Korešpondenčný seminár z programovania
XXXIV. ročník, 2016/17
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Návody k úlohám 1. kola letnej časti kategórie T

V kategórii T neuvádzame vzorové riešenia ale skôr návody na riešenie úloh. Ich cieľom je prezradiť vám hlavnú myšlienku riešenia, aby ste podľa návodu mohli riešenie domyslieť sami. Občas teda vynecháme niektoré drobné detaily, neuvádzame implementácie dátových štruktúr a nerozpisujeme niektoré kroky.

Neuvádzame ani vzorové programy, pretože chceme, aby ste po prečítaní návodu naprogramovali tie úlohy, ktoré ste nespravili počas trvania série. Keď si tieto riešenia sami naprogramujete, naučíte sa tým oveľa viac ako pozeraním sa na zdrojové kódy.

Implementácie všeobecne známych algoritmov a dátových štruktúr môžete nájsť vo vzorových riešeniach kategórií Z a O starších ročníkov KSP alebo aj na internete.

1. Trikrát bola párty

návod písal mišof
(max. 0 b za popis, 20 b za program)

Úloha má viac možných riešení, ukážeme si to, ktoré sa asi najľahšie implementuje.

Dvaja ľudia sú ekvivalentní, ak boli na tej istej množine párty. Ak máme vo vstupe dvoch ekvivalentných ľudí, napríklad Maru a Usamca, môžeme sa jedného z nich zbaviť tak, že do výstupu pridáme implikácie "ak Maru tak Usamec" a "ak Usamec tak Maru".

Keďže boli len tri párty, existuje len 8 rôznych typov ľudí. (Jeden z nich sú napr. ľudia, ktorí boli len na prvej párty, čiže tí, čo majú na vstupe stĺpec `100`.) Keď sme sa zbavili ekvivalentných ľudí, ostal nám teda vstup, v ktorom je nanajvýš osem ľudí.

Už v tomto okamihu by sa dala na doriešenie úlohy vhodne použiť hrubá sila, ale porozmýšľajme ešte.

Ak máme človeka Quasimoda, ktorý nebol na žiadnej párty, zbavíme sa ho pridaním podmienky "ak Quasimodo tak nie Quasimodo". A naopak, ak máme jeho opak Esmeraldu, tej sa zbavíme podmienkou "ak nie Esmeralda tak Esmeralda". A už nám ostalo ľudí len šesť.

Dvaja ľudia sú opační, ak bol na každej párty práve jeden z nich. Ak máme vo vstupe dvoch opačných ľudí, tiež sa vieme jedného zbaviť. Ak sú napríklad Jekyll a Hyde opační, stačí pridať implikácie "ak Jekyll tak nie Hyde" a "ak Hyde tak nie Jekyll".

A keď sme sa zbavili aj opačných ľudí, ostal nám vstup s nanajvýš troma ľuďmi.

To isté si teraz môžeme povedať aj matematicky. Máme n boolovských premenných a snažíme sa napísať logický výraz, ktorý bude splnený pre práve tri kombinácie hodnôt premenných, a to pre tie tri predpísané na vstupe. Vyššie uvedeným postupom sme zostrojili časť tohto výrazu: zbavili sme sa konštánt, ekvivalentných premenných, aj premenných, ktoré sú negáciou jedna druhej. Skončili sme v situácii, že máme nanajvýš tri *voľné premenné*. Keď zvolíme ich pravdivostné hodnoty, nami zostrojená sada implikácií jednoznačne určí hodnoty všetkých ostatných premenných. A ľahko nahliadneme, že všetky tri ohodnotenia zo vstupu sa nachádzajú medzi tými ohodnoteniami, ktoré takto vieme dostať.

Sú len dve možnosti: buď máme voľné premenné presne dve, alebo presne tri. (Rozumiete, prečo sa nemôže stať, že by bola len jedna?)

Ak sú presne dve, je doteraz zostrojený výraz pravdivý pre štyri rôzne ohodnotenia premenných, a my z nich potrebujeme ešte zakázať to jedno, ktoré nechceme. Toto sa vždy dá dosiahnuť. Predstavme si napríklad, že tie dve voľné premenné sú ľudia Rasťo a Marína, ktorým na vstupe zodpovedajú stĺpce `101` a `001`. Potom potrebujeme vylúčiť možnosť, v ktorej má Rasťo `0` a Marína `1`, čiže možnosť, že Rasťo na párty nebol a Marína áno. Toto spravíme pridaním implikácie "ak Marína tak Rasťo". Analogicky vieme postupovať pre ľubovoľné dve voľné premenné.

Ak sú voľné premenné presne tri, má doteraz zostrojená sada implikácií osem vyhovujúcich ohodnotení premenných. Spomedzi týchto by sme chceli tie tri dobré nechať a zvyšných päť zlých zakázať. Dá sa ale dokázať (rozmyslite si, ako), že toto nikdy nejde spraviť. Ak teda po spravení všetkých redukcií počtu premenných vidíme tri voľné premenné, úloha nemá riešenie.

2. Totálny zabiják

návod písal Samo
(max. 0 b za popis, 20 b za program)

Najprv vyriešime ľahšiu úlohu (aj keď ťažko predstaviteľnú) a to takú, že budeme predpokladať, že Emko až

taký hlúpy nie je. To znamená, že nie je ochotný zaplatiť za objednávku niektorého z nepriateľov, teda všetky čísla na vstupe budú nezáporné.

Hlavná myšlienka

Keďže chceme vypisovať hodnoty možných účtov od najmenšieho, najrozumnejšie je všetkých kandidátov na najlacnejší účet nahádzať do minimovej haldy a potom odtiaľ vyberať vždy najlacnejší. Kandidátov ale máme všetky podmnožiny, čo je teda až 2^n prvkov v halde. Stačí si však uvedomiť, že kandidátmi na najlacnejší (ešte nevypísaný účet) nie sú od začiatku všetky podmnožiny, ale sa nimi stanú až neskôr.

Kandidáti

Vieme napríklad povedať, že ak A je podmnožinou B, tak A vypíšeme určite skôr, lebo B získame z A pridaním niekoľko *nezáporných* prvkov. Takže množina sa stane kandidátom až keď vypíšeme všetky jej podmnožiny, a dovtedy ju do haldy vôbec nemusíme pridávať. Čo teda spôsobí, že v halde sa vyskytne výrazne menej vecí, keďže k je relatívne malé.

Vyššie uvedená úvaha sa ale dá potiahnuť ešte ďalej. Vieme si vybudovať graf závislostí, kde vrcholy sú možné účty a orientovaná hrana z A do B bude znamenať, že B pridáme do haldy (stane sa kandidátom) keď vypíšeme A.

Od takéhoto grafu budeme požadovať viacero vecí. Za prvé si musíme byť istý, že A je lacnejší ako B, teda že sa od nás nečakalo, že vypíšeme B a B sme ešte nemali ani v halde. A zároveň nechceme ten istý účet pridávať do haldy viackrát, teda do jedného vrchola môže smerovať najviac jedna hrana. Nakoniec musí byť každý vrchol dosiahnuteľný z vrcholu reprezentujúceho najlacnejší účet.

Rozmyslite si, prečo sú horeuvedené tri podmienky nutné a zároveň postačujúce na to, aby náš algoritmus (k-krát vyber najlacnejšieho kandidáta z haldy a pridaj do nej nových kandidátov) fungoval správne.

Ako rozumne zvoliť graf?

Ak máme vrchol reprezentujúci podmnožinu $\{a_1, a_2, ..., a_n\}$, tak budú z neho viesť hrany do vrcholov reprezentujúcich množiny $\{a_1, a_2, ..., a_n, b\}$ a $\{a_1, a_2, ..., a_{n-1}, b\}$, kde b označuje najmenšie číslo väčšie ako a_n.

Zjavne platí, že do každého vrchola vedie len jedna hrana. A tiež hrany vedú smerom od lacnejších účtov ku drahším, keďže prvý typ hrany vedie ku nadmnožine a pri druhom type vymeníme a_n za b, čo je aspoň tak veľké číslo. A nakoniec, na príklade ilustrujeme, akým spôsobom vieme dosiahnuť každý vrchol (a všeobecný prípad nechávame na rozmyslenie): ak sú všetky čísla $1, 2, 3, 4, 5, 6, 7, 8$, tak napríklad vrchol $\{1, 3, 4, 7\}$ dosiahneme takto:

$$\{1\} \to \{1,2\} \to \{1,3\} \to \{1,3,4\} \to \{1,3,4,5\} \to \{1,3,4,6\} \to \{1,3,4,7\}$$

Riešenie našej podúlohy

To čo teraz vieme spraviť je, že na začiatku pridáme do haldy najlacnejší prvok, a následne k-krát vyberieme z haldy najlacnejší prvok, vypíšeme ho, a pridáme 2 nasledujúcich kandidátov do haldy. Na to aby sme o každom vrchole vedeli povedať, kam smerujú jeho hrany a vedeli vypísať jeho hodnotu, nám stačí si pamätať súčet jeho členov a najväčší prvok.

Riešenie ak je Emko hlúpy

Toto vieme upraviť na úlohu, ktorú sme už vyriešili. Stačí ak spustíme náš algoritmus na absolútnych hodnotách a odpočítali vždy súčet záporných čísel na vstupe. Prečo to funguje? Totiž ak sa rozhodneme neakú podmnožinu vypísať, tak vlastne sa rozhodneme vypísať túto podmnožinu s tým, že tam pridáme všetky záporné čísla okrem tých ktoré sme už vybrali (v absolútnej hodnote) a teda sa nám akoby vynulujú.

Prvýkrát, keď by sme mali vypísať nulový účet, ho na výstup nedáme (lebo neprázdny účet nás nezaujíma).

Čo sa týka časovej zložitosti, tak potrebujeme načítať vstup a v k-kolách vždy jeden prvok odstrániť a dva pridať do haldy, teda $O(n + k \log k)$.

3. Triviálna interaktívna úloha

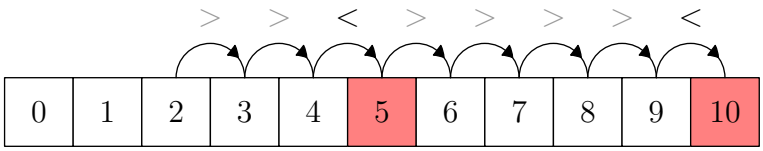
návod písal Buj
(max. 0 b za popis, 20 b za program)

Skúsme na začiatok iba vyriešiť úlohu, nezáleží na tom, ako efektívne. Keď budeme klásť iba otázky 1, budeme podľa odpovede vedieť presne povedať, čo sa stalo:

1. Ak sme dostali $>$, tak sa stav zvýšil o 1.

2. Ak sme dostali čokoľvek iné, tak sme *pretiekli* – stav dosiahol po pripočítaní 1 hodnotu m, a keď sme zobrali jeho zvyšok po delení m, dostali sme stav 0.

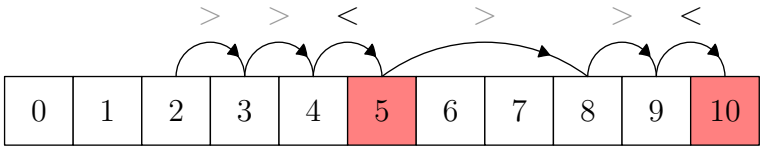
Nápad je taký, že ak by sme začínali v stave 0, budeme sa pýtať stále otázky 1, až kým nedostaneme = alebo $<$. Počet položených otázok je potom presne m. Do stavu 0 sa ale vieme dostať rovnakým spôsobom – pýtaním sa 1 dovtedy, kým nedostaneme niečo iné ako $>$. Máme tak algoritmus, ktorý sa opýta najviac $2n$ otázok.



Dolný odhad

Všimnime si, že ak sa k-krát spýtame otázku 1, a na každú dostaneme odpoveď $>$, tak určite platí $m \geq k+1$. Toto vieme využiť na zrýchlenie predchádzajúceho algoritmu:

Ak v prvej časti algoritmu použijeme k otázok, tak sa v druhej časti nemusíme na začiatku $(k-1)$-krát spýtať 1. Vieme, že na každú z tých otázok dostaneme $>$ – nedostaneme žiadnu informáciu. Namiesto toho, aby sme sa k-krát spýtali 1, sa raz spýtajme k. Máme tak algoritmus pýtajúci sa najviac $n+1$ otázok.



Načo nám je dolný odhad?

Pozrime sa na to, čo nám hovorí odpoveď na otázku a. Dostaneme $>$ ak $x + a < m$, v opačnom prípade môžeme dostať $>$, = aj $<$. Takže vo všeobecnosti to nevyzerá byť veľmi užitočné. Vieme, že ak dostaneme = alebo $<$, tak sme pretiekli. Ak sme ale dostali $>$, tak sme mohli ale nemuseli pretiecť.

Ako uvidíme, pomocou dolného odhadu budeme vedieť klásť také otázky a, že budeme **vedieť povedať, či sme pretiekli** alebo nie.

Jeden špeciálny prípad sme už videli, a to $a = 1$. (Zrejme $m \geq 1$, teda 1 naozaj je dolný odhad na m.) Pre nový stav $x + 1 \mod m$ totiž platí $x < x + 1 \leq x + m$, a ak sme pretiekli, tak máme $x + 1 \geq m$. Dokopy $m \leq x + 1 \leq x + m$, takže nový stav je rovný $x + 1 - m$, čo je nanajvýš x. Nemôžeme preto dostať pri pretečení $>$.

Namiesto $a = 1$ sme mohli ale uvažovať ľubovoľné $a \leq m$, a úvahy by prešli rovnako. Voľbu takýchto a nám umožňuje práve dolný odhad na m: ak l je dolný odhad na m, tak môžeme bezpečne zvoliť $a \leq l$.

Načo nám je schopnosť rozlíšiť pretečenie?

Ak sme pretiekli po položení otázky a, vieme, že sme v stave nanajvýš $a-1$. Keď sa ďalej spýtame otázky so súčtom s, tak sa náš stav zvýši nanajvýš o toľko, budeme teda v stave nanajvýš $\leq s + a - 1$. Takže za určitých okolností vieme zhora obmedziť náš stav.

Predpokladajme ďalej, že sme sa spýtali otázky so súčtom s, pričom sme pretiekli iba pri poslednej otázke. Ak by $m > s + a - 1$, tak by sme nepretiekli – tento prípad preto nemohol nastať, a preto $m \leq s + a - 1$. Takže schopnosť rozlíšiť pretečenie nám umožňuje zhora odhadnúť m.

Takisto nám ale umožňuje m odhadnúť zdola. Keď sa spýtame otázky so súčtom s, a pri ani jednej nepretečieme, tak nutne $m > s$.

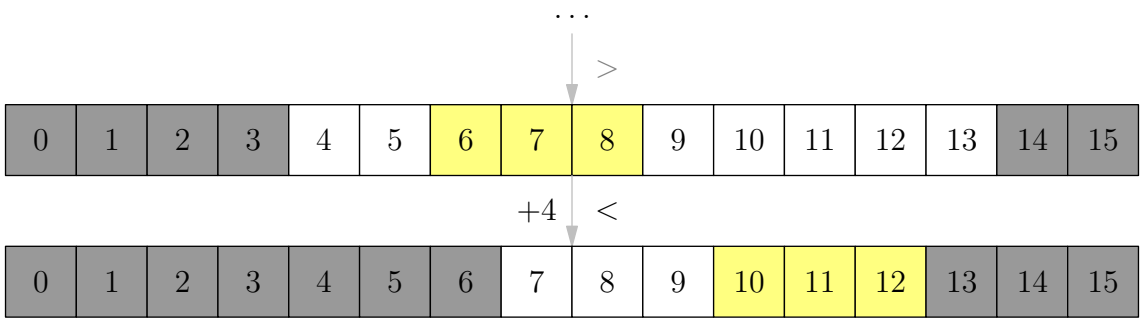
Zistime modulus

Videli sme, že ak vieme rozlíšiť pretečenie, tak vieme pomocou otázok zhora aj zdola obmedzovať m. To bude hlavná myšlienka nášho algoritmu.

Nech sme na začiatku v niektorom zo stavov $\{0, 1, \ldots, a-1\}$, a nech $l < m \leq r$: máme teda dolné aj horné obmedzenie na m. (Napríklad $l = 0$ a $r = n$.)

Budeme postupovať v *kolách*. V každom kole je naším cieľom zmenšiť interval, v ktorom je m, dvakrát – ak by sme to dokázali, počet kôl bude len $O(\log n)$. A ak by sme sa v každom kole spýtali iba konštantne veľa otázok, tak máme vyhraté.

V každom kole sa budeme pýtať dokola nejakú otázku b, až kým nepretečieme. Ak sme sa do pretečenia spýtali k otázok, tak máme dolný odhad $m > kb$, a horný odhad $m \leq (k+1)b + a - 1$. Okliešťime tak m v intervale dĺžky najviac $b + a - 1$.



Všimnime si ale, že v každom kole spravíme aspoň $\lceil \frac{l-a+1}{b} \rceil$ otázok – čo nie je zhora obmedzené konštantou, nakoľko l je rádovo n. Tento problém ale vyriešime známym trikom: rovno sa na začiatku spýtame $l-a+1$, čím sa "presunieme tesne pred l". (Konkrétne náš stav bude v $\{l-a+1, \ldots, l\}$.) Teraz sa v jednom kole spýtame rádovo $(\frac{r-l}{b})$-krát, takže chceme, aby b bolo rádovo $r-l$. Napríklad nech $b = \lceil \frac{r-l}{4} \rceil$.

Náš interval sa má zmenšiť dvakrát, teda má platiť $b + a - 1 \leq \lceil \frac{r-l}{2} \rceil$. Chceli by sme preto, aby platilo $a \leq \lceil \frac{r-l}{4} \rceil$. To vieme zaručiť tak, že pred každým kolom budeme mať *predkolo*, v ktorom sa budeme pýtať $\lceil \frac{r-l}{4} \rceil$ dovtedy, dokým nepretečieme. Po pretečení budeme v stave $\leq \lceil \frac{r-l}{4} \rceil - 1$, a teda $a = \lceil \frac{r-l}{4} \rceil$. Môže sa ale stať, že sa v predkole spýtame priveľa (nie konštantne veľa) otázok, čo vyriešime tak, že sa na začiatku predkola presunieme tesne pred l.

Zhrnutie

Našli sme teda vhodné konštanty a, b také, že sa náš algoritmus spýta najviac $O(\log n)$ otázok a zistí m. Pri tom sme využívali schopnosť rozlíšiť pretečenie – ako sme ale videli, vo všeobecnosti to nevieme robiť pre každú otázku. V skutočnosti sa algoritmus smie pýtať len otázky $\leq l$ (kde l je najlepšie dolné ohraničenie m, o ktorom aktuálne vie). Na to, aby uspel, potrebujeme nájsť $l < m \leq r$ také, že $r - l$ je rádovo nanajvýš l. Túto časť prenechávame vám – ak ste sa ale dočítali až sem, tak by ste s tým nemali mať problém.

Hľadanie pôvodného stavu

Po tom, čo nájdeme m, nám stačí zistiť aktuálny stav. Môžeme si totiž pamätať všetky položené otázky, a keď zistíme aktuálny stav, jednoducho dopočítame pôvodný stav x_0.

Ako nájsť aktuálny stav? No predsa binárnym vyhľadávaním. V ňom sa potrebujeme vedieť pýtať "je x aspoň b?" To vieme v našej úlohe zistiť tak, že položíme otázku $m - b$. Ak dostaneme $>$, tak $x < b$, v opačnom prípade $x \geq b$. Následne sa vrátime k predchádzajúcemu stavu tým, že sa spýtame b (a odpoveď ignorujeme).

4. Turnaj Veľkých Šachistov

návod písal Hodobox
(max. 0 b za popis, 20 b za program)

Pred čítaním vzorového riešenia tejto úlohy si určite prečítajte aj vzorák jej ľahšej verzie z minulej série.

Táto úloha, rovnako ako jej predchodca, sa vyznačuje obtiažnosťou implementovať riešenie v kontraste s jeho samotným vymyslením; berie však túto úvahu trocha ďalej. Zatiaľ čo v Turnaji Mladých Šachistov nám malé šachovnice dovolili naprogramovať prakticky bruteforce typu "skúšaj všetky možné ťahy" a na nás bolo len rozvážne sa týmto skúšaním prehrýzť, v tejto úlohe bolo potrebné dávať si pozor, aby riešenie, s ktorým prichádzame, bolo zvládnuteľné. Ľahko sa totiž môže stať, že vymyslíme riešenie, ktoré je síce dosť rýchle, ale naloží nám oveľa viac roboty, ako by malo. Čím viac častí nášho riešenia používalo rovnaký kód, čím bol principiálne totožnejší s riešením predošlej úlohy, tým lepšie.

Vzorové riešenie je teda podobné ako pre Turnaj Mladých Šachistov, len sa trochu viac zahráme s tým, čo vlastne musíme urobiť, aby sme vedeli povedať, ktorý kráľ je v ohrození a ako sa vie zachrániť. Pri skúšaní všetkých možných pohybov figúrok nám totiž veľká väčšina pokusov bude naničč – nás zakaždým zaujíma, či dokážeme vyhodiť jednu predom zvolenú nepriateľskú figúrku, a pritom na overenie tohto berieme postupne všetky naše figúrky, vyskúšame všetky možné pohyby, a potom overujeme, či sme sa náhodou netrafili. Vedeli by sme nejak konkrétnejšie povedať, ktoré figúrky majú šancu na úspech? Kľúč je v tom, že všetky pohyby okrem pešiakov sú symetrické. (Napríklad ak náš kôň ohrozuje nejakú figúrku, tak ak by táto figúrka bola koňom, ohrozovala by toho nášho.)

Zoberme si teda nejakú figúrku a skúsme zistiť, čím je ohrozená. Ak je ohrozená nepriateľským kráľom, ten musí byť na jednom z ôsmich políčok okolo nej. Namiesto toho, aby sme kráľom skúšali jeho osem pohybov, spravíme opak – vyskúšame osem pohybov našou figúrkou, ako keby bola kráľom, a zistíme, či na týchto pozíciách

nie je nepriateľský kráľ. Podobne je to s pešiakmi a koňmi – ak našu figúrku ohrozuje nejaký pešiak, ten môže byť len na dvoch možných políčkach, a kôň na ôsmich – vyskúšame teda týchto dokopy 10 pozícií, či sa práve na nich nenachádza konkrétne táto nepriateľská figúrka, a ak áno, našli sme figúrku, ktorá nás ohrozuje. Teda zatiaľ namiesto toho aby sme všetkými nepriateľskými koňmi, pešiakmi a kráľom skúšali vyhodiť nami zvolenú figúrku, stačí sa pozrieť na $8 + 2 + 8$ pozící okolo nej a vieme presne povedať či, koľko, a skadiaľ nás figúrky tohto typu ohrozujú.

Rovnaká úvaha zafunguje aj pre typy figúrok, ktoré sa pohybujú donekonečna v niektorých smeroch. Ak nás totiž ohrozuje nejaký strelec, ten musí byť v jednom zo štyroch smerov od našej figúrky, a navyše to musí byť prvá figúrka v tomto smere. Od nami zvolenej figúrky sa nám teda stačí pozrieť do všetkých ôsmich smerov, nájsť prvú figúrku na ktorú v tomto smere narazíme, a zistiť či sa táto figúrka dokáže v tomto smere donekonečna pohybovať – ak áno, ohrozuje nás. Samozrejme, vzhľadom na rozmery šachovnice túto časť už nestíhame simulovať krok po kroku; namiesto toho budeme pozerať na susedné figúrky v nami predpočítaných setoch.

Ak by sme teda chceli zistiť, či kolmo vľavo je alebo nie je ohrozujúca figúrka, tak najprv po načítaní vstupu vytvoríme pole setov veľkosti n, s tým, že set na pozícií i v sebe bude udržiavať figúrky s y-ovou súradnicou rovnou i, zoradené podľa ich x-ovej súradnice. Teraz v sete, v ktorej je umiestnená naša figúrka, ju vyhľadáme a pozrieme sa, čo sa v ňom nachádza pred našou figúrkou – ak tam je nepriateľská figúrka, o ktorej máme zaznamenané že je typu 'neobmedzený pohyb' a má povolený pohyb vpravo, tak nás ohrozuje. Analogicky, figúrka, ktorá je za našou figúrkou v tomto sete nás ohrozuje, ak má povolený pohyb vľavo.

Spolu s týmto poľom setov, ktoré používame na overovanie ohrozenia figúrky zľava a sprava, si vytvoríme pole setov na overovanie ohrozenia zhora a zdola (v sete budú figúrky usporiadané podľa y-ovej súradnice, a figúrku umiestnime do setu zodpovedajúcemu jej x-ovej súradnici), a pole setov reprezentujúcich diagonálu idúcu zľava hore doprava dole, a diagonálu idúcu zľava dole doprava hore. V každom z týchto setov si potom nájdeme našu figúrku, pozrieme sa na jej predchodcu a jej následníka, a overíme, či je to nepriateľská figúrka, ktorá sa vie hýbať v našom smere; ak áno, ohrozuje nás.

Overenie či je figúrka ohrozená nás teda stojí zhruba toľkoto operácii: $18 = O(1)$ (overenie konkrétnych pozícií na ktorých môže byť ohrozujúci kráľ, kôň, či pešiak) $+O(4 \log f)$ (v štyroch setoch vyhľadáme našu figúrku a pozrieme sa na predchodcu a následníka)

Vytvorenie štyroch polí setov a vloženie figúrok do nich nám na začiatku zaberie $O(n + f \log f)$ času.

Ostáva nám teda vymyslieť, ako vieme efektívnu funkciu 'koľko figúrok ma ohrozuje' využiť na vyriešenie celej úlohy. Najprv si teda zistíme, koľko figúrok ohrozuje bieleho a čierneho kráľa – ak sú ohrození alebo neohrození obaja, povieme príslušnú hlášku. Zaujímavé to teda je, ak je ohrozený práve jeden kráľ. Vtedy máme dve možnosti:

Prvá je pohnúť sa s kráľom, čím ho možno dostaneme z ohrozenia. To už s našou funkciou hravo zvládneme – vyskúšame všetky možné pohyby kráľa (pri ktorých ho najprv zmažeme zo všetkých štruktúr a potom naspäť pridáme s novými súradnicami), a po každom pohybe sa spýtame, či je kráľ ohrozený. Ak nie je, našli sme jeden platný ťah.

Druhá možnosť je zobrať inú priateľskú figúrku a pohnúť ňou tak, aby buď vyhodila alebo sa postavila do cesty tej figúrke, ktorá ohrozuje kráľa. Všimnite si, že toto sa ozaj dá spraviť iba ak kráľa ohrozuje práve jedna nepriateľská figúrka – ak ho totiž ohrozuje viacero, ich cesty ku kráľovi nemajú spoločný bod, a tak žiadnym pohnutím inou figúrkou nevieme súčasne ochrániť kráľa od viac ako jednej figúrky. Zoberme si teda tú figúrku, ktorá ohrozuje kráľa. To, že ho 'ohrozuje' vlastne znamená to, že má taký pohyb, ktorým sa vie presunúť na políčko s kráľom. Ak umiestnime ľubovoľnú figúrku (okrem kráľa) na hociktoré políčko na tejto ceste, vrátane začiatočnej pozície tejto figúrky, tak jej vlastne zabránime kráľa vyhodiť. Táto cesta od nepriateľskej figúrky ku kráľovi nie je dlhšia ako n políčok. Čo si teda môžeme dovoliť robiť je zobrať túto nepriateľskú figúrku, krok po kroku ju posúvať smerom k nášmu kráľovi, a zakaždým sa spýtať či ho nejaká priateľská figúrka neohrozuje. Tieto hodnoty spočítame; ak máme nulu, nevieme kráľa ochrániť, a má mat; inak má šach, a povieme počet platných ťahov, ktoré sme objavili. Ťahov touto nepriateľskou figúrkou spravíme teda $O(n)$ a každé overenie, koľko našich figúrok ju vie ohroziť, robíme v $O(\log f)$. Dokopy máme časovú zložitosť $O(n \log f)$, čo je dostatočne rýchle.

Treba si pritom dať ešte pozor na pár výnimiek.

Za prvé, pri skúšaní 'vyhodenia' nepriateľskej figúrky, ktorá ohrozuje kráľa na jej ceste k nemu, v prvom kroku (keď sa ešte nepohla) ju vie priateľský pešiak vyhodiť došikma, avšak vo všetkých ostatných krokoch ju vie vyhodiť len smerom dopredu (keďže táto nepriateľská figúrka reálne nestojí na pozícií, na ktorej sa ju

snažíme vyhodiť, iba sa cez neho posúva – pešiak sa na toto prázdne políčko naozaj vie dostať len obyčajným ťahom vpred, nie šikmým).

Za druhé, nesmieme použiť priateľskú figúrku, ktorá ak sa pohne, dovolí inej nepriateľskej figúrke ohrozovať kráľa. Tie nájdeme tak, že keď zisťujeme na začiatku, či sú králi v ohrození, a pri overovaní v jednom smere naďabíme na priateľskú figúrku, pozrieme sa ešte za ňu, a ak tu nájdeme nepriateľskú figúrku ktorá by nás ohrozovala ak by tam naša figúrka nebola prítomná, zaznačíme si na našu figúrku, že ňou nebudeme hýbať.

Nakoniec, predstavte si nasledovnú situáciu: biely kráľ je v strede šachovnice, tri políčka napravo od neho je čierna veža, a napravo od nej je biela veža. Keď budeme čiernou vežou hýbať ku kráľovi, zakaždým nám vyjde, že ju biela veža vie vyhodiť. Avšak bielou vežou máme len jeden platný ťah – vyhodiť čiernu vežu tam, kde najprv stála. Pre každú figúrku si teda ešte zapamätáme, v ktorom smere nám už vyšlo, že vie tohoto nepriateľa vyhodiť, a naozaj zarátame platný ťah len prvý krát pre každý smer.

Upozornenie: slabým náturám môže byť z nasledujúceho kódu nevoľno. Pred pokusmi prehrýzť sa ním odporúčame hlboký nádych a fľašu kofoly. Na úplný záver ešte dodávame, že existuje aj riešenie s časovou zložitosťou $O(f \log f)$, teda nezávislá na rozmeroch šachovnice. Nechávame ho na premotivovaného čitateľa.

Listing programu (C++)

```
#include <bits/stdc++.h>

using namespace std;

string
white_player = "Biely",
black_player = "Cierny",
impossible = "Nemozna situacia.",
nothing_special = "Neutralna situacia.",
checkmate_msg = " hrac ma mat.",
check_msg = " hrac ma sach. Ma ",
announce_msg = " platnych tahov.";

string answer(string who,int good_moves) //creates answer for check case as a single string
{
    stringstream ss;
    ss << good_moves;
    string ans;
    ss >> ans;
    ans = who + check_msg + ans + announce_msg;
    return ans;
}

struct chesspiece
{
    bool critical = false,colour;
    bool has_killed_in_dir[8] = {false};
    int id,x,y,type,killdir;
    char c;
};

map<pair<int,int>,int> board; //ID of piece on coordinates [x][y]
bool pawns_can_kill; //if true, pawns attempt to attack in is_under_threat(who).
//otherwise, they try to move into the spot
int n,f,typify[500]; //chessboard size, chesspiece #, identification of pieces
chesspiece a_threat;
vector<chesspiece>figurky;
vector< set< pair<int,int> > > row,col,lddiag,rddiag;
// data[pos](coord,id) i.e. row[which row](left to right coord, id)
// row: pos = y, coord = x (left to right) small to large
// column: pos = x, coord = y (small to large = downwards)
// lddiag: pos = x+y, coord = y towards down left (small to large)
// rddiag: pos = x-y+n = [1,2n-1], coord = y towards down right (small to large)

struct movement
{
    vector<int> dx,dy;
};
movement pieces[6];

bool valid(int x,int y)
{
    return (x>=1&&y>=1&&y<=n&&x<=n);
}

void clr() //wipes data from previous testcase
{
    board.clear();
    figurky.clear();
    row.clear();
    col.clear();
    lddiag.clear();
    rddiag.clear();
    pawns_can_kill = true;
}

// checks whether the distance between two pieces is 1, or more
int dist(chesspiece a,chesspiece b)
{
    int x = a.x,y=a.y;
```

```
    for(int i=0;i<pieces[0].dx.size();++i)
    {
        int nx = a.x+pieces[0].dx[i];
        int ny = a.y+pieces[0].dy[i];
        if(nx==b.x&&ny==b.y)
            return 1;
    }
    return 2;
}

void preprocess() //intitialize chesspieces,
{
    //typify
    typify[ 'k' ] = typify[ 'K' ] = 0;
    typify[ 'p' ] = typify[ 'P' ] = 1;
    typify[ 'h' ] = typify[ 'H' ] = 2;
    typify[ 'r' ] = typify[ 'R' ] = 3;
    typify[ 'b' ] = typify[ 'B' ] = 4;
    typify[ 'q' ] = typify[ 'Q' ] = 5;


    //hardcoded chessboard pieces

    //king
    pieces[0].dx.push_back(-1);pieces[0].dy.push_back(-1);
    pieces[0].dx.push_back(-1);pieces[0].dy.push_back(0);
    pieces[0].dx.push_back(-1);pieces[0].dy.push_back(1);
    pieces[0].dx.push_back(0);pieces[0].dy.push_back(-1);
    pieces[0].dx.push_back(0);pieces[0].dy.push_back(1);
    pieces[0].dx.push_back(1);pieces[0].dy.push_back(-1);
    pieces[0].dx.push_back(1);pieces[0].dy.push_back(0);
    pieces[0].dx.push_back(1);pieces[0].dy.push_back(1);

    //queen
    pieces[5].dx = pieces[0].dx;
    pieces[5].dy = pieces[0].dy;

    //rook
    pieces[3].dx.push_back(-1);pieces[3].dy.push_back(0);
    pieces[3].dx.push_back(1);pieces[3].dy.push_back(0);
    pieces[3].dx.push_back(0);pieces[3].dy.push_back(-1);
    pieces[3].dx.push_back(0);pieces[3].dy.push_back(1);

    //bishop
    pieces[4].dx.push_back(1);pieces[4].dy.push_back(1);
    pieces[4].dx.push_back(-1);pieces[4].dy.push_back(-1);
    pieces[4].dx.push_back(-1);pieces[4].dy.push_back(1);
    pieces[4].dx.push_back(1);pieces[4].dy.push_back(-1);

    //knight
    pieces[2].dx.push_back(-1);pieces[2].dy.push_back(-2);
    pieces[2].dx.push_back(-2);pieces[2].dy.push_back(-1);
    pieces[2].dx.push_back(-2);pieces[2].dy.push_back(1);
    pieces[2].dx.push_back(-1);pieces[2].dy.push_back(2);
    pieces[2].dx.push_back(1);pieces[2].dy.push_back(2);
    pieces[2].dx.push_back(2);pieces[2].dy.push_back(1);
    pieces[2].dx.push_back(2);pieces[2].dy.push_back(-1);
    pieces[2].dx.push_back(1);pieces[2].dy.push_back(-2);
}

// decides whether chesspiece who can threaten another in the direction how
// if its a king, pathlen has to be 1
int threatens(chesspiece who,pair<int,int> how,int pathlen)
{
    int type = who.type;

    if(type==1||type==2) //pawns and knights are dealt with in parallel.
        return -1;

    if(type==0&&pathlen>1)
        return -1;

    for(int i=0;i<pieces[who.type].dx.size();++i)
    {
        if( pieces[who.type].dx[i] == how.first && pieces[who.type].dy[i] == how.second)
            return i;
    }

    return -1;
}

void vanish(chesspiece who) //deletes 'who' from all sets
{
    board.erase({who.x,who.y});
    col[who.x].erase( {who.y,who.id} );
    row[who.y].erase( {who.x,who.id} );
    lddiag[who.x+who.y].erase( {who.y,who.id} );
    rddiag[who.x-who.y+n].erase( {who.y,who.id} );
}

bool appear(chesspiece who) //inserts 'who' into all sets
{
    if(board.find({who.x,who.y})!=board.end() ) return false;
    //returns true if no collission encountered
    board[{who.x,who.y}] = who.id;
    col[who.x].insert( {who.y,who.id} );
    row[who.y].insert( {who.x,who.id} );
    lddiag[who.x+who.y].insert( {who.y,who.id} );
    rddiag[who.x-who.y+n].insert( {who.y,who.id} );
```

```
    return true;
}

bool can_attack(chesspiece a,chesspiece b) //can 'a' kill 'b'?
{
    return(a.colour!=b.colour);
}

int check_set(chesspiece &a_threat, vector< set < pair<int,int> > > &S,
              int lookup,int order,int iterate_dir, pair<int,int>threat_dir,
              chesspiece who,int trial)
{
    /*
    Checks whether the neighbouring chesspiece of 'who' in set S
    in the iteration direction (left,right) = iter_dir contains a
    enemy piece that is able to threaten us in the direction threat_dir.

    Conforms to the use of 'king_trial' specified in is_under_threat,
    i.e. ignores critical pieces if it is set to 0...
    */

    int result = 0;
    set<pair<int,int> >::iterator me,neighbour,dead_end;
    chesspiece threat,crit;
    if(iterate_dir==-1) dead_end = S[lookup].begin();
    else
    {
        dead_end = S[lookup].end();
        dead_end--;
    }
    me = S[lookup].find({order,who.id});
    if(me!=dead_end)
    {
        neighbour = me;
        if(iterate_dir==-1) neighbour--;
        else neighbour++;
        threat = figurky[neighbour->second];
        if(threat.colour != who.colour && (trial!=0 || threat.critical==false) )
        {
            int killdir = threatens(threat,threat_dir,dist(threat,who));
            if(killdir!=-1)
            {
                //A piece can kill another piece in a move in a single direction only once
                //i.e. scenario: a rook threatening the king is lined up with our queen, which
                //thinks it can stop the rook at every point on its path as a unique 'move',
                //when in reality only taking it out in its original position is legitimate
                if(trial==0)
                    if(threat.has_killed_in_dir[killdir]==true) result--;
                    else figurky[neighbour->second].has_killed_in_dir[killdir] = true;

                result++;
                if(trial==1)
                {
                    a_threat = threat;
                    a_threat.killdir = killdir;
                }
            }

        }
        else if (trial == 1 && neighbour!=dead_end)
        {
            crit = threat;
            if(iterate_dir==-1) neighbour--;
            else neighbour++;
            threat = figurky[neighbour->second];
            if(threat.colour!=who.colour && threatens(threat,threat_dir,dist(threat,who))!=-1)
                figurky[crit.id].critical = true;

        }
    }
    return result;
}

int is_under_threat(chesspiece who,int king_trial)
{
    /*
    returns the number of enemy pieces which kill 'who'.

    if king_trial is set to 1, it additionally determines
    the killdir of threats and criticalness of allies, and sets a_threat
    to the last threatening enemy piece found.

    critical enemies are skipped if king_trial is 0.
    */
    int threat_count = 0;

    //check pawns

    int pawndir = 1;
    if(who.colour) pawndir = -1;

    for(int i=0;i<2;++i)
    {
        int dx = 1 - 2*i;
        if(pawns_can_kill==false) dx = 0;
        if(pawns_can_kill==false && i) break;
        if(valid(who.x+dx,who.y+pawndir))
        {
            if(board.find({who.x+dx,who.y+pawndir})==board.end()) continue;
```

```
            int id = board[{who.x+dx,who.y+pawndir}];
            if(figurky[id].type!=1) continue;
            if(figurky[id].colour==who.colour) continue;
            if(king_trial==0 && figurky[id].critical) continue;
            threat_count++;
            if(king_trial==1)
            {
                a_threat = figurky[id];
            }
        }
    }


    //check knights

    for(int i=0;i<pieces[2].dx.size();++i)
    {
        int nx = who.x + pieces[2].dx[i];
        int ny = who.y + pieces[2].dy[i];
        if(board.find({nx,ny})!=board.end())
        {
            int id = board[{nx,ny}];
            if(can_attack(figurky[id],who) == false) continue;
            if(figurky[id].type != 2) continue;
            if(king_trial==0 && figurky[id].critical) continue;
            threat_count++;
            if(king_trial==1)
            {
                a_threat = figurky[id];
                a_threat.killdir = i;
            }
        }
    }

    //check all 8 directions

    //up
    threat_count += check_set( a_threat, col, who.x, who.y, -1, {0,1}, who, king_trial);
    //down
    threat_count += check_set( a_threat, col, who.x, who.y,  1, {0,-1}, who, king_trial);
    //left
    threat_count += check_set( a_threat, row, who.y, who.x, -1, {1,0}, who, king_trial);
    //right
    threat_count += check_set( a_threat, row, who.y, who.x,  1, {-1,0}, who, king_trial);
    //left down
    threat_count += check_set( a_threat, lddiag, who.x+who.y , who.y,1,{1,-1},who,king_trial);
    //left up
    threat_count += check_set( a_threat, rddiag, who.x-who.y+n , who.y,-1,{1,1},who,king_trial);
    //right down
    threat_count += check_set( a_threat, rddiag, who.x-who.y+n , who.y,1,{-1,-1},who,king_trial);
    //right up
    threat_count += check_set( a_threat, lddiag, who.x+who.y , who.y,-1,{-1,1},who,king_trial);
    return threat_count;

}

string solve()
{

    clr(); //clear data from previous case
    //read input
    int i,k;
    cin >> n >> f;

    figurky.resize(f);
    col.resize(n+1);
    row.resize(n+1);
    rddiag.resize(n*2+2);
    lddiag.resize(n*2+2);

    chesspiece white,black;
    for(i=0;i<f;++i)
    {
        cin >> figurky[i].x >> figurky[i].y >> figurky[i].c;
        if(isupper( figurky[i].c ) == 0) figurky[i].colour = false;
        else figurky[i].colour = true;
        figurky[i].id = i;
        figurky[i].type = typify[ figurky[i].c ];
        if(figurky[i].c == 'k') white = figurky[i];
        if(figurky[i].c == 'K') black = figurky[i];

    }
    //insert into all 4 sets
    for(i=0;i<f;++i)
    {
        appear(figurky[i]);
        board[ {figurky[i].x,figurky[i].y} ] = i;
    }

    //try both kings
    int wa,ba;
    chesspiece white_killer,black_killer;
    wa = is_under_threat(white,1);
    white_killer = a_threat;
    ba = is_under_threat(black,1);
    black_killer = a_threat;
    if(wa && ba) return impossible;
    if(!wa && !ba) return nothing_special;
```

```
    int threats;
    chesspiece killer,king;
    string player;
    if(wa)
    {
        player = white_player;
        king = white;
        threats = wa;
        killer = white_killer;
    }
    else //ba
    {
        player = black_player;
        king = black;
        threats = ba;
        killer = black_killer;
    }
    int good_moves = 0;

    //try all 8 moves

    for(i=0;i<8;++i)
    {
        vanish(king);
        chesspiece temporary_king = king;
        temporary_king.x += pieces[0].dx[i];
        temporary_king.y += pieces[0].dy[i];
        if( valid(temporary_king.x,temporary_king.y) == false )
        {
            appear(king);
            continue;
        }
        if ( appear(temporary_king) )
        {
            if(is_under_threat(temporary_king,2)==0)
                good_moves++;

            vanish(temporary_king);
        }
        else
        {
            chesspiece collission = figurky[ board[ {temporary_king.x,temporary_king.y} ] ];
            if( can_attack(collission,temporary_king))
            {
                vanish(collission);
                appear(temporary_king);
                if(is_under_threat(temporary_king,2)==0)
                    good_moves++;

                vanish(temporary_king);
                appear(collission);
            }
        }

        appear(king);
    }

    if(threats>1)
    {
        if(good_moves == 0) return player + checkmate_msg;
        return answer(player,good_moves);
    }

    figurky[ king.id ].critical = true;
    int tmpkilldir = killer.killdir;
    killer = figurky[ board[ {killer.x,killer.y} ] ];
    killer.killdir = tmpkilldir;
    if(killer.type == 1 || killer.type == 2) //pawns and knights don't move. Either we kill them or we lose.
    {
        good_moves += is_under_threat(killer,0);
        if(good_moves == 0) return player + checkmate_msg;
        return answer(player,good_moves);
    }

    //Try to attack every tile on its path, including itself.
    while(killer.x!=king.x||killer.y!=king.y)
    {
        good_moves += is_under_threat(killer,0);
        pawns_can_kill = false; //killer moved, pawns can't attack new position
        //since he wasn't there originally
        vanish(killer);
        killer.x += pieces[ killer.type].dx[ killer.killdir ];
        killer.y += pieces[ killer.type].dy[ killer.killdir ];

        appear(killer);
    }

    if(good_moves == 0) return player + checkmate_msg;
    return answer(player,good_moves);

}

int main()
{
    ios::sync_with_stdio(false);
    cin.tie(0);
    cout.tie(0);
    preprocess();
```

```
    int t;
    string tmp;
    cin >> t;
    while(t--) cout << solve() << "\n";

    return 0;
}
```

návod písal Buj

5. Tam, kde žijú algoritmy...

(max. 0 b za popis, 20 b za program)

Ukážeme si dve riešenia, prvé bežiace v čase $O(n^2 \log n)$ (za ktoré sa dalo získať 12 až 20 bodov) a druhé $O(n^2)$ (za plných 20 bodov). Nie je to ale tak, že by to rýchlejšie "budovalo" na tom pomalšom, oplatí sa preto prečítať si obe riešenia.

Úvaha na začiatok

Zoberme si ľubovoľný strom a niektorý jeho vrchol v taký, že jeho odobratím sa strom rozpadne na podstromy veľkosti nanajvýš $\frac{n}{2}$. Potom ľahko vidno, že v je jeho jediný centroid práve vtedy, keď každý z tých podstromov má veľkosť ostro menšiu ako $\frac{n}{2}$.

Označme počet ľahkých stromov s n vrcholmi ako $f(n)$. Z vyššie uvedeného vyplýva, že $f(n) = n \cdot nieco(n)$.

$nieco(n)$ je vlastne počet rozdelení $a = n - 1$ vrcholov do niekoľkých častí, pričom každá z nich je ľahký strom s najviac $b = \lfloor \frac{n-1}{2} \rfloor$ vrcholmi, a jeden vrchol z každého stromu je spojený s centroidom. Pre všeobecné a a b to budeme označovať $rozdel(a, b)$.

Ak chceme vypočítať $rozdel(a, b)$, tak sa oplatí zamyslieť nad tým, ako by sme vedeli systematicky riešiť problém "máš a vrcholov, postav z nich niekoľko ľahkých stromov veľkosti najviac b".

Riešenie 0 "za nula bodov"

Môžeme napríklad stavať stromy postupne. Pri stavaní jedného stromu musíme určiť jeho veľkosť k, potom množinu jeho vrcholov, a jeho tvar (teda ako sú vrcholy v ňom pospájané). Nakoniec ho musíme niektorým vrcholom spojiť s centroidom. O ostatné stromy sa postaráme tak, že vlastne riešime ten istý problém: "máš $a - k$ vrcholov, postav z nich niekoľko ľahkých stromov veľkosti najviac b".

Ak sme si zvolili veľkosť k, tak na množinu jeho vrcholov máme $\binom{a}{k}$ možností, na jeho tvar je $f(k)$ možností, a vrchol spojený s centroidom vieme vybrať k spôsobmi. Teda to vyzerá tak, že

$$rozdel(a, b) = \sum_{k=1}^{b} \binom{a}{k} \cdot f(k) \cdot k \cdot rozdel(a - k, b)$$

Šikovný čitateľ si ale isto všimol, že horeuvedeným postupom vieme niektoré rozdelenia dosiahnúť viacero spôsobmi, a teda neplatí, že počet rôznych postupov je rovný $rozdel(a, b)$. Problém je vlastne v tom, že si môžeme vybrať poradie, v akom budeme stromy zostrojovať, pričom ale výsledné rozdelenie nezávisí od tohto poradia.

Riešenie 1 "od najväčšieho po najmenší"

To môžeme vyriešiť tak, že budeme stromy zostrojovať od najväčšieho po najmenší, pričom v jednom kroku zostrojíme naraz všetky najväčšie (ešte nepostavené) stromy. Pribudne nám teda ešte jeden parameter – počet stromov s veľkosťou b, ktorý môže byť od 0 po $\lfloor \frac{a}{b} \rfloor$.

Ak sme si zvolili počet stromov s, tak množiny vrcholov vieme vybrať

$$\frac{\prod_{i=0}^{s-1} \binom{a-i \cdot b}{b}}{s!}$$

spôsobmi. Na ich tvary máme $f(b)^s$ možností, a vrcholy spojené s centroidom vieme vybrať b^s spôsobmi. Dokopy

$$rozdel(a, b) = \sum_{s=0}^{\lfloor \frac{a}{b} \rfloor} \frac{\prod_{i=0}^{s-1} \binom{a-ib}{b}}{s!} \cdot f(b)^s \cdot b^s \cdot rozdel(a - sb, b - 1)$$

Pozrime sa teraz na časovú zložitosť. Všetky kombinačné čísla a faktoriály modulo p si vieme dopredu predrátať, čiže to je konštanta. Na výpočet $rozdel(a, b)$ potrebujeme rádovo $\frac{a}{b}$ sčítancov, a každý z nich vieme spočítať v konštantnom čase. Pre pevné a nás zaujímajú b od 1 po $\lfloor \frac{a}{2} \rfloor$, a stojí nás to rádovo $\frac{a}{1} + \frac{a}{2} + \ldots + \frac{a}{\lfloor \frac{a}{2} \rfloor} \approx a \log a$. Keď to sčítame cez všetky a (od 1 po n), dostaneme časovú zložitosť $O(n^2 \log n)$.

Riešenie 2 "to sú mi triky"

V tomto riešení nebudeme počítať $rozdel(a, b)$ všeobecne, ale len pre tú hodnotu, ktorá nás najviac zaujíma: pre $b = \lfloor \frac{a}{2} \rfloor$.

Zamyslime sa najprv nad tým, koľkými spôsobmi vieme a vrcholov rozdeliť do niekoľko ľahkých stromov a každý z nich spojiť jednou hranou s centroidom, ale bez obmedzenia na veľkosť stromu. V podstate nás teda zaujíma $rozdel(a, a)$.

Túto hodnotu budeme počítať podobným spôsobom, ako v riešení za nula bodov. Jednoznačnosť zaručíme tak, že v každom kroku zostrojíme strom obsahujúci vrchol s najmenším číslom. Dostávame tak mierne odlišný vzťah

$$rozdel(a, a) = \sum_{k=1}^{a} \binom{a}{k-1} \cdot f(k) \cdot k \cdot rozdel(a-k, a-k)$$

(Všimnite si, že pri voľbe množiny vrcholov vyberáme o 1 vrchol menej, a preto tam je $\binom{a}{k-1}$ namiesto $\binom{a}{k}$.)

Samozrejme, nemôžeme túto hodnotu prehlásiť za $\frac{f(a+1)}{a+1}$ – zarátali sme tam totiž aj rozdelenia, v ktorých existuje strom s viac ako $\lfloor \frac{a}{2} \rfloor$ vrcholmi.

Koľko takých *zlých* stromov ale môže existovať? Ak by boli dva, tak by sme mali aspoň $2 \cdot (\lfloor \frac{a}{2} \rfloor + 1)$ vrcholov okrem centroidu, čo je väčšie než $2 \cdot \frac{a}{2} = a$, skutočný počet vrcholov okrem centroidu. To by bol spor. Takže zlý strom je vždy najviac jeden.

To ale znamená, že všetky zlé možnosti vieme zostrojiť tak, že najprv zostrojíme zlý strom, a potom všetky ostatné (pri ktorých už nedbáme na obmedzenia veľkosti). Týchto možností teda je

$$zle(a) = \sum_{z=\lceil \frac{a}{2} \rceil}^{a} \binom{a}{z} \cdot f(z) \cdot z \cdot rozdel(a-z, a-z)$$

A máme tak

$$f(a+1) = (a+1) \cdot rozdel\left(a, \left\lfloor \frac{a}{2} \right\rfloor\right) = (a+1) \cdot (rozdel(a, a) - zle(a))$$

Toto riešenie má časovú zložitosť $O(n^2)$, lebo počítame iba n hodnôt $rozdel(1, 1), \ldots, rozdel(n, n)$, a každú z nich vypočítame v $O(n)$.