) == 1 for j in edges[i]):
        print('NO')
        exit()
print('YES')
print(*number[2:])
```

D. 蒙德的卡牌

题目有2种消耗1骰子的技能，1种消耗2骰子的技能，问你消耗k骰子有多少种方法，对998244353取模。

考虑$f(k)$表示消耗k骰子的方法数，那么

$$f(0)=1$$
$$f(1)=2$$
$$f(k)=2f(k-1)+f(k-2)$$

也就是花完k个骰子需要先花完$k-1$个骰子，然后再用1个骰子，或者先花完$k-2$个骰子，然后再用2个骰子。

解法一：矩阵

递推关系可以改写为：

$$\begin{pmatrix} f(k) \\ f(k-1) \end{pmatrix} = \begin{pmatrix} 2 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} f(k-1) \\ f(k-2) \end{pmatrix}$$

于是通项

$$\begin{pmatrix} f(k) \\ f(k-1) \end{pmatrix} = \begin{pmatrix} 2 & 1 \\ 1 & 0 \end{pmatrix}^{k-1} \begin{pmatrix} f(1) \\ f(0) \end{pmatrix}$$

关于怎么求一个东西的n次方，题目末尾给了。因为都是乘法，所以可以直接取模，计算中给矩阵里每个元素都取模。

这个矩阵实际上比较小，直接计算没啥问题，不必对角化。下面两种解法实际上相比这种做法都是常数优化。

标准程序：

```python
MOD = 998244353

def matrix_mul(A, B):
    n = len(A)
    C = [[0] * n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            for k in range(n):
                C[i][j] += A[i][k] * B[k][j]
                C[i][j] %= MOD
    return C

def matrix_pow(A, k):
    n = len(A)
    res = [[0] * n for _ in range(n)]
    for i in range(n):
        res[i][i] = 1
    while k > 0:
        if k % 2 == 1:
            res = matrix_mul(res, A)
        A = matrix_mul(A, A)
        k //= 2
    return res

k = int(input())
t = matrix_pow([
    [2, 1],
    [1, 0]
], k - 1)
print((2 * t[0][0] + t[0][1]) % MOD)
```

解法二：扩展数域

考虑特征根法解线性常系数递推关系，特征方程$\lambda^2-2\lambda-1=0$（实际上就是上一种解法的矩阵的本征值，如果你做对角化的话也会来到这个地方）。

解出$\lambda_1=1+\sqrt{2}$，$\lambda_2=1-\sqrt{2}$。

注意到$\sqrt{2}$实际上在有理数内开方是开不出来的，在实数范围内可以开但是模运算对实数无能为力。我们暂且先忽略这个问题，等会回来解决。

我们现在知道通解是

$$f(k)=b(1+\sqrt{2})^k+d(1-\sqrt{2})^k$$

代入前两项确定参数b和d:

$$f(0)=b+d=1$$
$$f(1)=b(1+\sqrt{2})+d(1-\sqrt{2})$$

解得$b=\dfrac{2+\sqrt{2}}{4}$，$d=\dfrac{2-\sqrt{2}}{4}$，于是通项:

$$f(k)=\frac{2+\sqrt{2}}{4}(1+\sqrt{2})^k+\frac{2-\sqrt{2}}{4}(1-\sqrt{2})^k$$

现在回到$\sqrt{2}$开不出来的问题，我们都学过复数，我们尝试构造一个数域。

复数是为了解决$x^2=-1$在实数范围内没有根的问题，令$i^2=-1$，构造出数域$\{a+bi|a,b\in\mathbb{R}\}$。

依据上述思想，我们可以构造出数域$\{a+b\sqrt{2}|a,b\in\mathbb{Q}\}$，然后把模运算扩展到这个数域上，定义$(a+b\sqrt{2})\bmod p=(a\bmod p)+(b\bmod p)\sqrt{2}$。

如此扩展能够保证四则运算的封闭性，加法减法乘法和模运算的分配律，证明略。

同时仿照复数的Re记号，我们记$R(a+b\sqrt{2})=a$。

注意到通项公式的两部分有理部分相等而$\sqrt{2}$部分互为相反数，实际上我们算一半就可以了，然后拿有理数部分除以2。

也就是

$$f(k)=\frac{R((2+\sqrt{2})(1+\sqrt{2})^k)}{2}$$

每一步都模的话，实际上不能保证最后结果一定能被2整除，但是这个问题很好解决，如果整除不了加上998244353再除就好了，另一种做法是将模数乘以2，但 后者必须很小心溢出。

标准程序：

```python
MOD = 998244353 * 2 # Multiply by 2 so that we can always divide by 2 at the end

def ir_mul(a, b): # Multiply two irrationals in form a + b sqrt(2)
    return ((a[0] * b[0] + 2 * a[1] * b[1]) % MOD, (a[0] * b[1] + a[1] * b[0]) % MOD)

def ir_pow(a, n):
    ret = (1, 0)
    while n:
        if n & 1:
            ret = ir_mul(ret, a)
        a = ir_mul(a, a)
        n >>= 1
    return ret

print(ir_mul(ir_pow((1, 1), int(input())), (2, 1))[0] // 2)
```

解法三：模数

超纲解法，这里只解释基本思路，不追求严谨。

回到上面$\sqrt{2}$开不出来的地方，这次我们走另一条路，因为反正后面要取模，考虑模数里面开，换句话说，我们想要找到：

$$x^2 \equiv 2 \pmod{998244353}$$

的根，比较好的事情是这玩意是有根的，写个程序把1到$998244353-1$挨个试一遍得到两个根之一$x_1 = 116195171$。

除以4也可以通过类似方法解决，求

$$4x \equiv 1 \pmod{998244353}$$

解得$x = 748683265$。

于是代入表达式

$$f(k) \equiv 748683265((2+116195171)(1+116195171)^k + (2-116195171)(1-116195171)^k) \pmod{998244353}$$

好了可以计算了。

需要注意的是，这种解法需要出题人的数比较好，如果上述方程没有解就比较寄。

标准程序：

```python
MOD = 998244353; SQRT2 = 116195171; INV4 = 748683265

def mod_pow(x, n):
    ret = 1
    while n > 0:
        if n & 1:
            ret = ret * x % MOD
        x = x * x % MOD
        n >>= 1
    return ret

k = int(input())
print(((2 + SQRT2) * mod_pow(1 + SQRT2, k) + (2 - SQRT2) * mod_pow(1 - SQRT2, k)) * INV4 % MOD)
```

E. 须弥的山头

如题，这题的题目总结得很到位，给出一个有红蓝两种边的无向带权图，求至多经过一条蓝边的从起点到终点的最短路。或者断言无论如何都无法到达终点。

注意到没有负边权单源最短路，优先考虑Dijkstra算法而不是Bellman-Ford。

开始之前让我们先复习一下Dijkstra， 状态转移方程：

$$dp(u)=0,u\text{是起点}$$
$$dp(u)=\min_{v,w}\{dp(v)+w\},u\text{不是起点}$$

可以使用贪心思想求解，每次选择未确定距离的点中当前最小距离的点确定下来，然后松弛这个点有关的边，因为这个过程中处理的每个点上的距离是单调不减的，又没有负边权，所以肯定不会松弛到已经确定的位置。

我们这里使用一个符号∞表示实数集的上确界。

算法如下：

$DIJKSTRA(G,start):$
    $Q=\varnothing$ //没有确定长度的点
    $S=\varnothing$ //从已知点过来的最短长度
    for v **in** $G.vertices:$
        $S(v)=\infty$
        $INSERT(Q,v,\infty)$
    $S(start)=0$ //设置起点为0
    $UPDATE(Q,start,0)$
    while $Q\neq\varnothing:$
        $u,d=EXTRACT\_MIN\_VALUE(Q)$ //取出当前未确定的点中距离最小的
        for v,w **in** $u.edges:$
            if $S(v)>d+w:$ //松弛
                $S(v)=d+w$
                $UPDATE(Q,v,d+w)$
    return S

令G是一个n个点m条边的图，假如$EXTRACT_MIN_VALUE()$，$INSERT()$和$UPDATE()$时间复杂度都是$O(\lg n)$，那么Dijkstra算法时间复杂度为$O((n+m)\lg n)$。

（这是稀疏图的做法，如果图比较稠密，即$m\approx n(n-1)/2$，我们选择$EXTRACT_MIN_VALUE()$是$O(n)$，$UPDATE()$是$O(1)$，这样总时间复杂度是$O(n^2+m)$，符合这个条件的数据结构有数组、链表等。）

实践中常常向各大编程语言标准库中不支持$UPDATE()$操作的优先队列妥协，用$INSERT()$代替$UPDATE()$，取出使时再检查是否被覆盖过，虽然 这样Q里会有很多重复元素，但实际上一般不会慢很多，多余元素肯定不超过$m-n$个，时间复杂度$O((n+m)\lg m)$。但事实上$m\leq\dfrac{n(n-1)}{2}$，这两个运行时间在渐进意义上没什么区别，是相等的，加了点常数。

$$
\begin{aligned}
&DIJKSTRA(G, start):\\
&\quad Q = \varnothing\\
&\quad S = \varnothing\\
&\quad \textbf{for } v \textbf{ in } G.vertices:\\
&\quad\quad S(v) = \infty\\
&\quad S(start) = 0\\
&\quad INSERT(Q, start, 0)\\
&\quad \textbf{while } Q \neq \varnothing:\\
&\quad\quad u, d = EXTRACT\_MIN\_VALUE(Q)\\
&\quad\quad \textbf{if } S(u) \neq d: \quad //\text{之后被覆盖过，就跳过此次处理}\\
&\quad\quad\quad \textbf{continue}\\
&\quad\quad \textbf{for } v, w \textbf{ in } u.edges:\\
&\quad\quad\quad \textbf{if } S(v) > d + w:\\
&\quad\quad\quad\quad S(v) = d + w\\
&\quad\quad\quad\quad INSERT(Q, v, d + w)\\
&\quad \textbf{return } S
\end{aligned}
$$

有人可能会说，咱可没学过堆，你这题超纲了。

Dijkstra算法压根没说必须要用堆，老师的PPT里也压根没出现堆这个词。

不知道你们有没有听说过这样一句话，叫：

没万叶可以用砂糖

堆没学过我们可以找下位替代，在我们学过的数据结构里找到一个，它满足更新操作$UPDATE()$或者插入$INSERT()$操作时间复杂度是$O(\lg n)$，且取出最小值操作$EXTRACT_MIN_VALUE()$时间复杂度也是$O(\lg n)$。

显然就有一个AVL树，它满足这个条件，可以用AVL树作为Q的实现，于是Dijkstra算法得以实现，时间复杂度$O((n+m)\lg n)$。

实践中采用堆而不是AVL树、红黑树、线段树的原因是堆本身支持的操作比较少，维护容易，运行时间常数小，不同于后三者支持一大堆操作，但相应的维护也麻烦。

如果你担心因为使用AVL树而运行超时，可以采用C/C++/Java而不是Python作答。为了方便，标准程序使用编程语言标准库中的优先队列，我们更关注Dijkstra算法本身，而不是一些细枝末节。（实际上，即使你真的用PyPy3交AVL也不会TLE）

复习完了Dijkstra，我们回到这题。

解法一：从两侧求最短路

因为至多经过一条蓝边，所以我们可以枚举经过的蓝边，计算蓝边连接的两个点分别到起点和终点的最短路，然后取最小，再和不经过蓝边最短路的取最小，记$d(a,b)$为从a到b的最短路，也就是

$$
ans = \min\{\min_{(u,v,w)\in blue\_edges}\{d(1,u)+w+d(v,\ 2),\ d(1,v)+w+d(u,2)\},\ d(1,2)\}
$$

标准程序：

```python
import queue

BIG = 1_000_000_000_000_000

n, m, k = map(int, input().split())
red_edges = [[] for _ in range(n + 3)]
blue_edges = []

for _ in range(m):
    u, v, w = map(int, input().split())
    red_edges[u].append((v, w))
    red_edges[v].append((u, w))

```

```
for _ in range(k):
    u, v, w = map(int, input().split())
    blue_edges.append((u, v, w))

def dijkstra(dp, start):
    qu = queue.PriorityQueue()
    qu.put((0, start))
    dp[start] = 0
    while not qu.empty():
        d, u = qu.get()
        if dp[u] != d:
            continue
        for v, w in red_edges[u]:
            if dp[v] > dp[u] + w:
                dp[v] = dp[u] + w
                qu.put((dp[v], v))

dp1 = [BIG] * (n + 3)
dp2 = [BIG] * (n + 3)
dijkstra(dp1, 1)
dijkstra(dp2, 2)

ans = dp1[2]
for u, v, w in blue_edges:
    ans = min(ans, dp1[u] + dp2[v] + w, dp1[v] + dp2[u] + w)
print(ans if ans < BIG else -1)
```

解法二：改成二维

我们动手改改方程，使得它的状态记录经过了多少条蓝边。

令$dp(u,i)$表示从起点经过恰好i条蓝边到达u的最短路，或者∞表示不存在这样的路。

$$dp(u,i)=\infty, i<0$$

$$dp(u,i)=0, u\text{是起点且}i=0$$

$$dp(u,i)=\min\{\min_{(v,w)\in u.red\_edges}\{dp(v,i)+w\}\}, \min_{(v,w)\in u.blue\_edges}\{dp(v,i-1)+w\}\}, u\text{不是起点或}i\geq 0$$

相应修改算法，事实上，Dijkstra算法是不太常见的推送型动态规划，也就是前面的项计算时会将自己的影响推送到后面的项上，而不是后面的项计算的时候从前面的项拉取数值，这是很多人看到这个算法觉得很陌生的原因。

$DIJKSTRA\_2D(G, start, max\_blue):$ //max_blue:最大允许经过的蓝边数
    $Q = \varnothing$
    $S = \varnothing$
    for v **in** $G.vertices:$
        for $i = 0$ **to** $max\_blue:$
            $S(v, i) = \infty$
            $INSERT(Q, (v, i), \infty)$
    $S(start, 0) = 0$
    $UPDATE(Q, (start, 0), 0)$
    while $Q \neq \varnothing:$
        $(u, i), d = EXTRACT\_MIN\_VALUE(Q)$
        for v, w **in** $u.red\_edges:$ //方程前一项
            if $S(v, i) > d + w:$
                $S(v, i) = d + w$
                $UPDATE(Q, (v, i), d + w)$
        if $i == max\_blue:$
            continue //没有比它经过蓝边更多的了项了，再大就不合法了
        for v, w **in** $u.blue\_edges:$ //方程后一项
            if $S(v, i + 1) > d + w:$
                $S(v, i + 1) = d + w$
                $UPDATE(Q, (v, i + 1), d + w)$
    return S

这个算法在允许多次经过蓝边时显得比较自然，但在只允许经过一次时显得有点杀鸡用牛刀。

实际上，如果不修改Dijkstra本身的话，把每个点拆成两个也基本上属于这一种思路。

标准程序：

```python
import queue

BIG = 1_000_000_000_000_000

n, m, k = map(int, input().split())
red_edges = [[] for _ in range(n + 3)]
blue_edges = [[] for _ in range(n + 3)]

for _ in range(m):
    u, v, w = map(int, input().split())
    red_edges[u].append((v, w))
    red_edges[v].append((u, w))

for _ in range(k):
    u, v, w = map(int, input().split())
    blue_edges[u].append((v, w))
    blue_edges[v].append((u, w))

def dijkstra(dp, start):
    qu = [(0, start, 0)]
    dp[start][0] = 0
    while qu:
        d, u, i = queue.heappop(qu)
        if dp[u][i] != d:
            continue
        for v, w in red_edges[u]:
            if dp[v][i] > d + w:
                dp[v][i] = d + w
                queue.heappush(qu, (d + w, v, i))
        if i != 0:
            continue
        for v, w in blue_edges[u]:
```

```
            if dp[v][i + 1] > d + w:
                dp[v][i + 1] = d + w
                queue.heappush(qu, (d + w, v, i + 1))


dp = [[BIG, BIG] for _ in range(n + 3)]
dijkstra(dp, 1)
ans = min(dp[2][0], dp[2][1])
print(ans if ans < BIG else -1)
```

另外后面有AVL版Dijkstra程序（使用C语言编写）。

附常见的树比较，感兴趣自行了解：

名称	用途	插入	更新	区间更新	删除	查找元素	查询最值
二叉堆	为查询最值而生	$O(\lg n)$	$O(\lg n)$	$O(n)$	$O(\lg n)$	$O(n)$	$O(1)$
红黑树	二叉搜索树	$O(\lg n)$	$O(\lg n)$	$O(n)$	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$
AVL树	二叉搜索树	$O(\lg n)$	$O(\lg n)$	$O(n)$	$O(\lg n)$	$O(\lg n)$	$O(\lg n)$
线段树	为区间操作而生	不支持	$O(\lg n)$	$O(\lg n)$	不支持	$O(\lg n)$	$O(1)$

附录

E题解法一AVL版Dijkstra

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// AVL 子树
struct AVLTree {
    int key;
    long long weight;
    int height;
    struct AVLTree *child[2];
};

static struct AVLTree AVL_NIL = {0, 0, 0, {&AVL_NIL, &AVL_NIL}};

// 比较AVL两个节点
static long long avltree_compare(const struct AVLTree *lhs, const struct AVLTree *rhs) {
    return lhs->weight != rhs->weight ? lhs->weight - rhs->weight : lhs->key - rhs->key;
}

// 重新计算AVL节点的高度
static void avltree_recompute_height(struct AVLTree *node) {
    int lh = node->child[0]->height, rh = node->child[1]->height;
    node->height = 1 + (lh > rh ? lh : rh);
}

// 向dir方向旋转AVL子树
static void avltree_rotate(struct AVLTree **node, int dir) {
    struct AVLTree *tmp = (*node)->child[!dir];
    (*node)->child[!dir] = tmp->child[dir];
    tmp->child[dir] = *node;
    avltree_recompute_height(*node);
    avltree_recompute_height(tmp);
    *node = tmp;
}

// 修复AVL子树，其中可能高度较高的一侧是dir方向
static void avltree_fixup(struct AVLTree **node, int dir) {
    int ih = (*node)->child[dir]->height, oh = (*node)->child[!dir]->height;
    if (ih - oh >= 2) {
        if ((*node)->child[dir]->child[dir]->height < (*node)->child[dir]->child[!dir]->height) {
            avltree_rotate(&(*node)->child[dir], dir);
        }
        avltree_rotate(node, !dir);
    }
}

// 向AVL树中插入节点
static void avltree_insert(struct AVLTree **node, struct AVLTree *item) {
    if (*node == &AVL_NIL) {
        *node = item;
        return;
    }
    int dir = avltree_compare(item, *node) > 0;
    avltree_insert(&(*node)->child[dir], item);
    avltree_fixup(node, dir);
```

```
    avltree_recompute_height(*node);
}

// 取出AVL树中的最小节点
static struct AVLTree *avltree_extract_min(struct AVLTree **node) {
    if (*node == &AVL_NIL) {
        return &AVL_NIL;
    } else if ((*node)->child[0] != &AVL_NIL) {
        struct AVLTree *ret = avltree_extract_min(&(*node)->child[0]);
        avltree_fixup(node, 1);
        avltree_recompute_height(*node);
        return ret;
    } else if ((*node)->child[1] != &AVL_NIL) {
        struct AVLTree *ret = *node;
        *node = (*node)->child[1];
        return ret;
    } else {
        struct AVLTree *ret = *node;
        *node = &AVL_NIL;
        return ret;
    }
}

// 边
struct Edge {
    int to;           // 目标点
    long long weight; // 权重
    int next;         // 下一条边
};

// 图
struct Graph {
    int vertices_count;
    int *adj;
    struct Edge *edges;
    int edges_count;
};

// 创建图
static struct Graph graph_new(int vertices_count, int *adj, struct Edge *edges_pool) {
    struct Graph g = {vertices_count, adj, edges_pool, 0};
    memset(g.adj, -1, vertices_count * sizeof(int));
    return g;
}

// 添加边
static void graph_add_edge(struct Graph *g, int from, int to, long long weight) {
    g->edges[g->edges_count++] = (struct Edge){to, weight, g->adj[from]};
    g->adj[from] = g->edges_count - 1;
}

static const long long INF = 1e15;

// Dijkstra求最短路径
static void graph_dijkstra(const struct Graph *g, int start, long long *dp) {
    for (int i = 0; i < g->vertices_count; i++) {
        dp[i] = INF;
    }
    dp[start] = 0;
    struct AVLTree avl_pool[g->edges_count + 1], *avl_offset = avl_pool, *q = &AVL_NIL;
    *avl_offset = (struct AVLTree){start, 0, 1, {&AVL_NIL, &AVL_NIL}};
    avltree_insert(&q, avl_offset++);
```

```
    while (q != &AVL_NIL) {
        struct AVLTree *node = avltree_extract_min(&q);
        if (dp[node->key] != node->weight) {
            continue;
        }
        for (int i = g->adj[node->key]; i != -1; i = g->edges[i].next) {
            struct Edge *e = &g->edges[i];
            if (dp[e->to] > dp[node->key] + e->weight) {
                dp[e->to] = dp[node->key] + e->weight;
                *avl_offset = (struct AVLTree){e->to, dp[e->to], 1, {&AVL_NIL,
&AVL_NIL}};
                avltree_insert(&q, avl_offset++);
            }
        }
    }
}

int main(void) {
    int n, m, k;
    scanf("%d%d%d", &n, &m, &k);
    int adj_pool[n + 3];
    struct Edge edge_pool[2 * m];
    struct Graph g = graph_new(n + 3, adj_pool, edge_pool);
    for (int i = 0; i < m; i++) {
        int u, v, w;
        scanf("%d%d%d", &u, &v, &w);
        graph_add_edge(&g, u, v, w);
        graph_add_edge(&g, v, u, w);
    }
    long long dp1[n + 3], dp2[n + 3];
    graph_dijkstra(&g, 1, dp1);
    graph_dijkstra(&g, 2, dp2);
    long long ans = dp1[2];
    for (int i = 0; i < k; i++) {
        int u, v, w;
        scanf("%d%d%d", &u, &v, &w);
        long long l1 = dp1[u] + w + dp2[v], l2 = dp2[u] + w + dp1[v];
        ans = l1 > ans ? ans : l1;
        ans = l2 > ans ? ans : l2;
    }
    printf("%lld\n", ans >= INF ? -1 : ans);
    return 0;
}
```