P-III-4 Mafiáni u doktora

Částečné body

Body za jednodušší sady vstupů bylo možné získat následovně:

- Všichni jsou jedna rodina: ošetřují se v pořadí, v jakém přišli, stačí si pamatovat, kolik pacientů zbývá v čekárně a nejmenší z jejich čísel.
- Nejprve všichni do čekárny: stačí všechny pacienty jednou seřadit, když doktor zavolá prvního z nich.
- Všichni mají stejný vliv; či nikdo nemá stejný vliv ani rodinu: v obou případech si stačí v haldě pamatovat záznamy ve tvaru (vliv, rodina, id pacienta).
- Malé vstupy: vždy projdeme celou čekárnu, když lékař volá dalšího pacienta.
- Málo rodin: ve vzorovém řešení (viz níže) stačí místo uspořádané množiny rodin použít hrubou sílu.

Vzorové řešení

Provedeme jednoduchou simulaci s použitím následujících datových struktur:

- Pole, ve kterém budeme pro každou rodinu uchovávat její aktuální vliv.
- Pole front, ve kterých máme pro každou rodinu ty její členy, kteří jsou momentálně v čekárně.
- Uspořádaná množina C, ve které jsou záznamy pro rodiny, které momentálně mají někoho v čekárně. Tyto jsou uspořádány podle aktuálního vlivu rodin.

Když lékař zavolá do ordinace dalšího pacienta, provedeme následující:

1. Podíváme se na největší prvek C a tím zjistíme rodinu, jejíhož člena máme poslat dovnitř.
2. Vybereme z fronty dané rodiny toho, kdo čeká nejdéle.
3. Pokud tato rodina již nemá nikoho dalšího čekajícího, odstraníme její záznam z C.

Když do čekárny přijde nový pacient, zkontrolujeme, zda už z jeho rodiny někdo čeká. Pokud ne, zpracujeme ho následovně:

1. V případě potřeby upravíme vliv jeho rodiny.
2. Do C vložíme nový záznam o této rodině.
3. Do fronty pro tuto rodinu vložíme pacienta, který právě přišel.

Pokud zde jeho rodina již má někoho čekajícího, postup bude trochu jiný:

1. Zjistíme, zda nám nový pacient změní názor na vliv jeho rodiny. Pokud ano, dočasně odstraníme záznam této rodiny z C, poté upravíme její vliv a následně vložíme do C nový záznam s upraveným vlivem.
2. Do fronty pro tuto rodinu vložíme pacienta, který právě přišel.

Každou operaci s množinou C můžeme provést v čase $\mathcal{O}(\log r)$, jelikož v každém okamžiku tato množina obsahuje pro každou z r rodin nejvýše jeden záznam. Všechny ostatní operace můžeme provést v konstantním čase. Celkově tedy každou událost zpracujeme v čase $\mathcal{O}(\log r)$.

Ještě poznamenejme, že místo uspořádané množiny (setu) lze použít i haldu, jen to pak vyžaduje o něco složitější implementaci: Pro každou rodinu si musíme udržovat ukazatel/index na její záznam v haldě a poté při změně vlivu rodiny použít operaci IncreaseKey – tedy záznam rodiny posunout blíže ke kořenu, na místo, kam patří podle nového vlivu.

Program (C++):

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

const int MAXR = 200000;

struct rodina { int id, vliv; };
bool operator< (const rodina &A, const rodina &B) {
    if (A.vliv != B.vliv) return A.vliv < B.vliv;
    return A.id > B.id;
}

int dalsi_id;
set<rodina> rodiny_v_cekarne;
vector<int> vliv_rodiny;
vector<queue<int> > pacienti_v_cekarne;

void do_cekarny() {
    int id = dalsi_id++;
    int vliv, rodina;
    cin >> vliv >> rodina;
    if (vliv > vliv_rodiny[rodina]) {
        rodiny_v_cekarne.erase( {rodina, vliv_rodiny[rodina]} );
        vliv_rodiny[rodina] = vliv;
    }
    rodiny_v_cekarne.insert( {rodina, vliv_rodiny[rodina]} );
    pacienti_v_cekarne[rodina].push(id);
};

int do_ordinace() {
    if (rodiny_v_cekarne.empty()) return -1;
    int rodina = rodiny_v_cekarne.rbegin()->id;
    int vitaz = pacienti_v_cekarne[rodina].front();
```

```
    pacienti_v_cekarne[rodina].pop();
    if (pacienti_v_cekarne[rodina].empty()) {
        rodiny_v_cekarne.erase( {rodina, vliv_rodiny[rodina]} );
    }
    return vitaz;
}

int main() {
    dalsi_id = 0;
    vliv_rodiny.resize(MAXR, -1);
    pacienti_v_cekarne.resize(MAXR);
    while (true) {
        int typ;
        cin >> typ;
        if (typ == 0) break;
        if (typ == 1) do_cekarny();
        if (typ == 2) cout << do_ordinace() << "\n";
    }
}
```

P-III-5 Voda

Částečné body

Některé body za jednodušší sady vstupů bylo možné získat za objev a implementaci následujících postřehů:

- Pokud se mapa skládá z více místností, můžeme každou z nich vyřešit samostatně a výsledky vynásobit.
- Pro obdélník tvořený x řádky existuje přesně $x + 1$ možností, jak do něj nalít vodu.
- Stalaktity shora nic zásadního nemění: obdélník se stalaktity má v podstatě stále stejná řešení jako bez nich.
- Pro sgalagmity lze snáze objevit i implementovat myšlenky vedoucí k úplnému řešení úlohy.

Vzorové řešení

Máme-li v řádku vedle sebe několik prázdných políček, bude jejich stav v každém řešení stejný: pokud některá z nich obsahují vodu, musí ji obsahovat všechna. Proto můžeme na každý takový úsek políček pohlížet jako na jeden objekt. Začneme tedy tím, že projdeme mapu jeskyně a každému úseku přiřadíme pořadové číslo od 0 do $n - 1$, například takto:

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

V tomto příkladu se úseky označené čísly 0 a 1 se navzájem přímo neovlivňují; je možné, že jeden z nich obsahuje vodu a druhý ne. Ale úseky s čísly 2, 3 a 4 mají pro nás zajímavou vlastnost: jakmile obsahuje vodu libovolný z nich, obsahují ji nutně všechny tři, takže všechny tři mají nutně pro každou jeskyni s vodou stejný stav. Takové úseky budeme nazývat *ekvivalentní*. Přesněji řečeno, dva úseky ve stejném řádku mapy budeme nazývat *ekvivalentní*, pokud mají v každé možné jeskyni s vodou stejný stav, tedy buď je v obou voda, nebo jsou obě prázdné.

Podívejme se na libovolný řádek mapy. Jak můžeme určit, které dvojice úseků v něm jsou ekvivalentní? Zapomeňme prozatím na všechny předchozí řádky mapy a podívejme se pouze na vše od tohoto řádku dolů. V této části mapy můžeme všechna volná políčka rozdělit na komponenty souvislosti. Tvrdíme, že dva úseky v našem řádku jsou ekvivalentní právě tehdy, jsou-li v tomto „odříznutém“ kusu mapy součástí téže komponenty.

To lze snadno odůvodnit: Pokud úseky leží v různých komponentách, nemohou být ekvivalentní, protože naplnit jednu komponentu vodou a celý zbytek jeskyně (včetně druhé komponenty) nechat prázdný dává stabilní jeskyni s vodou. Naopak, pokud dva úseky leží ve stejné komponentě, existuje cesta *uvnitř dané komponenty* z jednoho z nich do druhého a snadno si uvědomíme, že pokud má první úsek vodu, musí mít vodu postupně všechna políčka této cesty, a tedy i druhý úsek.

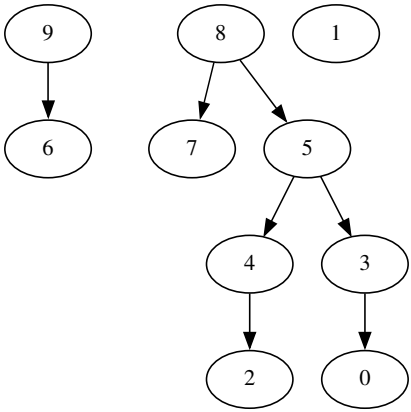
Ekvivalentní úseky tedy můžeme najít tak, že postupně procházíme mapu jeskyně zdola nahoru a během toho průběžně udržujeme informace o komponentách souvislosti. To je možné i v lineárním čase pomocí postupného prohledávání, ale v ukázkové implementaci uvedené níže jsme zvolili jen trochu pomalejší, ale implementačně jednodušší možnost pomocí algoritmu union-find.

Během právě popsaného průchodu mapou přiřadíme úsekům nová čísla tak, aby ekvivalentní úseky dostaly stejné číslo, čímž je jakoby spojíme do jednoho většího logického celku. Příklad takového číslování pro jednu větší jeskyni je na obrázku níže. Všimněte si, že v rámci řádku nemusí být toto nové číslování souvislé.

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

Toto nové číslování jeskyně již obsahuje všechny informace potřebné k tomu, abychom mohli snadno spočítat, kolik různých rozložení vody v ní existuje. Nyní si ukážeme, proč to tak je a jak tento výpočet provést.

Pro každé číslo x se podívejme na ta čísla $y_1, \ldots, y_k$, které leží bezprostředně pod x, tedy na ta, do kterých se z úseků s číslem x můžeme dostat jedním krokem přímo dolů. Čísla y_i budeme nazývat *dětmi* čísla x a číslo x bude jejich *rodičem*. Např. na obrázku výše jsou dětmi čísla $x = 5$ právě čísla 3 a 4. Na obrázku níže jsou znázorněny všechny tyto vztahy pro náš příklad.



Všimněme si, že žádné číslo nemůže mít více než jednoho rodiče, či jinými slovy, dvě různá čísla x a y nikdy nemohou mít společné dítě z. V takové situaci bychom totiž již při předchozím průchodu mapou jeskyně zjistili, že x a y jsou ekvivalentní, a tedy bychom jim přidělili stejné číslo. Naše nové očíslování jeskyně

má tedy stromovou topologii. Přesněji řečeno, jde o *les*, který se může skládat z více zcela nezávislých stromů.

Podívejme se nyní na libovolné číslo x na takto zpracované mapě jeskyně a položme si následující otázku: kolika způsoby můžeme rozmístit vodu v části jeskyně, která má ve svém nejvyšším řádku políčka s tímto číslem a kromě nich obsahuje vše, co je z těchto políček dosažitelné směrem dolů (ve výše uvedeném příkladu se u čísla 5 jedná o oblast tvořenou čísly 5, 4, 3, 2 a 0)? Odpověď na tuto otázku označíme s_x. V grafové terminologii je s_x počet způsobů, jak rozmístit vodu v podstromu s kořenem x.

Čísla s_x můžeme efektivně spočítat dalším průchodem mapou jeskyně zdola nahoru. Pro konkrétní číslo x hodnotu s_x dostaneme sečtením počtů ze dvou případů:

- Pokud je na čísle x voda, je voda nutně i ve všech úsecích dosažitelných dolů. To nám vždy dává právě jednu možnost.
- Pokud je na čísle x sucho, dostáváme pro každé z dětí čísla x samostatný podproblém. Např. v našem příkladu rozhodnutí, kde je a kde není voda v části jeskyně začínající číslem 3, nijak neovlivňují rozhodnutí, kde je a kde není voda v části jeskyně začínající číslem 4 a naopak.

  Celkový počet takovýchto rozmístění vody proto vypočítáme tak, že se pro každé dítě y podíváme na jeho počet s_y (který jsme již dříve spočítali) a tyto počty mezi sebou vynásobíme.

  Speciálně v případě, kdy nějaké číslo nemá žádné potomky, dostaneme prázdný součin s hodnotou 1. Jinými slovy, v této situaci máme právě jednu novou možnost „tato políčka jsou suchá a pod nimi už nic není“.

Výslednou hodnotu, kterou máme vypsat na výstup, pak získáme tak, že mezi sebou vynásobíme počty možností pro jednotlivé stromy, na které se mapa jeskyně rozložila. Odpovědí je tedy součin hodnot s_x pro ta čísla x, která nemají rodiče.

Program (Python):

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

# union-find pomocí komprese cesty při find() a náhodném spojování při 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
```

```
# načteme mapu jeskyně a zdola nahoru očíslujeme úseky políček
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

# pomocí union-find najdeme komponenty souvislosti a vypočteme nová čísla
boss = list(range(N))
T = 0
trida = [ -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_tridy = {}
    for c in range(C):
        if segment[r][c] != -1:
            sr = get_root(boss, segment[r][c])
            if sr not in nove_tridy:
                nove_tridy[sr], T = T, T+1
            trida[ segment[r][c] ] = nove_tridy[sr]

# najdeme si pro každé číslo jeho děti
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[ trida[ segment[r][c] ] ].add( trida[ segment[r+1][c] ] )

# průchodem zdola nahoru spočítame počty řešení pro všechny podstromy
MOD = 10**9 + 7
odpovedi = [ 1 for _ in range(T) ]
for t in range(T):
    for d in deti[t]:
        odpovedi[t] = (odpovedi[t] * odpovedi[d]) % MOD
    odpovedi[t] = (odpovedi[t] + 1) % MOD

# najdeme všechny kořenu stromů a vynásobíme jejich odpovědí
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 * odpovedi[t]) % MOD

print(spolu)
```

P-III-6 Návrat řešiče

Podúloha A: investice se synergiemi

Kromě dosavadních binárních proměnných x_i pro jednotlivé projekty zavedeme také nové binární proměnné y_j pro jednotlivé synergie. Zisk, který se snažíme maximalizovat, bude nyní obsahovat i členy odpovídající synergiím, pro každé j k němu tedy přičteme hodnotu $y_j \cdot t_j$, kde t_j je příslušný bonus/malus, který získáme, pokud je $y_j = 1$.

Nyní potřebujeme povoleným způsobem zapsat podmínku „$y_j = 1$ tehdy a jen tehdy, pokud jsou podpořeny všechny projekty $u_{i,1}$ až u_{i,ℓ_i}“. To bychom mohli matematicky zapsat tak, že řekneme, že y_j se má rovnat součinu příslušných x_i. Takový zápis je sice matematicky správný, ale v ILP neumíme násobit proměnné. Musíme tedy vymyslet, jak tento stejný vztah zapsat jinak. Uděláme to následovně.

Nejprve pro každý projekt i, kterého se tato synergie týká, budeme mít podmínku $y_j \leq x_i$. Pokud tedy projekt i nepodpoříme, bude y_j nutně také 0. Kdyby všechny synergie měly nezáporné bonusy t_j, stačilo by to: jelikož maximalizujeme zisk, vždy, když můžeme mít $y_j = 1$, řešič to skutečně udělá.

Pokud by však některé t_j mohly být záporné, tento ILP by ještě nefungoval správně: řešič by mohl najít nepřípustné řešení, ve kterém sice všechny relevantní x_i budou 1, ale zároveň ponechá $y_j = 0$, protože nechce vzít záporné t_j. To mu musíme zakázat přidáním podmínky, která říká „pokud jsou všechny tyto $x_i = 1$, pak i y_j musí být 1“. Bez násobení toho dosáhneme následovně: $y_j \geq x_{u_{j,1}} + \cdots + x_{u_{j,\ell_j}} - (\ell_j - 1)$. Pokud je některé z použitých x_i nulové, je tato podmínka automaticky splněna. Pokud jsou naopak všechna rovna 1, dostáváme $y_j \geq \ell_j - (\ell_j - 1) = 1$, což je přesně to, co jsme chtěli.

Program (Python):

```
import pulp

def solve_one():
    # Načteme 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, neboť interně číslujeme 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 proměnné
    problem = pulp.LpProblem('Investice', 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, jehož 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) ))

    # Přidáme podmínky
    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

    # Pustíme řešič a vypíšeme výsledek
    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: tyče

V tomto úkolu řešíme různé varianty tzv. vícerozměrného problému batohu. Popíšeme si několik různých postupů, které bylo možné použít.

Pro první dva vstupy nám stačily jednoduché binární proměnné: Řekněme, že chceme rozhodnout, zda existuje řešení pro nějakou pevnou hodnotu t. Tyče zakoupené v obchodě očíslujeme od 0 do $t-1$, naše požadované tyče od 0 do $n-1$ a pro každou dvojici (i,j) budeme mít binární proměnnou $x_{i,j}$ udávající, zda z obchodní tyče i odříznout požadovanou tyč j. Podmínky našeho ILP jsou pak celkem přímočaré: pro každé j musí být některé $x_{i,j}$ rovno 1 (stačí podmínka, že jejich součet je alespoň 1) a pro každé i musí platit, že součet délek těch tyčí, které odřízneme z obchodní tyče i, nesmí překročit ℓ. Pak stačí ověřit, zda výsledný ILP má řešení. Optimální hodnotu t můžeme nalézt tak, že budeme postupně zkoušet $t = 1, 2, \ldots$.

Toto řešení můžeme dále zobecnit tak, že místo binárních proměnných použijeme nezáporná celá čísla: pro každou obchodní tyč i a každý *typ* požadovaných tyčí j budeme mít celočíselnou proměnnou $y_{i,j}$ udávající *kolik* kusů tyče j odřízneme z tyče i. Podmínky jsou přirozeným zobecněním těch uvedených výše: pro každé j musí být součet všech $x_{i,j}$ větší nebo roven požadovanému počtu p_j tyčí daného typu, a pro každé i musíme z i-té obchodní tyče vyrobit kusy s celkovou délkou nepřesahující ℓ. Takový program by měl úspěšně vyřešit první tři testovací vstupy.

Program (Python):

```
import pulp

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

    problem = pulp.LpProblem('Tyce', pulp.LpMaximize)
    kolik = [ [ 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 += kolik[i][j] >= 0
    for j in range(t):
        problem += sum( kolik[i][j] * delky[i] for i in range(n) ) <= l
    for i in range(n):
        problem += sum( kolik[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

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

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

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

V posledních dvou testovacích vstupech zjevně platí, že z každé obchodní tyče získáme jen několik málo požadovaných tyčí. Snadno si všimneme, že existuje nanejvýš několik tisíc možností, jakou sadu požadovaných tyčí můžeme z obchodní tyče vyrobit. Naše nová strategie proto bude následující: Vygenerujeme všechny tyto možnosti a pro každou z nich budeme mít jednu celočíselnou proměnnou udávající, kolik obchodních tyčí chceme rozřezat tímto konkrétním způsobem. V sadě 4 můžeme dokonce mít binární proměnné, protože se nemůže vyplatit rozřezat dvě obchodní tyče stejným způsobem, a navíc stačí generovat pouze taková rozřezání, ve kterých se žádná tyč neopakuje.

Jak tentokrát vypadají podmínky, kterými náš ILP zajistí, že dostaneme optimální platné řešení? Pro každý typ požadovaných tyčí nám stačí mít podmínku, že jejich celkový počet je dostatečný. A tentokrát místo pouhé kontroly, zda řešení existuje, použijeme řešič přímo k optimalizaci: zadáme mu za úkol minimalizovat součet všech proměnných – jinými slovy, celkový počet použitých tyčí.

Program (Python):

```
import pulp
import sys

def solve_one():
```

```
    l = int(input())
    n = int(input())
    delky = [ 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(delky[i])
                if pat.count(delky[i]) > pocty[i]: break
                if sum(pat) > l: break
                new_patterns.add( tuple(pat) )
        patterns = new_patterns

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

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

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

    problem += sum( kolik[i] for i in range(p) )
    for i in range(p):
        problem += kolik[i] >= 0
    for j in range(n):
        problem += sum( kolik[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'
    kolik = [ int(round( pulp.value( kolik[i] ) )) for i in range(p) ]

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

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