A-III-4 Mafiáni u lekára

Čiastočné body

Body za ľahšie sady vstupov sa dalo získať nasledovne:

- Všetci sú jedna rodina: sú ošetrení v poradí, v ktorom prišli, stačí si pamätať, koľko ich je v čakárni a akým číslom začínajú.
- Najskôr všetci do čakárne: stačí si všetkých pacientov raz usporiadať keď doktor zavolá prvého z nich.
- Všetci majú rovnaký vplyv; nik nemá rovnaký vplyv ani rodinu: v oboch prípadoch si stačí v halde pamätať záznamy tvaru (vplyv, rodina, id pacienta).
- Malé vstupy: vždy prejdeme celú čakáreň, keď doktor volá ďalšieho.
- Málo rodín: vo vzorovom riešení (viď nižšie) namiesto usporiadanej množiny rodín použijeme hrubú silu.

Vzorové riešenie

Použijeme nasledovné dátové štruktúry:

- Pole, v ktorom si pre každú rodinu budeme pamätať jej aktuálny vplyv.
- Pole front, v ktorých máme pre každú rodinu tých jej členov, ktorí sú momentálne v čakárni.
- Usporiadanú množinu C, v ktorej sú záznamy pre rodiny, ktoré momentálne majú niekoho v čakárni. Tieto sú usporiadané podľa aktuálneho vplyvu rodín.

Keď doktor zavolá do ordinácie ďalšieho pacienta, spravíme nasledovné:

1. Pozrieme sa na najväčší prvok C. Dozvieme sa rodinu, ktorej člena máme poslať dnu.
2. Vyberieme najdlhšie čakajúceho z fronty pre danú rodinu.
3. Ak už táto rodina nemá nikoho ďalšieho čakajúceho, odstránime jej záznam z C.

Keď do čakárne príde nový pacient, pozrieme sa, či z jeho rodiny už máme iného čakajúceho. Ak nie, spracujeme ho nasledovne:

1. Ak treba, upravíme si vplyv jeho rodiny.
2. Do C vložíme nový záznam o tejto rodine.
3. Do fronty pre túto rodinu vložíme pacienta, ktorý práve prišiel.

Ak tu jeho rodina už niekoho čakajúceho má, postup bude trochu iný:

1. Pozrieme sa, či nám nový pacient zmení názor na vplyv jeho rodiny. Ak áno, odstránime dočasne záznam tejto rodiny z C, potom si upravíme jej vplyv a následne vložíme do C nový záznam s upraveným vplyvom.
2. Do fronty pre túto rodinu vložíme pacienta, ktorý práve prišiel.

Každú operáciu s množinou C vieme spraviť v čase $O(\log n)$, kde n je počet rodín, ktoré práve obsahuje. Všetky ostatné operácie vieme spraviť v konštantnom čase. Dokopy teda každú udalosť spracujeme v čase nanajvýš logaritmickom od celkového počtu rodín.
Podotkneme ešte, že namiesto usporiadanej množiny (setu) sa dá použiť aj halda, len to potom vyžaduje trochu komplikovanejšiu implementáciu: potrebujeme si pre každú rodinu udržiavať pointer/index na jej záznam v halde a potom pri zmene vplyvu rodiny použiť operáciu IncreaseKey – teda záznam rodiny „prebublať“ bližšie ku koreňu, na miesto kam patrí podľa nového vplyvu.

Listing programu (C++)

```
#include <algorithm>
#include <iostream>
#include <queue>
#include <set>
#include <vector>
using namespace std;

const int MAXR = 200000;

struct rodina { int id, vplyv; };
bool operator< (const rodina &A, const rodina &B) {
```

```
    if (A.vplyv != B.vplyv) return A.vplyv < B.vplyv;
    return A.id > B.id;
}

int dalsie_id;
set<rodina> rodiny_v_cakarni;
vector<int> vplyv_rodiny;
vector<queue<int> > pacienti_v_cakarni;

void do_cakarne() {
    int id = dalsie_id++;
    int vplyv, rodina;
    cin >> vplyv >> rodina;
    if (vplyv > vplyv_rodiny[rodina]) {
        rodiny_v_cakarni.erase( {rodina, vplyv_rodiny[rodina]} );
        vplyv_rodiny[rodina] = vplyv;
    }
    rodiny_v_cakarni.insert( {rodina, vplyv_rodiny[rodina]} );
    pacienti_v_cakarni[rodina].push(id);
};

int do_ordinacie() {
    if (rodiny_v_cakarni.empty()) return -1;
    int rodina = rodiny_v_cakarni.rbegin()->id;
    int vitaz = pacienti_v_cakarni[rodina].front();
    pacienti_v_cakarni[rodina].pop();
    if (pacienti_v_cakarni[rodina].empty()) {
        rodiny_v_cakarni.erase( {rodina, vplyv_rodiny[rodina]} );
    }
    return vitaz;
}

int main() {
    dalsie_id = 0;
    vplyv_rodiny.resize(MAXR, -1);
    pacienti_v_cakarni.resize(MAXR);
    while (true) {
        int typ;
        cin >> typ;
        if (typ == 0) break;
        if (typ == 1) do_cakarne();
        if (typ == 2) cout << do_ordinacie() << "\n";
    }
}
```

A-III-5 Voda

Čiastočné body

Niektoré body za ľahšie sady vstupov sa dalo získať za objavenie a implementáciu nasledovných pozorovaní:

- Ak sa mapa skladá z viacerých komponentov súvislosti, vieme vyriešiť každý zvlášť a odpovede vynásobiť.
- Pre obdĺžnik tvorený x riadkami existuje presne $x + 1$ možností, ako doň dať vodu.
- Kvaple zhora nič zásadné nemenia: obdĺžnik s kvapľami má stále v podstate tie isté riešenia ako bez nich.
- Pre kvaple zdola sa dajú ľahšie objaviť aj implementovať myšlienky vedúce k plnému riešeniu úlohy.

Vzorové riešenie

Ak máme v riadku vedľa seba niekoľko prázdnych políčok, ich stav bude v každom riešení rovnaký: ak niektoré z nich má vodu, musia ju mať všetky. Preto sa môžeme na každý takýto úsek dívať ako na jeden objekt. Začneme teda tým, že prejdeme mapu jaskyne a každému úseku priradíme jeho poradové číslo od 0 po $n - 1$.

Všimnime si teraz sadu úsekov v nasledovnom príklade. Úseky označené číslami 0 a 1 sa navzájom priamo neovplyvňujú – je možné, že jeden z nich má vodu a druhý nie. Ale úseky s číslami 2, 3 a 4 majú pre nás zaujímavú vlastnosť: akonáhle obsahuje vodu ľubovoľný jeden z nich, obsahujú ju nutne všetky tri, takže všetky tri majú nutne pre každú jaskyňu s vodou ten istý stav. Takéto úseky budeme volať *ekvivalentné*.

```
#22#333#44#
##000#111##
###########
```

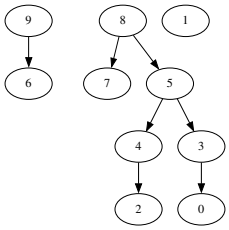
Presnejšie, dva úseky v tom istom riadku mapy budeme volať *ekvivalentné*, ak v každej možnej jaskyni s vodou majú rovnaký stav – buď je v oboch voda alebo sú oba prázdne.

Pozrime sa na ľubovoľný riadok mapy. Ako vieme povedať, ktoré dvojice úsekov v ňom sú ekvivalentné? Zabudnime nateraz na všetky vyššie riadky mapy a pozrime sa len na všetko od tohto riadku dodola. V tejto časti mapy vieme všetky voľné políčka rozdeliť na komponenty súvislosti. Tvrdíme, že dva úseky v našom riadku sú ekvivalentné práve vtedy, ak sú v tomto „odrezanom“ kuse mapy súčasťou toho istého komponentu.
Toto je ľahké zdôvodniť. Ak úseky ležia v rôznych komponentoch, tak ekvivalentné byť nemôžu, lebo dať jeden komponent plný vody a úplne celý zvyšok jaskyne (vrátane druhého komponentu) prázdny je zjavne platné riešenie. A naopak, ak dva úseky ležia v tom istom komponente, tak existuje cesta *vnútrom daného komponentu* z jedného z nich do druhého a ľahko nahliadneme, že ak má prvý úsek vodu, tak musia mať vodu postupne všetky políčka tejto cesty, a teda aj druhý úsek.
Ekvivalentné úseky teda vieme nájsť tak, že postupne prechádzame mapu jaskyne zdola hore a priebežne si počas toho udržiavame informáciu o komponentoch súvislosti. Dá sa to implementovať v lineárnom čase pomocou postupných prehľadávaní, ale v ukážkovej implementácii uvedenej nižšie sme si vybrali o chlp pomalšiu ale implementačne jednoduchšiu možnosť: algoritmus union-find.

Počas práve popísaného prechodu mapou priradíme úsekom nové čísla tak, aby ekvivalentné úseky dostali rovnaké číslo – čím ich akoby spojíme do jedného väčšieho logického celku. Príklad takéhoto očíslovania pre jednu väčšiu jaskyňu je na obrázku nižšie. Všimnite si, že v rámci riadku toto nové číslovanie nemusí byť súvislé.

```
####################      ####################
#.####..#####.#...#      #8####99#####8#888#
#.#.#...#.....#.#.#      #5#5#666#55555#5#7#
#...#####.###.#.###  ->  #333#####3###4#4###
###.......#.#.....#      ###0000000#1#22222#
####################      ####################
```

Toto nové očíslovanie jaskyne už nesie všetku informáciu potrebnú na to, aby sme vedeli ľahko spočítať, koľko existuje rôznych rozmiestnení vody do nej. Ukážeme si teraz, prečo to tak je aj ako tento výpočet spraviť.
Pre konkrétne číslo x sa pozrime na tie čísla $y_1, \ldots, y_k$, ktoré ležia bezprostredne pod x, teda na ktoré vieme z x spraviť krok dodola. Čísla y_i budeme volať *deťmi* čísla x a číslo x bude ich *rodičom*. (Napr. na obrázku vyššie sú deťmi čísla $x = 5$ čísla 3 a 4. Na obrázku nižšie sú znázornené všetky tieto vzťahy pre náš príklad)



Všimnime si, že žiadne číslo nemôže mať viac ako jedného rodiča – inými slovami, dve rôzne x a y nikdy nemôžu mať spoločné dieťa z. V takejto situácii by sme totiž už pri predchádzajúcom prechode mapou jaskyne zisili, že x a y sú ekvivalentné, a teda by dostali to isté číslo, nie dve rôzne čísla x a y. Naše nové očíslovanie jaskyne má teda stromovú topológiu. Presnejšie, ide o *les*, ktorý sa môže skladať z viacerých úplne nezávislých stromov.
Pozrime sa teraz na ľubovoľné konkrétne číslo x na takto spracovanej mape jaskyne a položme si nasledovnú otázku: koľkými spôsobmi vieme rozmiestniť vodu v časti jaskyne, ktorá má vo svojom najvyššom riadku políčka s týmto číslom a okrem nich obsahuje všetko dosiahnuteľné z týchto políčok dodola? Odpoveď na túto otázku označíme s_x. (V grafovej terminológii je s_x počet spôsobov, ako rozmiestniť vodu v podstrome s koreňom x.)
Vo vyššie uvedenom príklade pre číslo 5 pôjde o oblasť tvorenú číslami 5, 4, 3, 2 a 0.
Čísla s_x vieme spočítať efektívne ďalším prechodom cez mapu jaskyne zdola hore. Pre konkrétne číslo x budeme postupovať nasledovne:

- Ak je na čísle x voda, je voda nutne aj vo všetkých dodola dosiahnuteľných úsekoch. Toto nám vždy dáva práve jednu možnosť.
- Ak je na čísle x sucho, dostávame pre každé z detí čísla x samostatný podproblém. (Napr. v našom príklade rozhodnutia, kde je a kde nie je voda v časti jaskyne začínajúcej číslom 3 nijak neovplyvňujú rozhodnutia, kde je a kde nie je voda v časti jaskyne začínajúcej číslom 4 a naopak.)

Celkový počet takýchto rozmiestnení vody preto vypočítame tak, že pre každé dieťa y sa pozrieme na jeho počet s_y (ktorý sme už skôr spočítali) a tieto počty medzi sebou vynásobíme.

(Špeciálne sa pozrime na prípad, kedy nejaké číslo nemá žiadne deti. Vtedy dostaneme prázdny súčin s hodnotou 1. Inými slovami, v tejto situácii máme práve jednu novú možnosť „tieto políčka sú suché a pod nimi už nič nie je“.)

Výslednú hodnotu, ktorú máme vypísať na výstup, potom dostaneme tak, že medzi sebou vynásobíme počty možností pre jednotlivé stromy, na ktoré sa nám mapa jaskyne rozpadla. Odpoveďou je teda súčin hodnôt s_x pre tie x, ktoré nemajú rodiča.

Listing programu (Python)

```
from random import randint, seed
from collections import defaultdict
seed(47)

# union-find: robíme kompresiu cesty pri find() a náhodné spájanie pri union()
def get_root(boss, v):
    if v == boss[v]:
        return v
    else:
        tmp = get_root(boss, boss[v])
        boss[v] = tmp
        return tmp

def union(boss, u, v):
    ru, rv = get_root(boss, u), get_root(boss, v)
    if ru == rv: return
    if randint(0, 1): ru, rv = rv, ru
    boss[rv] = ru

# načítame mapu jaskyne a zdola hore očíslujeme úseky políčok
R, C = [ int(_) for _ in input().split() ]
mapa = [ input() for r in range(R) ]
N = 0
segment = [ [ -1 for c in range(C) ] for r in range(R) ]
for r in reversed(range(R)):
    for c in range(C):
        if mapa[r][c] != '.': continue
        if mapa[r][c-1] != '.': N += 1
        segment[r][c] = N-1

# pomocou union-find nájdeme komponenty súvislosti a vypočítame nové čísla
boss = list(range(N))
T = 0
trieda = [ -1 for _ in range(N) ]
for r in reversed(range(R)):
    for c in range(C):
        if segment[r][c] != -1 and segment[r+1][c] != -1:
            union(boss, segment[r][c], segment[r+1][c] )
    nove_triedy = {}
    for c in range(C):
        if segment[r][c] != -1:
            sr = get_root(boss, segment[r][c])
            if sr not in nove_triedy:
                nove_triedy[sr], T = T, T+1
            trieda[ segment[r][c] ] = nove_triedy[sr]

# nájdeme si pre každé číslo jeho deti
deti = [ set() for _ in range(T) ]
for r in reversed(range(R)):
    for c in range(C):
        if segment[r][c] != -1 and segment[r+1][c] != -1:
            deti[ trieda[ segment[r][c] ] ].add( trieda[ segment[r+1][c] ] )

# prechodom zdola hore spočítame počty riešení pre všetky podstromy
MOD = 10**9 + 7
odpovede = [ 1 for _ in range(T) ]
for t in range(T):
    for d in deti[t]:
        odpovede[t] = (odpovede[t] * odpovede[d]) % MOD
    odpovede[t] = (odpovede[t] + 1) % MOD

# nájdeme všetky korene stromov a vynásobíme ich odpovede
top = [ True for _ in range(T) ]
for t in range(T):
    for d in deti[t]:
        top[d] = False

spolu = 1
for t in range(T):
    if top[t]:
        spolu = (spolu * odpovede[t]) % MOD

print(spolu)
```

A-III-6 Nechaj to na solver III

Podúloha A: investície so synergiami

Okrem doterajších binárnych premenných x_i pre jednotlivé projekty zavedieme aj nové binárne premenné y_j pre jednotlivé synergie.
Profit, ktorý sa snažíme maximalizovať, bude mať teraz aj členy zodpovedajúce synergiám: pre každé j k nemu pripočítame hodnotu $y_j \cdot t_j$ (kde t_j je príslušný bonus/malus, ktorý dostaneme, ak je $y_j = 1$.

Teraz potrebujeme povoleným spôsobom zapísať podmienku „$y_j = 1$ vtedy a len vtedy, ak sú podporené všetky projekty $u_{i,1}$ až u_{i,ℓ_i}“.
Toto by sme vedeli matematicky zapísať tak, že povieme, že y_j má byť rovné súčinu príslušných x_i. Takýto zápis je síce matematicky správny, ale v ILP nevieme násobiť premenné. Musíme teda vymyslieť, ako tento istý vzťah zapísať ináč. Spravíme to nasledovne:
Pre každý projekt i, ktorého sa týka táto synergia, budeme mať podmienku $y_j \leq x_i$. Ak teda projekt i nepodporíme, y_j bude nutne tiež 0.

Ak by všetky synergie mali nezáporné bonusy t_j, toto by stačilo: keďže maximalizujeme profit, vždy, keď môžeme mať $y_j = 1$, to naozaj solver spraví. Niektoré vstupy mali túto vlastnosť a solver vám ich potom mal správne vyriešiť už s vyššie uvedenou sadou podmienok.
Ak ale niektoré t_j môžu byť záporné, tento ILP by ešte nefungoval správne: solver by mohol nájsť nepovolené riešenie, v ktorom síce všetky relevantné x_i budú 1, ale zároveň nechá $y_j = 0$, lebo nechce zobrať záporné t_j. Toto mu musíme zakázať – teda povedať mu ešte jednu podmienku hovoriacu „ak sú všetky tieto $x_i = 1$, tak aj y_j musí byť 1“.
Bez násobenia toto vieme dosiahnuť nasledovne: $y_j \geq x_{u_{j,1}} + \cdots + x_{u_{j,\ell_j}} - (\ell - 1)$.
Ak je niektoré z použitých x_i nula, táto podmienka je automaticky splnená. A ak sú všetky rovné 1, dostávame $y_j \geq \ell - (\ell - 1) = 1$, čo je presne to, čo sme chceli.

Listing programu (Python)

```
import pulp

def solve_one():
    # načítame vstup
    e, n = [ int(_) for _ in input().split() ]
    S = [ int(_) for _ in input().split() ]
    V = [ int(_) for _ in input().split() ]

    z = int(input())
    D = [ [ int(_)-1 for _ in input().split() ] for __ in range(z) ] # -1 lebo interne cislujeme od 0

    k = int(input())
    C = [ [ int(_)-1 for _ in input().split() ] for __ in range(k) ]

    q = int(input())
    T = [ int(_) for _ in input().split() ]
    ignore = input()
    U = [ [ int(_)-1 for _ in input().split() ] for __ in range(q) ]

    # vyrobíme si premenné
    problem = pulp.LpProblem('Investicie', pulp.LpMaximize)
    project = [ pulp.LpVariable(f'project{i}', cat='Binary') for i in range(n) ]
    synergy = [ pulp.LpVariable(f'synergy{i}', cat='Binary') for i in range(q) ]

    # zadáme výraz, ktorého hodnotu maximalizujeme
    problem += e + pulp.lpSum( project[i] * (V[i] - S[i]) for i in range(n) ) + pulp.lpSum( synergy[j] * T[j] for j in range(q) )

    # pridáme podmienky
    problem += pulp.lpSum( project[i] * S[i] for i in range(n) ) <= e
    for conflict in C:
        problem += pulp.lpSum( project[i] for i in conflict ) <= 1
    for dependency in D:
        problem += project[dependency[0]] <= project[dependency[1]]
    for sval, inputs in zip(synergy, U):
        for x in inputs: problem += sval <= project[x]
        problem += sval >= pulp.lpSum( project[x] for x in inputs ) - len(inputs) + 1

    # spustíme solver a vypíšeme výsledok
    status = problem.solve(pulp.PULP_CBC_CMD(msg=0))
    assert pulp.LpStatus[status] == 'Optimal'
    answer = int(round( pulp.value(problem.objective) ))
    print(answer)
```

```
tc = int( input() )
for test in range(tc): solve_one()
```

Podúloha B: pílenie tyčí

V tejto úlohe riešime rôzne variácie na tzv. viacrozmerný problém batoha. Popíšeme si niekoľko rôznych postupov, ktoré sa dali použiť.

Pre prvé dva vstupy sme si mohli vystačiť s jednoduchými binárnymi premennými: tyče kúpené v obchode si očíslujeme od 0 po $t-1$, naše želané tyče od 0 po $n-1$ a pre každú dvojicu (i, j) budeme mať binárnu premennú $x_{i,j}$ hovoriacu, či z obchodnej tyče i odrezať želanú tyč j. Podmienky nášho ILP sú potom vcelku priamočiare: pre každé j musí niektoré $x_{i,j}$ byť 1 (stačí podmienka, že ich súčet je aspoň 1) a pre každé i musí platiť, že súčet dĺžok tých tyčí, ktoré odrežeme z obchodnej tyče i, nesmie prekročiť ℓ.

Toto riešenie môžeme ďalej zovšeobecniť tak, že namiesto binárnych premenných použijeme nezáporné celočíselné: pre každú obchodnú tyč i a každý *typ* želaných tyčí j budeme mať premennú $y_{i,j}$ hovoriacu *koľko* kusov tyče j odpílime z tyče i. Podmienky sú prirodzeným zovšeobecnením tých uvedených vyššie: pre každé j musí súčet všetkých $x_{i,j}$ väčší alebo rovný ako želaný počet p_j tyčí daného typu, a pre každé i musíme z i-tej obchodnej tyče vyrobiť kusy s celkovou dĺžkou nepresahujúcou ℓ. Takýto program by mal úspešne vyriešiť prvé tri testovacie vstupy.

Listing programu (Python)

```
import pulp

def over_pocet(t, l, dlzky, pocty):
    n = len(dlzky)

    problem = pulp.LpProblem('Investicie', pulp.LpMaximize)
    kolko = [ [ pulp.LpVariable(f'odpil_{i}_{j}', cat='Integer') for j in range(t) ] for i in range(n) ]
    problem += t
    for i in range(n):
        for j in range(t):
            problem += kolko[i][j] >= 0
    for j in range(t):
        problem += sum( kolko[i][j] * dlzky[i] for i in range(n) ) <= l
    for i in range(n):
        problem += sum( kolko[i][j] for j in range(t) ) >= pocty[i]

    status = problem.solve(pulp.PULP_CBC_CMD(msg=0))
    if pulp.LpStatus[status] != 'Optimal': return False

    kolko = [ [ int(round(pulp.value(kolko[i][j]))) for j in range(t) ] for i in range(n) ]
    print(t)
    print(t)
    for j in range(t):
        tyc = [ f'{kolko[i][j]}x{dlzky[i]}' for i in range(n) if kolko[i][j] ]
        print(1, *tyc)
    return True

def solve_one():
    l = int(input())
    n = int(input())
    dlzky = [ int(_) for _ in input().split() ]
    pocty = [ int(_) for _ in input().split() ]
    celkova_dlzka = sum( d*p for d,p in zip(dlzky, pocty) )
    t = (celkova_dlzka + l - 1) // l
    while not over_pocet(t, l, dlzky, pocty): t += 1

tc = int( input() )
for test in range(tc): solve_one()
```

Generovanie možných tyčí

V posledných dvoch testovacích vstupoch zjavne platí, že z každej obchodnej tyče dostaneme len zopár želaných tyčí. Ľahko nahliadneme, že existuje len nanajvýš pár tisíc možností, akú sadu želaných tyčí vieme vyrobiť z obchodnej tyče. Naša nová stratégia preto bude nasledovná: Vygenerujeme si všetky tieto možnosti a pre každú z nich budeme mať jednu celočíselnú premennú hovoriacu, koľko obchodných tyčí chceme rozpíliť týmto konkrétnym spôsobom. (V sade 4 môžeme dokonca mať binárne premenné, keďže sa nemôže oplatiť dve obchodné tyče rozpíliť rovnako. A tiež môžeme generovať len také rozpílenia, v ktorých sa žiadna tyč neopakuje.)

Ako tentokrát vyzerajú podmienky, ktorými náš ILP zabezpečí, že dostaneme optimálne platné riešenie? Pre každý typ želaných tyčí nám stačí mať podmienku, že ich celkový počet je dostatočný. No a tentokrát namiesto len kontroly, či riešenie existuje, použijeme solver priamo na optimalizáciu: dáme mu za úlohu minimalizovať súčet všetkých premenných – inými slovami, celkový počet použitých obchodných tyčí.

Listing programu (Python)

```
import pulp
import sys

def solve_one():
    l = int(input())
    n = int(input())
    dlzky = [ int(_) for _ in input().split() ]
    pocty = [ int(_) for _ in input().split() ]

    patterns = set( [ tuple() ] )
    for i in range(n):
        new_patterns = set()
        for pat in patterns:
            new_patterns.add(pat)
            pat = list(pat)
            while True:
                pat.append(dlzky[i])
                if pat.count(dlzky[i]) > pocty[i]: break
                if sum(pat) > l: break
                new_patterns.add( tuple(pat) )
        patterns = new_patterns

    patterns = [ [ pat.count(d) for d in dlzky ] for pat in patterns ]
    p = len(patterns)

    celkova_dlzka = sum( d*p for d,p in zip(dlzky, pocty) )
    t = (celkova_dlzka + l - 1) // l

    problem = pulp.LpProblem('Tyce', pulp.LpMinimize)
    kolko = [ pulp.LpVariable(f'kusov_{i}', cat='Integer') for i in range(p) ]

    problem += sum( kolko[i] for i in range(p) )
    for i in range(p):
        problem += kolko[i] >= 0
    for j in range(n):
        problem += sum( kolko[i] * patterns[i][j] for i in range(p) if patterns[i][j] ) >= pocty[j]

    status = problem.solve(pulp.PULP_CBC_CMD(msg=0))
    assert pulp.LpStatus[status] == 'Optimal'
    kolko = [ int(round( pulp.value( kolko[i] ) )) for i in range(p) ]

    print( int(round( pulp.value( problem.objective ) )) ) # celkový počet tyčí
    print( sum(x > 0 for x in kolko) ) # počet inštrukcií
    for i in range(p):
        if kolko[i] == 0: continue
        tyc = [ f'{patterns[i][j]}x{dlzky[j]}' for j in range(n) if patterns[i][j] ]
        print( kolko[i], *tyc )

tc = int( input() )
for test in range(tc): solve_one()
```

ŠTYRIDSIATY PRVÝ ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Forišek, Sebastian Hajdu, Paulína Smolárová
Recenzia: Michal Anderle, Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: NIVAM – Národný inštitút vzdelávania a mládeže, Bratislava 2026