P-II-1 Reverze

Úloha má snadné řešení s kvadratickou časovou složitostí: třídění BubbleSort. Opakovaně procházíme polem a vždy, když najdeme vedle sebe stojící dvojici čísel v pořadí větší-menší, použitím jedné reverze je vyměníme. Takové řešení ale není optimální, v nejhorším případě potřebuje provést až kvadratický počet reverzí.

Lineární počet reverzí

Poměrně přímočaře lze navrhnout řešení s lineárním počtem reverzí. Základní myšlenka řešení je jednoduchá. Najdeme největší prvek v poli, nechť se nachází na pozici i. Jednou reverzí ho umíme dostat na pozici $n-1$, kam patří: jednoduše obrátíme celý úsek od i do $n-1$. Od tohoto okamžiku můžeme tento prvek nechat na svém místě a už si ho nevšímat. Dostali jsme tím původní problém, ale už jenom se zbývajícími $n-1$ prvky. Další postup řešení proto bude vypadat úplně stejně. Dokud neuspořádáme celé pole, najdeme vždy největší prvek v ještě neuspořádané části pole a jednou reverzí ho přesuneme na její konec.

Toto řešení provede v nejhorším případě $n-1$ reverzí. Jeho časová složitost je ovšem kvadratická – v každé iteraci potřebujeme nalézt maximum mezi ještě neuspořádanými čísly a na kopii pole si odsimulovat provedenou reverzi, abychom věděli, který prvek se kam přesunul.

Komprese souřadnic

Ukážeme si dvě jiná řešení úlohy, která také potřebují lineární počet reverzí, ale dokážou je provést efektivněji.

Obě naše lepší řešení budou mít společný první krok. Začneme tím, že si prvky zadaného pole uspořádáme a v pořadí od nejmenšího po největší je nahradíme čísly od 0 do $n-1$. Tím jsme si úlohu trochu zjednodušili: pro každé i nyní platí, že číslo i potřebujeme dostat na pozici i.

Uvedený první krok umíme provést v čase $\mathcal{O}(n \log n)$ – například tak, že si uspořádáme sadu záznamů tvaru „hodnota $A[i]$ je na pozici i“, nebo tak, že uspořádáme kopii pole ze vstupu a potom při přečíslování původního pole použijeme buď binární vyhledávání v uspořádané kopii, nebo vhodnou datovou strukturu.

Řešení založené na výměnách

Když jsme prvky původního pole nahradili čísly od 0 do $n-1$, dostali jsme tím *permutaci,* kterou označíme P. K ní můžeme v lineárním čase sestrojit *inverzní permutaci* Q předpisem $\forall i : Q[P[i]] = i$. (Permutace Q je pole, v němž máme ke každé hodnotě uloženou informaci, kde v poli P se tato hodnota nachází.)

Nyní už jen stačí uvědomit si, že pomocí dvou reverzí vyměníme libovolnou dvojici prvků v našem poli: chceme-li vyměnit $P[x]$ a $P[y]$ (kde $x < y$), provedeme

nejprve reverzi úseku od x do y a potom (když je třeba) reverzi úseku od $x+1$ do $y-1$.

Takovouto změnu v poli dokážeme odsimulovat v konstantním čase. Ještě důležitější je, že v konstantním čase ji umíme odsimulovat také v poli Q, takže nadále budeme vědět, na které pozici máme kterou hodnotu.

Toto nám už stačí na kompletní řešení. Nejprve v čase $\mathcal{O}(n \log n)$ změníme čísla v původním poli na čísla 0 až $n-1$, potom v lineárním čase sestrojíme inverzní permutaci, a na závěr postupně pro každé x od $n-1$ do 1 umístíme pomocí dvou reverzí číslo x na pozici x. Dohromady tak dostáváme řešení s $\mathcal{O}(n)$ reverzemi a časovou složitostí $\mathcal{O}(n \log n)$.

```
def najdi(usporadane_pole, prvek):
    # predpokladame, ze prvek se v poli nachazi
    # predstavime si, ze usporadane_pole[N] = nekonecno
    # invariant: usporadane_pole[lo] <= prvek < usporadane_pole[hi]
    lo, hi = 0, len(usporadane_pole)
    while hi - lo > 1:
        med = (lo + hi) // 2
        if usporadane_pole[med] <= prvek: lo = med
        else:                             hi = med
    return lo

# nacteme vstup
N = int( input() )
A = [ int(_) for _ in input().split() ]

# usporadame vstup a sestrojime permutace P a Q
B = sorted(A)
P = [ najdi(B,a) for a in A ]
Q = [ None for n in range(N) ]
for n in range(N): Q[ P[n] ] = n

# postupne provadime reverze, ktere spravne umisti prvky N-1 az 1
for x in reversed(range(1,N)):
    kde, kam = Q[x], x
    if kde == kam: continue
    print(kde, kam)
    if kam > kde+2:
        print(kde+1, kam-1)

    P[kde], P[kam] = P[kam], P[kde]
    Q[ P[kde] ] = kde
    Q[ P[kam] ] = kam
```

Řešení založené na posouvání

Ukážeme si ještě druhé, stejně dobré řešení, avšak založené na jiné myšlence. Přesněji řečeno, ukážeme si, jak lze vylepšit a pomocí vhodných datových struktur zrychlit naše první řešení, které používalo $n-1$ reverzí, ale potřebovalo kvadratický čas. Tentokrát namísto výměny dvou prvků budeme pomocí dvou reverzí simulovat posunutí nějakého prvku.

Když máme číslo $n-1$ na pozici $i < n-1$, provedeme postupně dvě reverze: jednu na úseku od i do $n-1$ a potom druhou na úseku od i do $n-2$. Druhou

reverzí vrátíme všechny prvky v daném úseku do jejich původního pořadí. Výsledný efekt těchto dvou reverzí je stejný, jako kdybychom největší číslo vybrali z pole a přesunuli ho na konec. Všechna čísla, která byla nalevo od něho, zůstala na svých původních pozicích, zatímco všechna ostatní čísla jsou nyní v poli posunuta o jednu pozici doleva.

Jak bude řešení pokračovat? Předpokládejme, že už jsme několik největších čísel přesunuli na jejich správná místa na konci pole. V následujícím kroku bychom chtěli totéž provést s číslem x. K tomu ale potřebujeme vědět, kde se nyní nachází.

Víme, kde se x nacházelo na začátku. Po každé dvojici reverzí buď zůstalo na místě, nebo se posunulo o jednu pozici doleva. Kolikrát se posunulo doleva? Tolikrát, kolik větších čísel bylo v původním poli nalevo od x. Když budeme znát tuto hodnotu, dokážeme si vypočítat, kde se x nachází nyní, a tedy jaké reverze potřebujeme provést, abychom ho dostali na správné místo.

Celé řešení můžeme shrnout následovně:

1. Přečíslujeme prvky pole na 0 až $n-1$. Pro každé x si zapamatujeme, na které pozici se nyní nachází.
2. Pro každé x zjistíme, kolik větších čísel je nalevo od něj.
3. Postupně od $x = n-1$ po $x = 1$ umístíme číslo x na správné místo.

Už víme, že krok 1 můžeme provést v čase $\mathcal{O}(n \log n)$ a krok 3 dokonce v lineárním čase. Zbývá doplnit detaily o kroku 2.

Existuje více způsobů, jak provést krok 2 v čase $\mathcal{O}(n \log n)$. Jednou možností je pole jednoduše projít zleva doprava, přičemž si již zpracované prvky pamatujeme ve vhodném vyvažovaném binárním stromě. Po přečtení každého čísla pomocí stromu v logaritmickém čase zjistíme, kolik dříve zpracovaných čísel bylo větších. Jiným řešením (bez komplikovaných datových struktur) je představit si, že začínáme s prázdným polem a postupně se v něm objevují čísla v pořadí od největšího po nejmenší. Nad polem si postavíme intervalový strom, v němž si budeme pro každý úsek pamatovat, kolik čísel se v něm už objevilo. S každým novým číslem dokážeme v logaritmickém čase zjistit počet dříve objevených čísel nalevo od něho a následně upravit ty vrcholy intervalového stromu, které přidání nového prvku změnilo. Třetím řešením je upravit třídicí algoritmus MergeSort. Detaily tohoto řešení přenecháváme jako cvičení.

Bonus: menší než lineární počet reverzí nestačí

Ve vašich řešeních jsme toto pozorování nevyžadovali, ale pro úplnost vzorového řešení uvedeme ještě stručný důkaz toho, že řešení s menším než lineárním počtem reverzí nemůže existovat.

Když se podíváme na jakékoliv pole, dokážeme pro každý jeho prvek určit, které (nejvýše) dva prvky s ním budou v uspořádaném poli sousedit. Ve správně uspořádaném poli nutně musí mít každý prvek správné sousedy.

Je ovšem možné, že na začátku vůbec nikdo žádného správného souseda nemá – viz například pole $(0, 2, 4, \ldots, 1, 3, 5, \ldots)$. Při každé reverzi se změní sousedé nejvýše

čtyřem prvkům – těm na obou krajích obraceného úseku, a těm, kteří s obraceným úsekem sousedí. Když tedy potřebujeme každému prvku změnit sousedy, musíme nutně provést alespoň lineární počet reverzí.

P-II-2 Potrubí

Řešení hrubou silou můžeme založit na myšlence, že na pořadí trubek nezáleží. Stačí tedy vyzkoušet všechny *podmnožiny* trubek a pro každou spočítat, zda má potrubí správnou délku a dostatečný počet filtrů.

Obě efektivnější řešení, která si ukážeme, budou mít podobnou hlavní myšlenku. Začneme tím, že si všechny trubky uspořádáme podle průtoku od největšího po nejmenší. V tomto pořadí budeme trubky jednu po druhé zpracovávat, dokud se nám poprvé nepodaří z již zpracovaných trubek postavit vyhovující potrubí. V tom okamžiku můžeme přestat a jako odpověď vypíšeme průtok poslední přidané trubky.

Řešení za 7 bodů: Budeme si postupně generovat všechny kombinace (délka potrubí, počet filtrů), které se dají z dané sady trubek postavit. Když přidáme novou trubku, zůstanou nám všechny staré možnosti a přibudou k nim nějaké možnosti nové. Ty získáme tak, že ke každé staré možnosti zkusíme přidat novou trubku.

Samozřejmě nemá smysl pamatovat si možnosti, které jsou příliš dlouhé. Více než f filtrů je pro nás už totéž jako přesně f filtrů. Všech možností, které můžeme sestrojit, je proto $\mathcal{O}(\ell f)$. Pro každou z n trubek jednou projdeme množinu aktuálních možností. Řešení má tudíž časovou složitost $\mathcal{O}(n\ell f)$ a paměťovou složitost $\mathcal{O}(\ell f)$.

Na *řešení za plný počet bodů* potřebujeme už jenom jedno pozorování navíc: když umíme vytvořit potrubí délky d s třemi filtry a jiným způsobem vytvoříme potrubí stejné délky s pěti filtry, stačí si pamatovat tento druhý způsob. Kdybychom totiž v budoucnu pomocí přidaných trubek postavili nějaké platné řešení z prvního potrubí, můžeme namísto něho použít druhé potrubí a také dostaneme platné řešení (stejné délky a s více filtry).

Obecně tedy platí, že pro každou délku potrubí si potřebujeme pamatovat pouze jedno číslo: největší počet filtrů, s nimiž dokážeme této přesné délky dosáhnout. (Jestliže nějakou délku potrubí ještě neumíme získat, můžeme si to označit tím, že maximální počet filtrů bude $-\infty$.)

V libovolném okamžiku si tedy potřebujeme pamatovat jenom $\mathcal{O}(\ell)$ čísel. Taková je i paměťová složitost našeho řešení. Časová složitost je $\mathcal{O}(n\ell)$, neboť seznam $\mathcal{O}(\ell)$ možností budeme n-krát procházet.

```
from sys import exit

L, F = [ int(_) for _ in input().split() ]
N = int( input() )
trubky = []
for n in range(N):
    delka, prutok, filtr = input().split()
    trubky.append( ( int(prutok), int(delka), 1 if filtr=='A' else 0 ) )

NELZE = -(2**30)
nejvic_filtru = [ NELZE for _ in range(L+1) ]
nejvic_filtru[0] = 0
```

```
for prutok, delka, filtr in reversed(sorted(trubky)):
    for dosud in reversed(range(0,L-delka+1)):
        if nejvic_filtru[dosud] != NELZE:
            nejvic_filtru[dosud+delka] = max( nejvic_filtru[dosud+delka],
                                              nejvic_filtru[dosud]+filtr )
    if nejvic_filtru[L] >= F:
        print(prutok)
        exit()

print(-1)
```

P-II-3 Virus

Je zjevné, že virus se nemůže rozšířit z jedné komponenty souvislosti počítačové sítě do druhé. Občas se ale stane, že virus nedokáže nakazit ani celý souvislý kus sítě. Máme-li například čtyři počítače zapojené do kruhu, ať bychom nakazili kterýkoliv z nich, ten nakazí pouze protilehlý počítač a tím šíření viru skončí. Dva ze čtyř počítačů vždy zůstanou nenakažené.

Počítač x může nakazit počítač y právě tehdy, když se z x do y dá dojít sudým počtem kroků (v každém kroku přejdeme z nějakého počítače na sousední). Jak ale máme poznat, kdy se takto dají postupně nakazit všechny počítače v síti a kdy ne?

Tvrzení: Máme-li souvislou počítačovou síť tvořenou více než jedním počítačem, virus ji dokáže celou nakazit právě tehdy, když síť obsahuje cyklus liché délky – tedy jestliže existuje posloupnost počítačů $p_0, \ldots, p_{2k}$ taková, že každý p_i sousedí s p_{i+1} a poslední sousedí s prvním.

Důkaz první implikace: Mějme souvislou síť obsahující cyklus liché délky. Chceme ukázat, že po nakažení libovolného počítače budou časem nakažené všechny.

Protože síť je souvislá, od libovolného počítače se po kabelech dostaneme k tomuto cyklu (a naopak). Předpokládejme, že jsme jako první nakazili nějaký počítač x a že y je nějaký jiný počítač v této síti. Uvažujme dvě různé cesty z x do y. Možnost 1: přejdeme z x k p_0 (konkrétní počítač v cyklu liché délky) a odtud k y. Možnost 2: totéž jako možnost 1, ale po příchodu do p_0 nejprve jednou obejdeme dokola celý cyklus a až potom pokračujeme dále.

Jelikož cyklus má lichou délku, počet kroků v možnosti 2 má opačnou paritu než počet kroků v možnosti 1. Jedna z těchto dvou možností proto určitě představuje způsob, jak se z x dostat do y sudým počtem kroků, takže počítač y bude časem nakažený.

Důkaz druhé implikace: Dokážeme, že pokud v naší souvislé síti (tvořené alespoň dvěma počítači) není žádný cyklus liché délky, pak ji určitě nedokážeme celou nakazit.

Představme si, že jsme z libovolného počítače s v naší síti spustili prohledávání do šířky a pro každý jiný počítač jsme tak zjistili jeho minimální vzdálenost od s. Obarvíme nyní počítače podle této vzdálenosti: počítače se sudou vzdáleností (včetně samotného s) budou bílé, ostatní počítače budou černé.

Navíc obarvíme také kabely: Pro každý počítač různý od s obarvíme zeleně kabel, kterým jsme do něj poprvé přišli. Všechny ostatní kabely budou červené.

Tvrdíme, že v naší síti nemůže existovat dvojice sousedních počítačů stejné barvy. Každý zelený kabel zjevně spojuje černý a bílý počítač. Kdybychom měli červený kabel, který spojuje dva počítače stejné barvy, potom zelené kabely tvořící cestu mezi těmito dvěma počítači a tento červený kabel by dohromady vytvořily cyklus liché délky.

Nyní je už jasné, že když v takovéto síti nakazíme bílý počítač, nikdy se od něho nemůže nakazit žádný černý počítač, a naopak.

Tím se dostáváme k celkovému řešení naší úlohy. Načteme vstupní data, rozdělíme si graf na komponenty souvislosti a zkontrolujeme, zda některá obsahuje cyklus liché délky (algoritmus jsme popsali v důkazu druhé implikace).

Máme-li k komponent souvislosti, určitě potřebujeme přidat alespoň $k-1$ kabelů. Potřebujeme totiž, aby všechno bylo propojené, a každým kabelem dokážeme snížit počet komponent nejvýše o 1.

Pokud některá komponenta souvislosti obsahuje lichý cyklus, přesně $k-1$ kabelů stačí. Jedním konkrétním způsobem, jak je možné kabely přidat, je zvolit si v každé komponentě jeden počítač a propojit jeden z nich se všemi ostatními.

Jestliže žádná komponenta souvislosti neobsahuje lichý cyklus, budeme potřebovat přesně k kabelů. Ukážeme, že $k-1$ kabelů nestačí, ale že k už ano.

Představme si, že v každé komponentě máme počítače obarvené černě a bíle. Kdyby stačilo $k-1$ kabelů, musí každý z nich spojovat dvě komponenty, které dosud spojeny nebyly. V tom případě ale vždy můžeme upravit obarvení počítačů tak, aby nadále platilo, že každý kabel spojuje počítače různých barev. (Jestliže nový kabel spojí počítače různých barev, nemusíme dělat nic. Jestliže chceme spojit počítače stejné barvy, nejprve celou jednu komponentu přebarvíme na opačné barvy a až potom přidáme nový kabel.) Potom ale i po přidání všech $k-1$ kabelů budeme mít počítače obarvené na černé a bílé a stále ještě nebudeme mít žádný cyklus liché délky.

V popsané situaci snadno dokončíme řešení přidáním k-tého kabelu: stačí propojit libovolné dva počítače stejné barvy. Dva takové určitě najdeme mezi počítači s čísly 0, 1 a 2.

Při dobré implementaci má celé řešení časovou složitost $\mathcal{O}(n+m)$, tedy lineární vzhledem k velikosti vstupu. Pro pořádek ještě dodáváme, že v níže uvedené implementaci jsme na obarvení komponent použili prohledávání do hloubky, nikoliv do šířky. Rozmyslete si, že to nevadí.

```
#include <bits/stdc++.h>
using namespace std;

int N, M;
vector< vector<int> > sousedi;
vector<int> barva;
vector<int> reprezentanti;

bool obarvi_komponentu(int v, int f) {
    // Obarvi všechny vrcholy komponenty střídavě černou a bílou,
```

```
    // přičemž když komponenta nemá lichý cyklus, stejné barvy nikdy
    // nebudou sousedit. Vrátí true, pokud najde lichý cyklus.
    barva[v] = f;
    bool odpoved = false;
    for (int w : sousedi[v]) {
        if (barva[w] == f)   { odpoved = true; continue; }
        if (barva[w] == 1-f) {                 continue; }
        obarvi_komponentu(w,1-f);
    }
    return odpoved;
}

void spoj_dva_stejne() {
    for (int a=0; a<3; ++a)
        for (int b=0; b<a; ++b)
            if (barva[a] == barva[b]) {
                cout << a << " " << b << endl;
                return;
            }
}

int main() {
    // Načteme vstup
    cin >> N >> M;
    sousedi.resize(N);
    for (int m=0; m<M; ++m) {
        int x, y;
        cin >> x >> y;
        sousedi[x].push_back(y);
        sousedi[y].push_back(x);
    }

    // Obarvíme komponenty tak, že první začneme barvou 0 a každou
    // další barvou 1.
    barva.resize(N, -1);
    reprezentanti.push_back(0);
    bool mame_lichy_cyklus = obarvi_komponentu(0,0);
    for (int n=1; n<N; ++n) if (barva[n] == -1) {
        reprezentanti.push_back(n);
        mame_lichy_cyklus |= obarvi_komponentu(n,1);
    }

    // Spojíme komponenty, pokud máme více než jednu
    for (unsigned i=1; i<reprezentanti.size(); ++i)
        cout << reprezentanti[0] << " " << reprezentanti[i] << endl;

    // Pokud nemáme lichý cyklus, je třeba ještě spojit
    // libovolné dva vrcholy stejné barvy
    if (!mame_lichy_cyklus)
        spoj_dva_stejne();

    // Je jedno, který počítač nakazíme jako první
    cout << 0 << endl;
}
```

P-II-4 Exaktní exponenciální algoritmy

Podúloha A: pořadí úkolů

Začneme nejprve několika poměrně zjevnými pozorováními.

Nikdy se nevyplatí odpočívat a nic nedělat, když ještě nemáme všechny úkoly hotové. Pokud bychom totiž odpočinek vynechali, všechny později splněné úkoly by byly dokončeny o něco dříve, takže i pokuta za ně by byla nižší.

Nikdy se nevyplatí střídavě pracovat na dvou nebo více úkolech. Vždy existuje optimální řešení, v němž vždy nejprve zcela dokončíme jeden úkol a až potom jdeme na další. Když se totiž podíváme na jakékoliv řešení a vypíšeme si, v jakém pořadí jsme skončili práci na jednotlivých úkolech, a potom se podíváme na řešení, které tytéž úkoly vykoná jeden po druhém ve stejném pořadí, snadno nahlédneme, že toto druhé řešení každý úkol dokončí buď ve stejném čase nebo dříve oproti prvnímu řešení.

Když tedy víme, že stačí hledat řešení, v němž úkoly plníme jeden po druhém a bez přestávek, hledáme vlastně *permutaci* úkolů, které odpovídá nejmenší možný celkový součet pokut.

Algoritmus s časovou složitostí $\hat{\mathcal{O}}(2^n)$ pro tento problém vypadá například následovně: Použijeme dynamické programování, přičemž úlohu postupně vyřešíme pro každou podmnožinu úkolů.

Pro libovolnou podmnožinu X naší množiny úkolů označme $B(X)$ nejmenší součet pokut, které bychom dostali, kdybychom plnili pouze úkoly z množiny X.

Víme, že když budeme plnit tyto úkoly, poslední z nich dokončíme v čase $t_X = \sum_{x \in X} t_x$. Musíme se rozhodnout, který z našich úkolů bude tím, co dokončíme jako poslední. Toto rozhodnutí uděláme snadno: vyzkoušíme všechny možnosti a vybereme nejlepší z nich. Pokud jako poslední vykonáme úkol x, pak v optimálním řešení nejprve optimálně splníme ostatní úkoly – což nám přinese celkovou pokutu $B(X-\{x\})$, plus součet všech $s_{y,x}$ za úkoly dokončené dříve než x – a potom splníme úkol x. Platí tedy:

$$B(X) = \min_{x \in X} \left(p(x, t_X) \;+\; B(X - \{x\}) \;+\; \sum_{y \in X - \{x\}} s_{y,x} \right).$$

Pomocí tohoto vztahu postupně pro každou z 2^n podmnožin naší množiny úkolů spočítáme v čase $\mathcal{O}(n)$ její optimální řešení.

(Na závěr doplníme, že existuje také řešení se stejnou časovou složitostí, v němž zkoušíme, který úkol vykonáme jako *první,* nikoliv *poslední.* V tomto řešení ale musíme místo $B(X)$ uvažovat jiné hodnoty: $C(X)$ je optimální suma pokut za povinnosti z množiny X, ale za předpokladu, že začínáme ne v čase 0, ale v čase, když jsme dokončili všechny ostatní povinnosti.)

Podúloha B: vrcholové pokrytí grafu

Klíčovým pozorováním při řešení této úlohy je, že libovolné *vrcholové pokrytí* (platné rozmístění záchodů ze zadání úlohy) je vždy doplňkem nějaké *nezávislé množiny* (zvířátka ze studijního textu) a naopak. Totiž:

1. Máme-li nějakou nezávislou množinu lokalit X ve městě, pak žádná ulice nemá oba své konce v X (neboť žádné dvě lokality v X nejsou přímo propojené). Jestliže tedy postavíme záchody všude kromě X, bude jistě na každé ulici alespoň jeden záchod – tudíž doplněk X tvoří vrcholové pokrytí.
2. Máme-li nějaké vrcholové pokrytí Y, pak každá ulice má v Y alespoň jeden svůj konec, takže mimo Y má nevýše jeden svůj konec. V doplňku Y proto nejsou žádné dvě lokality, které by byly propojeny ulicí, takže doplněk Y je nezávislá množina.

Velikost nejmenšího vrcholového pokrytí tedy můžeme určit tak, že najdeme $i =$ velikost největší nezávislé množiny a následně spočítáme a vrátíme hodnotu $n - i$.

Nejlepším algoritmem, s nímž jsme se v rámci našeho seriálu seznámili, je algoritmus z řešení domácího kola. Jeho časová složitost je $\hat{\mathcal{O}}(1{,}3803^n)$.

V současné době jsou známé i lepší algoritmy řešící tuto úlohu, například Xiaův-Nagamochiho algoritmus z roku 2013 má časovou složitost dokonce jenom $\hat{\mathcal{O}}(1{,}1996^n)$. Tyto algoritmy však svou náročností přesahují rámec našeho seriálu úloh a na plný počet bodů o nich nemusíte ani vědět.