Message route

Syrjälä hálózatán található n darbab számítógép. A feladatunk az, hogy megállapítsuk, hogy Uolevi képes-e üzenetet küldeni Maija felé és amennyiben igen, úgy mi az útvonalon található minimális számítógépek száma.

Bemenet

Az első bemeneti sorban található két szám n és m a számítógépek és kapcsolatok száma. A számítógépek 1-től n-ig vannak számozva, ahol Uolevi gépe az 1-es, Maija gépe az n-edik.

Az input ezen sorát követi m sor, amely leírja a kapcsolatokat. Ezt követően m sorban két egész számként megadjuk a számítógépek közötti kapcsolatot.

Minden kapcsolat két különböző számítógép között él, és legfeljebb egy kapcsolat létezhet bármely két számítógép között.

Kimenet

Amennyiben lehetséges üzenetet küldeni, először írjuk ki k-t: az érvényes útvonalon szereplő számítógépek minimális számát. Ezután írjuk ki egy ilyen útvonal példáját. Bármely érvényes megoldás kiírható.
Amennyiben nem található útvonal, írjuk ki az "IMPOSSIBLE" szót.

A program kódja:

Töltsük be a szükséges modulokat:

- A `deque` egy olyan adatstruktúra, amely hatékonyan tud elemeket hozzáadni vagy eltávolítani mindkét végéről, így ideális a BFS (szélességi keresés) algoritmusban használt sorhoz.
- A `defaultdict` segítségével létrehozhatunk egy olyan szótárat, amely automatikusan inicializál egy üres listát minden új kulcshoz. Ez segít a gráf gyors és egyszerű építésében.

```
from collections import deque, defaultdict
```

Hozzunk létre egy függvényt a `legrovidebb_ut_keresese()` néven. Ez lesz a program „magja”.

```
def legrovidebb_ut_keresese():
```

Olvassuk be a konzolról a számítógépek (`n`) és a kapcsolatok (`m`) számát egész számként, szóközök mentén feldarabolva. Ezzel fogjuk meghatározni a gráf méretét és hogy hány kapcsolatot kell beolvasni:

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

Létrehozunk egy `defaultdict` típusú változót `graf` néven, ahol minden kulcshoz egy üres lista lesz az alapértelmezett érték. Ez a változó tárolja a számítógépek közötti kapcsolatokat szomszédsági lista formájában, ahol minden csúcshoz a szomszédos csúcsok listáját rendeljük:

```
graf = defaultdict(list)
```

`m` sorban beolvassuk a kapcsolatokat, minden sorban két számot (`a` és `b`), majd mindkét irányban hozzáadjuk őket a `graf`-hoz. (Ez a lépés építi fel a gráfot, ahol minden csúcs szomszédjait tároljuk. Mivel nem irányított gráfról van szó, mindkét irányban tároljuk a kapcsolatokat.)

```
for _ in range(m):
    a, b = map(int, input().split())
    graf[a].append(b)
    graf[b].append(a)
```

Létrehozunk egy `sor` nevű `deque`-t, amelybe beletesszük az `1`-es csúcsot, ahol a keresést kezdjük.

```
sor = deque([1])
```

Létrehozunk egy `tavolsag` nevű szótárat, ahol az `1`-es csúcsot kezdőpontként beállítjuk 1-es távolságra (az út eleje). Ez a szótár tárolja minden csúcs minimális távolságát az 1-es csúcstól, segítve a legrövidebb út megtalálását.

```
tavolsag = {1: 1}
```

Létrehozunk egy `elozo_csucs` nevű szótárat, ahol az `1`-es csúcs elődjét `None`-ra állítja. Az `elozo_csucs` szótár segít a legrövidebb út visszakövetésében a végponttól a kezdőpontig.

```
elozo_csucs = {1: None}
```

Menjünk végig a gráf minden csúcsán. Amíg a `sor` nem üres, kiveszünk egy csúcsot a sor elejéről és eltávolítjuk azt.

```
while sor:
    csucs = sor.popleft()
```

Amennyiben elérjük az n-edik csúcsot, álljunk le, mivel elértük a cél csúcsot, így nincs szükség további keresésre:

```
if csucs == n:
    break
```

Menjünk végig a jelenlegi `csucs` minden `szomszed`-ján. Ha egy szomszédot még nem látogattunk meg, állítsuk be annak távolságát és előző csúcsát, majd adjuk hozzá a sorhoz. (így járjuk be a gráfot, és rögzítjük minden csúcs távolságát az 1-es csúcstól, valamint azt, hogy melyik csúcsról érkeztünk)

```
for szomszed in graf[csucs]:
    if szomszed not in tavolsag:
        tavolsag[szomszed] = tavolsag[csucs] + 1
        elozo_csucs[szomszed] = csucs
        sor.append(szomszed)
```

Ellenőrizzük le, hogy elértük-e az n-edik csúcsot. Ha nem, akkor írjuk ki, hogy "`IMPOSSIBLE`", mivel nincs elérhető út az 1-es csúcsból az `n`-edikbe.

```
if n not in tavolsag:
    print("IMPOSSIBLE")
    return
```

Létrehozunk egy `utvonal` listát, és a célcsúcsról visszafelé követjük az előző csúcsokat egészen az 1-es csúcsig. (Ez a legrövidebb út felépítése, de visszafelé, éppen ezért ez fordított lesz, amit majd a következő lépésben korrigálni fogunk!)

```
utvonal = []
jelenlegi_csucs = n
while jelenlegi_csucs is not None:
    utvonal.append(jelenlegi_csucs)
    jelenlegi_csucs = elozo_csucs[jelenlegi_csucs]
```

Itt fordítjuk meg:

```
utvonal.reverse()
```

Kiírjuk a legrövidebb út hosszát (`tavolsag[n]`), majd a legrövidebb út csúcsait írjuk ki egy sorban, szóközökkel elválasztva. (A `" ".join(...)` művelet az átalakított szöveges elemeket egyetlen szöveggé fűzi össze, minden elem közé egy szóközt téve.)

```
print(tavolsag[n])
print(" ".join(map(str, utvonal)))
```

A függvényünknek ez a vége.

Hívjuk meg a függvényt:

```
legrovidebb_ut_keresese()
```