Dvadsiaty tretí ročník Olympiády v informatike

Odborným garantom súťaže Olympiáda v informatike je Slovenská informatická spoločnosť. Viac sa o nej dozviete na stránke `http://www.informatika.sk/`

RNDr. Michal Forišek, Slovenská komisia OI
Dvadsiaty tretí ročník Olympiády v informatike
ISBN 978-80-8072-095-7

Obsah

O priebehu 23. ročníka Olympiády v informatike

Dvadsiaty tretí ročník Olympiády v informatike (OI) bol zároveň jej druhým ročníkom v „novom šate“ – predchádzajúcich 21 rokov fungovala táto súťaž pod hlavičkou Matematickej olympiády ako kategória P.

Podobne ako v minulom ročníku, aj v tomto prebehla súťaž v dvoch kategóriách. V oboch kategóriách sa uskutočnilo domáce a krajské kolo. kategórii B, ktorá je určená pre prvákov a druhákov klasických štvorročných stredných škôl, súťaž týmto kolom skončila. V kategórii A boli najlepší riešitelia krajských kôl pozvaní na celoštátne kolo, ktoré sa uskutočnilo v Banskobystrickom kraji – v krásnej obci Hronec, kde začína Čiernohronská železnica.

Najlepší riešitelia kategórie A boli následne pozvaní na týždňové výberové sústredenie, kde sa určila reprezentácia Slovenska na medzinárodných súťažiach – Stredoeurópskej olympiáde v informatike (CEOI) v nemeckých Drážďanoch a Medzinárodnej olympiáde v informatike (IOI) v egyptskej Káhire.

Na celoslovenskej úrovni sa počas celého ročníka o plynulý chod OI starala Slovenská komisia OI v nasledujúcom zložení:

- doc. RNDr. Gabriela Andrejková, CSc.,
  predsedkyňa, ÚI PF UPJŠ, Košice
- RNDr. Michal Forišek, podpredseda, KI FMFI UK, Bratislava
- RNDr. Andrej Blaho, FMFI UK, Bratislava
- Mgr. Juliana Lipková, FMFI UK, Bratislava
- Bc. Ján Katrenič, PF UPJŠ, Košice
- Bc. Lukáš Poláček, FMFI UK, Bratislava
- PaedDr. Miloslava Sudolská, PhD.,
  KI FPV UMB, krajská predsedkyňa pre BB
- RNDr. Eva Hanulová, Gym. J. Hronca, krajská predsedkyňa pre BA
- RNDr. Rastislav Krivoš-Belluš,
  Ústav informatiky PF UPJŠ, krajský predseda pre KE
- prof. Ing. Veronika Stoffová, CSc.,
  KI PF UJS, krajská predsedkyňa pre NR

- Mgr. Mária Majherová, Ph.D.,
  Katedra matematiky FHPV PU, krajská predsedkyňa pre PO
- Ing. Andrea Julény, KI FMech TnUAD, krajská predsedkyňa pre TN
- Mgr. Blanka Thomková, Gym. Jána Hollého, krajská predsedkyňa pre TT
- RNDr. Peter Varša, Ph.D., KI FRI ŽU, krajský predseda pre ZA

Samozrejme, OI by nemohla fungovať bez mnohých ďalších ľudí – či už Ing. Tomáša Lučeniča z IUVENTY, ktorý neúnavne zabezpečuje organizačnú stránku súťaže, dobrovoľníkov z Korešpondenčného seminára z programovania, ktorí pracujú v úlohovej komisii, zabezpečujú po technickej stránke celoštátne kolo a výberové sústredenie, študentov a pedagógov, ktorí v rámci krajských komisii vyhodnocujú súťažné riešenia, pracovníkov na krajských školských úradoch a v centrách voľného času, a samozrejme v prvom rade učiteľov stredných škôl, ktorí si aj v dnešnej dobe dokážu nájsť čas a chuť na prácu s nadanými žiakmi.

Michal Forišek, podpredseda SK OI

Zadania domáceho kola kategórie A

A-I-1 O zdanlivom kopci

Miško a Jožko sa chystali na výlet. Vymysleli si trasu, kadiaľ spolu pôjdu, potom každý vytiahol svoju mapu a začal si trasu pozerať. Niektoré body, cez ktoré trasa viedla, mali na mapách zapísanú nadmorskú výšku.

„To je zaujímavé," povedal po chvíli Miško. „Keď sa pozerám len na nadmorské výšky bodov, cez ktoré pôjdeme, vyzerá to, že ideme len cez jeden kopec: najskôr hore, potom dole."

„Ale kdeže, to nie je pravda." zadivil sa Jožko.

Netrvalo dlho, kým zistili, kde vznikol problém: mali rôzne presné mapy. Niektoré výšky, ktoré boli na Jožkovej mape vyznačené, na Miškovej chýbali.

Súťažná úloha:

Napíšte program, ktorý dostane na vstupe zoznam výšok na Jožkovej mape a zistí, koľko najviac z nich môže byť vyznačených aj na Miškovej mape.

Formát vstupu:

Prvý riadok vstupu obsahuje jedno celé číslo N ($1 \leq N \leq 100\,000$) – počet takých bodov na trase výletu, ktoré majú na Jožkovej mape vyznačenú nadmorskú výšku.

Ďalšie riadky obsahujú dokopy N celých čísel z rozsahu 1 až $1\,000\,000\,000$ – nadmorské výšky uvedených bodov (v nejakých vám neznámych jednotkách). Výšky sú uvedené v poradí, v akom na trase nasledujú.

Formát výstupu:

Výstup má obsahovať jediný riadok a v ňom jediné celé číslo K – najväčší počet výšok, ktoré mohli byť vyznačené na Miškovej mape.

Ak $a_1, \ldots, a_K$ sú výšky vyznačené na Miškovej mape, musia spĺňať nasledujúcu podmienku: pre nejaké x z množiny $\{1, \ldots, K\}$ musí platiť:

$$\forall i \in \{1, \ldots, x-1\};\ a_i < a_{i+1} \quad \wedge \quad \forall i \in \{x, \ldots, K-1\};\ a_i > a_{i+1}$$

Príklad:

Vstup

```
12
112 247 211 209 244
350 470 510 312 215
117 217
```

Výstup

```
9
```

Jedna možnosť ako mohli vyzerať výšky na Miškovej mape: 112, 211, 244, 350, 470, 510, 312, 215, 117.

Pomalšie riešenia:
Maximálne 7 bodov: Riešte tú istú úlohu pre $N \leq 1000$.
Maximálne 4 body: Riešte tú istú úlohu pre $N \leq 20$.

A-I-2 Rezervácie miesteniek

„V horách sa našlo zlato!“ šepkali si ľudia po celej krajine už niekoľko týždňov. Dobrodružnejšie povahy si už balili veci a smerovali priamo do hôr, do legendami opradenej oblasti pýšiacej sa menom Vyšná Klondika.

Keď však Klondiku zaplavila vlna nových zlatokopov, nastal nečakaný problém. Jediný miestny vláčik, ktorý na Klondike mali, odrazu býval tak preplnený, že sa doň polovica záujemcov ani nezmestila. A nedivte sa, že zlatokopa, ktorý sa ženie za čo najvýhodnejšou parcelou, takéto niečo poriadne nazlostí.

Keď to už vyzeralo, že onedlho príde na každej stanici ku krviprelievaniu, dostali železničiari spásonosný nápad. Budú na vláčik predávať miestenky.

Súťažná úloha:

Napíšte program, ktorý bude spracúvať rezervácie miest na jednu jazdu vlaku. Trať vlaku má $N + 1$ staníc, ktoré si v poradí, v akom na trati ležia, očíslujeme od 0 po N. Vo vlaku je M miest, medzi každou dvojicou po sebe nasledujúcich staníc teda vie previezť najviac M zlatokopov.

Váš program bude mať postupne spracovať niekoľko požiadaviek na rezerváciu miest. Každá požiadavka je tvaru „x ľudí chce ísť zo stanice y na stanicu z“. Ak ešte je na každom úseku trate medzi stanicami y a z vo vlaku ešte aspoň x miest voľných, má váš program takúto požiadavku prijať, inak ju má odmietnuť.

Formát vstupu:

Prvý riadok vstupu obsahuje tri celé čísla N, M a P ($1 \leq N, M, P \leq 100\,000$) – počet úsekov trate, počet miest vo vlaku a počet požiadaviek o rezerváciu.

Nasleduje P riadkov. Každý z nich popisuje jednu požiadavku v poradí, v akom prišli do systému. Presnejšie, i-ty z týchto riadkov obsahuje tri celé čísla x_i, y_i, z_i oddelené medzerami ($1 \leq x_i \leq M$, $0 \leq y_i < z_i \leq N$). Význam týchto hodnôt bol popísaný vyššie.

Formát výstupu:

Pre každú požiadavku vypíšte jeden riadok a v ňom buď reťazec „`prijata`“, ak danú požiadavku bolo ešte možné splniť, alebo reťazec „`odmietnuta`“, ak už vo vlaku nebolo dosť voľných miest.

Príklad:

Vstup

```
4 6 6
2 1 4
2 1 3
3 2 4
3 1 2
6 0 1
4 3 4
```

Výstup

```
prijata
prijata
odmietnuta
odmietnuta
prijata
prijata
```

Po prijatí prvých dvoch požiadaviek vieme, že v úsekoch medzi stanicami 1 a 2 a medzi 2 a 3 už budú len dve voľné miesta. Preto musíme odmietnuť nasledujúce dve trojčlenné skupiny, ktoré chcú cestovať aj na týchto úsekoch trate. Posledné dve požiadavky opäť môžeme splniť. U prvej je to zjavné, u druhej nám v stanici 3 vystúpi dosť ľudí na to, aby sa na posledný úsek trate uvoľnili presne tie potrebné štyri miesta.

Pomalšie riešenia:

Maximálne 7 bodov: Riešte pôvodnú úlohu pre $N \leq 10\,000$, s tým, že navyše môžete predpokladať, že pre približne 99% požiadaviek zo vstupu bude odpoveď „`odmietnuta`“.

Maximálne 4 body: Riešte pôvodnú úlohu pre $N \leq 100$.

A-I-3 Fibonacciho sústava

Fibonacciho postupnosť vyzerá nasledovne:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$$

Zostrojiť ju vieme tak, že prvé dva členy sú 0 a 1, každý nasledujúci je súčtom dvoch predchádzajúcich. Matematicky zapísané:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_{n+2} &= F_{n+1} + F_n \qquad (\forall n \geq 0) \end{aligned}$$

Pozrime sa teraz na to, ako fungujú pozičné číselné sústavy. Pozičná číselná sústava je určená dvoma údajmi: množinou používaných cifier a postupnosťou, ktorá udáva pre každú pozíciu hodnotu, ktorou sa príslušná cifra násobí.

Napríklad naša desiatková sústava používa množinu cifier $\{0, 1, 2, \ldots, 9\}$ a jednotlivé pozície v čísle majú hodnoty (sprava doľava) 1, 10, 100, 1000, ...

Fibonacciho číselná sústava je pozičná číselná sústava, ktorá používa len cifry 0 a 1 a v ktorej sú hodnoty jednotlivých pozícií Fibonacciho čísla (začínajúc od $F_2 = 1$). Teda napr. zápis 10011 vo Fibonacciho sústave predstavuje číslo $1 \cdot 8 + 0 \cdot 5 + 0 \cdot 3 + 1 \cdot 2 + 1 \cdot 1 = 11$.

Na rozdiel od bežných sústav, vo Fibonacciho sústave nemajú niektoré čísla jednoznačný zápis. Napr. aj zápis 10100 predstavuje číslo 11.

Súťažná úloha:

a) *(3 body)* Zápis čísla vo Fibonacciho sústave voláme *pekný*, ak sa v ňom nevyskytujú dve po sebe idúce jednotky. Dokážte, že každé prirodzené číslo má vo Fibonacciho sústave práve jeden pekný zápis. Napíšte program, ktorý zo vstupu načíta prirodzené číslo N a vypíše jeho pekný zápis.

b) *(7 bodov)* Napíšte program, ktorý pre dané k, A a B spočíta, koľko čísel z množiny $\{A, A + 1, \ldots, B\}$ má v peknom zápise práve k jednotiek.

Formát vstupu:

V podúlohe a) je na vstupe jediné prirodzené číslo N ($1 \leq N \leq 1\,000\,000\,000$).

V podúlohe b) sú na vstupe tri prirodzené čísla k, A a B ($1 \leq k \leq 30$, $1 \leq A < B \leq 1\,000\,000\,000$).

Formát výstupu:

V podúlohe a) vypíšte jeden riadok a v ňom reťazec núl a jednotiek, predstavujúci pekný zápis čísla N.

V podúlohe b) vypíšte jeden riadok a v ňom jedno celé číslo – počet čísel v zadanom intervale, ktoré majú v peknom zápise práve k jednotiek.

Príklady pre podúlohu a):

Vstup	Výstup
11	10100

Vstup	Výstup
174591	10100010000000000100010010

Príklady pre podúlohu b):

Vstup	Výstup
1 4 13	3

Pre $k = 1$ je výstupom počet Fibonacciho čísel v danom rozsahu. V zadanom rozsahu ležia Fibonacciho čísla 5, 8 a 13.

Vstup	Výstup
2 4 13	6

Vstup	Výstup
10 102000 103000	86

Pomalšie riešenia:
Maximálne 2 body za časť b):

Riešte pôvodnú úlohu navyše s podmienkou $|B - A| \leq 100\,000$.

A-I-4 Prekladacie stroje

V tomto ročníku olympiády sa budeme v teoretickej úlohe stretávať s prekladacími strojmi. V študijnom texte uvedenom za zadaním tejto úlohy sú tieto stroje popísané.

Súťažná úloha:

a) *(1 bod)* Všimnite si prekladacie stroje B a C z príkladov v študijnom texte. Tieto dva stroje zjavne robia preklad „opačnými smermi“. Dalo by sa preto očakávať, že platí nasledujúce tvrdenie:

Nech M je ľubovoľná (aj nekonečná) množina, ktorej každý prvok je nejaký reťazec písmen a, e a i. Potom $C(B(M)) = M$.

Slovne: Keď zoberieme množinu M, preložíme ju pomocou B a výsledok preložíme pomocou C, dostaneme pôvodnú množinu.

Ak toto tvrdenie naozaj platí, dokážte ho. Ak nie, nájdite protipríklad.

b) *(1 bod)* To isté ako v predchádzajúcej podúlohe, len M obsahuje reťazce tvorené znakmi ▬ a • a zaujíma nás, či musí platiť $B(C(M)) = M$.

c) *(3 body)* Hovoríme, že reťazec je zaujímavý, ak obsahuje len písmená a a b, pričom písmen a je dvakrát viac ako písmen b. Nech X je množina všetkých zaujímavých reťazcov. Teda napr. $aaabab \in X$, ale $baba \notin X$.

Nech Y je množina všetkých reťazcov, ktoré obsahujú najskôr niekoľko písmen a a za nimi trikrát toľko písmen b. Teda napr. $abbb \in Y$, ale $aaabab \notin Y$.

Zostrojte prekladací stroj, ktorý preloží X na Y.

d) *(5 bodov)* Na začiatku máme množinu M_1, ktorá obsahuje práve všetky reťazce z písmen a, b, ktoré obsahujú rovnako veľa písmen a ako b. Teda napr. $abbbaa \in M_1$, ale $aabab \notin M_1$.

Nové množiny môžeme zostrojovať nasledovnými operáciami:
– prekladom nejakej už zostrojenej množiny nejakým strojom (môžeme použiť zakaždým iný stroj)
– zjednotením dvoch už zostrojených množín
– prienikom dvoch už zostrojených množín

Na čo najmenej operácií zostrojte množinu G, ktorá obsahuje práve všetky reťazce, kde je najskôr niekoľko písmen a, potom toľko isto b, a nakoniec toľko isto c. Teda napr. $aabbcc \in G$, ale $abcc \notin G$, a ani $bac \notin G$.

Študijný text

Prekladací stroj je stroj, ktorý dostáva ako vstup reťazec znakov. Tento reťazec postupne číta a podľa vopred zvolenej sady pravidiel (teda svojho programu) občas nejaké znaky zapíše na výstup. Po tom, ako stroj spracuje celý vstupný reťazec a úspešne skončí výpočet, zoberieme reťazec znakov zapísaný na výstup a nazveme ho **prekladom** vstupného reťazca.

Výpočet stroja nemusí byť jednoznačne určený. Inými slovami, sada pravidiel môže niekedy stroju umožniť rozhodnúť sa. V takomto prípade sa môže stať, že niektorý reťazec bude mať viac rôznych prekladov.

Naopak, môže sa stať, že v nejakej situácií sa podľa danej sady pravidiel nebude dať v preklade pokračovať vôbec. V takomto prípade sa môže stať, že niektorý reťazec nebude mať vôbec žiadny preklad.

Formálnejšia definícia prekladacieho stroja:

Každý prekladací stroj pracuje nad nejakou vopred zvolenou konečnou množinou znakov. Túto budeme volať **abeceda** a značiť Σ. V súťažných úlohách bude vždy Σ známa zo zadania úlohy. Abeceda nikdy nebude obsahovať znak \$, ten budeme používať na označenie konca vstupného reťazca.

Stroj si môže počas prekladu reťazca pamätať konečne veľkú informáciu. Formálne toto definujeme tak, že stroj sa v každom okamihu prekladu nachádza v jednom z *konečne veľa* stavov. Nutnou súčasťou programu prekladacieho stroja je teda nejaká konečná **množina stavov**, v ktorých sa môže nachádzať. Túto množinu označíme K. Okrem samotnej množiny stavov je taktiež potrebné uviesť, v ktorom stave sa stroj nachádza na začiatku každého prekladu. Tento stav budeme volať **začiatočný stav**.

Program stroja sa bude skladať z konečného počtu prekladových pravidiel. Každé pravidlo je štvorica (p, u, v, q), kde $p, q \in K$ sú nejaké dva stavy a u, v sú nejaké dva reťazce znakov z abecedy Σ.

Stavy p a q môžu byť aj rovnaké. Reťazec u môže ako svoj jediný znak obsahovať znak \$. Reťazce u a v môžu byť aj rovnaké a môžu byť aj prázdne. Aby sa program ľahšie čítal, budeme namiesto prázdneho reťazca písať ε.

Význam takéhoto pravidla je nasledujúci: „Ak je stroj práve v stave p a neprečítaná časť vstupu začína reťazcom u, môže tento reťazec zo vstupu prečítať, na výstup zapísať reťazec v a zmeniť stav na q.“ Všimnite si, že pravidlo tvaru (p, ε, v, q) môžeme použiť vždy, keď sa stroj nachádza v stave p, bez ohľadu na to, aké znaky ešte zostávajú na vstupe.

Ešte potrebujeme povedať, kedy preklad úspešne skončil. V prvom rade budeme vyžadovať, aby prekladací stroj dočítal celý vstupný reťazec. Okrem toho umožníme stroju „povedať“, či sa mu preklad podaril alebo nie. Toto spravíme tak, že niektoré stavy stroja nazveme **ukončovacie stavy**. Množinu ukončovacích stavov budeme značiť F.

Úplne formálna definícia prekladacieho stroja:

Prekladací stroj je usporiadaná pätica (K, Σ, P, q_0, F), kde Σ a K sú *konečné* množiny, $q_0 \in K$, $F \subseteq K$ a P je *konečná* množina prekladových pravidiel popísaných vyššie. Presnejšie, nech Σ^* je množina všetkých reťazcov tvorených znakmi zo Σ, potom P je *konečná* podmnožina množiny $K \times (\Sigma^* \cup \{\$\}) \times \Sigma^* \times K$.

(Pre každé $q \in K$ budeme množinu pravidiel, ktorých prvá zložka je q, volať „prekladové pravidlá zo stavu q“.)

Ak teda chceme definovať konkrétny prekladací stroj, musíme uviesť všetkých päť vyššie uvedených objektov.

Keď už máme definovaný konkrétny stroj $A = (K, \Sigma, P, q_0, F)$, môžeme definovať, ako tento stroj prekladá konkrétny reťazec. Povieme najskôr formálnu definíciu, potom ju slovne vysvetlíme.

Množina **platných prekladov** reťazca u prekladacím strojom A je:

$$A(u) = \Big\{ \quad v \quad \Big| \quad \begin{array}{l} \exists n \geq 0; \quad \exists (p_1, u_1, v_1, r_1), \ldots, (p_n, u_n, v_n, r_n) \in P; \\ (\forall i \in \{1, \ldots, n-1\};\ r_i = p_{i+1}) \ \wedge \ p_1 = q_0 \ \wedge \ r_n \in F \ \wedge \\ \wedge \ \exists k \geq 0;\ u_1 u_2 \ldots u_n = u \underbrace{\$ \ldots \$}_{k} \ \wedge \ v_1 v_2 \ldots v_n = v \end{array} \Big\}$$

Definícia hovorí, kedy je reťazec v prekladom reťazca u. Vysvetlíme slovne význam jednotlivých riadkov definície:

- Prvý riadok hovorí, že aby sa dalo u preložiť na v, musí existovať nejaká postupnosť prekladových pravidiel, ktorú pri tomto preklade použijeme. Zvyšné dva riadky popisujú, ako táto postupnosť musí vyzerať.

- Druhý riadok zabezpečuje, aby stavy v použitých pravidlách boli správne: Prvé pravidlo musí byť pravidlom zo začiatočného stavu, každé ďalšie pravidlo musí byť pravidlom z toho stavu, do ktorého sa stroj dostal použitím predchádzajúceho pravidla.

  Navyše stav, v ktorom bude stroj na konci výpočtu, musí byť ukončovací.

- Posledný riadok hovorí o reťazcoch, ktoré stroj pri použití dotyčných prekladových pravidiel prečíta a zapíše.

  Reťazec, ktorý pri použití týchto pravidiel stroj prečíta zo vstupu, musí byť naozaj zadaný reťazec u, prípadne sprava doplnený vhodným počtom znakov \$.

  Reťazec, ktorý stroj zapíše na výstup, musí byť presne reťazec v.

Na čo budeme používať prekladacie stroje?:

Nám budú prekladacie stroje slúžiť na preklad jednej množiny reťazcov na inú množinu reťazcov. Ak A je prekladací stroj a $M \subseteq \Sigma^*$ nejaká množina reťazcov, tak preklad množiny M strojom A je množina $A(M) = \bigcup_{u \in M} A(u)$.

Inými slovami, výslednú množinu $A(M)$ zostrojíme tak, že zoberieme všetky reťazce z M a pre každé z nich dáme do $A(M)$ všetky jeho platné preklady.

Príklad 1:

Majme abecedu $\Sigma = \{0, \ldots, 9\}$. Nech M je množina všetkých tých reťazcov, ktoré predstavujú zápisy kladných celých čísel v desiatkovej sústave. Zostrojíme prekladací stroj A, pre ktorý bude platiť, že prekladom tejto množiny M bude množina zápisov všetkých kladných celých čísel, ktoré sú deliteľné tromi.

Riešenie:

Najjednoduchšie bude jednoducho vybrať z M tie čísla, ktoré sú deliteľné tromi. Náš prekladací stroj bude kopírovať cifry zo vstupu na výstup, pričom si bude v stave pamätať, aký zvyšok po delení tromi dáva doteraz prečítané (a zapísané) číslo. Ak je po dočítaní vstupu v stave zodpovedajúcom zvyšku 0, prejde do ukončovacieho stavu.

Formálne A bude pätica $(K, \Sigma, P, 0, F)$, kde $K = \{0, 1, 2, end\}$, $F = \{end\}$ a prekladové pravidlá vyzerajú nasledovne:

$$P = \Big\{(x, y, y, z) \;\Big|\; x \in \{0,1,2\} \wedge y \in \Sigma \wedge z = (10x{+}y) \bmod 3\Big\} \cup \Big\{(0, \$, \varepsilon, end)\Big\}$$

Príklad 2:

Majme abecedu $\Sigma = \{a, e, i, \bullet, \text{━}\}$. Zostrojíme prekladací stroj B, pre ktorý bude platiť, že prekladom ľubovoľnej množiny M, ktorá obsahuje len reťazce z písmen a, e a i, bude množina tých istých reťazcov v morzeovke (bez oddeľovačov medzi znakmi). Zápisy našich písmen v morzeovke sú nasledujúce: a je $\bullet$ ━, e je $\bullet$ a i je $\bullet\bullet$.

Napríklad množinu $M = \{ae, eea, ia\}$ by náš stroj mal preložiť na množinu $\{\bullet$ ━ $\bullet, \bullet\bullet\bullet$ ━$\}$. (Všimnite si, že reťazce eea a ia majú v morzeovke bez oddeľovačov rovnaký zápis.)

Riešenie:

Prekladací stroj B bude jednoducho čítať vstupný reťazec po znakoch a zakaždým zapíše na výstup kód prečítaného znaku.

Formálne B bude pätica $(K, \Sigma, P, \heartsuit, F)$, kde $K = \{\heartsuit\}$, $F = \{\heartsuit\}$ a prekladové pravidlá vyzerajú nasledovne:

$$P = \{(\heartsuit, a, \bullet\!\!-, \heartsuit),\ (\heartsuit, e, \bullet, \heartsuit),\ (\heartsuit, i, \bullet\bullet, \heartsuit)\}$$

Všimnite si, že nepotrebujeme explicitne kontrolovať, či sme na konci vstupu. Totiž počas celého prekladu sme v ukončovacom stave, a teda akonáhle prečítame posledný znak zo vstupu, je zapísaný preklad platný.

Príklad 3:

Majme abecedu $\Sigma = \{a, e, i, \bullet, -\}$. Zostrojíme prekladací stroj C, pre ktorý bude platiť, že prekladom ľubovoľnej množiny M, ktorá obsahuje len reťazce zo znakov $\bullet$ a $-$, bude množina **všetkých** reťazcov z písmen a, e a i, ktoré keď zapíšeme v morzeovke (bez oddeľovačov medzi znakmi), dostaneme reťazec z M. Napr. množinu $M = \{\bullet - \bullet, \bullet\bullet\bullet -\}$ by náš stroj mal preložiť na $\{ae, eea, ia\}$.

Riešenie:

Nášmu prekladaciemu stroju dáme možnosť sa v každom okamihu prekladu rozhodnúť, že ide čítať kód nejakého písmena a zapísať na výstup toto písmeno. Potom každej možnosti, ako rozdeliť vstupný reťazec na kódy písmen, bude zodpovedať jeden platný preklad.

Formálne C bude pätica $(K, \Sigma, P, \diamondsuit, F)$, kde $K = \{\diamondsuit\}$, $F = \{\diamondsuit\}$ a prekladové pravidlá $P = \{(\diamondsuit, \bullet\!\!-, a, \diamondsuit),\ (\diamondsuit, \bullet, e, \diamondsuit),\ (\diamondsuit, \bullet\bullet, i, \diamondsuit)\}$.

Ako príklad si ukážeme, ako mohol prebiehať preklad reťazcov z vyššie uvedenej množiny M. Sú tieto tri možnosti:

$(\diamondsuit, \bullet\!\!-, a, \diamondsuit), (\diamondsuit, \bullet, e, \diamondsuit)$
$(\diamondsuit, \bullet\bullet, i, \diamondsuit), (\diamondsuit, \bullet\!\!-, a, \diamondsuit)$
$(\diamondsuit, \bullet, e, \diamondsuit), (\diamondsuit, \bullet, e, \diamondsuit), (\diamondsuit, \bullet\!\!-, a, \diamondsuit)$

Ak by sme skúsili na ľubovoľnom vstupe z M používať prekladové pravidlá v inom poradí – napr. na vstupe $\bullet\bullet\bullet -$ použiť trikrát pravidlo $(\diamondsuit, \bullet, e, \diamondsuit)$ – nepodarí sa nám dočítať vstupný reťazec do konca.

Príklad 4:

Majme abecedu $\Sigma = \{a, b, c\}$. Nech množina X obsahuje práve všetky tie reťazce, v ktorých je rovnako a a b. Teda napr. $abbccac \in X$, ale $cbaa \notin X$.

Nech množina Y obsahuje práve všetky tie reťazce, ktoré neobsahujú žiadne a, neobsahujú podreťazec bc, a písmen c je dvakrát toľko ako písmen b. Teda napr. $ccccbb \in Y$, ale $cccbcb \notin Y$ a $acacba \notin Y$.

Zostrojíme prekladací stroj D, ktorý preloží X na Y.

Riešenie:

Budeme prekladať len niektoré vhodné reťazce z množiny X, presnejšie to budú tie, v ktorých nie sú žiadne c a všetky a idú pred všetkými b. Takýto reťazec preložíme tak, že najskôr každé a prepíšeme na cc, a potom skopírujeme na výstup všetky b. Teda napr. prekladom slova $aabb$ bude slovo $ccccbb$.

Formálne D bude pätica $(K, \Sigma, P, \textbf{čítaj-a}, F)$, kde $K = \{\textbf{čítaj-a}, \textbf{čítaj-b}\}$, $F = \{\textbf{čítaj-b}\}$ a prekladové pravidlá vyzerajú nasledovne:
$P = \{(\textbf{čítaj-a}, a, cc, \textbf{čítaj-a}),\ (\textbf{čítaj-a}, b, b, \textbf{čítaj-b}),\ (\textbf{čítaj-b}, b, b, \textbf{čítaj-b})\}$

Prečo tento prekladací stroj funguje? Ak vstupný reťazec obsahuje nejaké písmeno c, pri jeho prekladaní sa pri prvom výskyte c náš stroj zasekne. Preto takéto reťazce nemajú žiaden platný preklad. Podobne nemajú platný preklad reťazce, kde nie je dodržané poradie písmen a a b. Totiž akonáhle náš stroj prečíta prvé b, prejde do stavu **čítaj-b**, a ak sa teraz ešte na vstupe vyskytne a, stroj sa zasekne.

Platné preklady teda existujú naozaj len pre slová vyššie popísaného tvaru. Je zjavné, že z každého z nich vyrobíme nejaký reťazec z Y, preto $D(X) \subseteq Y$. A naopak, ak si povieme konkrétne slovo z Y, ľahko nájdeme slovo z X, ktoré sa naň preloží, preto $Y \subseteq D(X)$, a teda $Y = D(X)$.

Zadania domáceho kola kategórie B

B-I-1 O spájaní polí

Máme dve polia celých čísel. V nich máme uložených dokopy $M + N$ **kladných** celých čísel.

Prvé pole sa volá A a obsahuje $M + N$ prvkov. Prvých M čísel v ňom je kladných, na ostatných N pozíciách sú nuly (ktoré predstavujú voľné miesto). Prvých M čísel je navyše utriedených vzostupne.

Druhé pole sa volá B a obsahuje N kladných čísel utriedených vzostupne.

Vašou úlohou bude naprogramovať čo najefektívnejší algoritmus, ktorý presunie všetkých $M + N$ **kladných** čísel do poľa A, a to navyše tak, aby pole A bolo na konci vzostupne utriedené.

Príklad:

na začiatku		*na konci*	
pole A:	1, 4, 5, 7, 0, 0, 0	pole A:	1, 2, 4, 5, 5, 7, 10
pole B:	2, 5, 10		

Súťažná úloha:

Napíšte vo vami zvolenom programovacom jazyku procedúru/funkciu, ktorá dostane ako parametre polia A a B (a prípadne aj čísla M a N), a upraví obsah poľa A podľa zadania.

Dôležité obmedzenie: Okrem týchto dvoch polí už smiete použiť len konečný počet celočíselných premenných. Inými slovami, je zakázané používať pomocné polia.

B-I-2 Krtkovou norou

Kdesi v okolí Brna žijú prúdoví krtkovia. Žijú pod zemou, majú tam svoje brlôžky a medzi nimi vyrazené chodbičky. Chodbičky sú rôzne dlhé, priamo úmerne tomu, koľko dní danú chodbičku krtkovia razili. A že sú krtkovia prúdoví, chodbičky razia rýchlo, a teda im to nikdy dlho netrvá.

V jednom z brlôžkov býva šéf, krtek Vítek. Ten rád chodí na návštevy k ostatným krtkom. Zaujímalo by ho teraz, ku komu to vlastne má najďalej.

Súťažná úloha:

Napíšte program, ktorý načíta popis siete brlôžkov, a zistí, kam to má krtek Vítek zo svojho brlôžka najďalej.

Príklad:

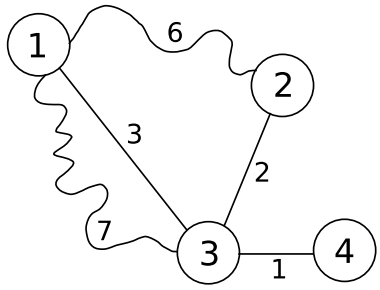
Vstup

```
4 5
1 2 6
1 3 7
1 3 3
2 3 2
3 4 1
```

Najkratšia cesta do brlôžkov 2 aj 4 vedie cez brlôžok 3.

výstup

```
2
```



Formát vstupu a výstupu:

V prvom riadku vstupu sú dve celé čísla N a M ($1 \leq N \leq 1\,000$, $0 \leq M \leq 100\,000$), kde N je počet brlôžkov a M počet chodbičiek medzi nimi. Brlôžky sú očíslované od 1 do N. Krtek Vítek býva (ako správny šéf) v brlôžku číslo 1.

Nasleduje M riadkov. V každom z nich sú tri kladné celé čísla a, b a d, kde a a b sú čísla brlôžkov spojených chodbičkou a d udáva, koľko dní krtkovia chodbičku razili.

Všimnite si, že d je celé číslo. Keďže krtkovia sú prúdoví a razia chodby rýchlo, môžete predpokladať, že $d \leq 7$. Navyše môžete predpokladať, že sieť chodbičiek je súvislá – teda že sa krtek Vítek vie dostať do hociktorého brlôžka.

Na výstup vypíšte jeden riadok a v ňom medzerami oddelené čísla všetkých brlôžkov, ktoré sú od brlôžka číslo 1 najďalej.

B-I-3 Kontrola XML

So skratkou XML (eXtensible Markup Language) ste sa už pravdepodobne stretli. Zjednodušene môžeme povedať, že ide o akýsi univerzálny formát na uloženie a prenos dát akéhokoľvek druhu.

Dáta v XML dokumente môžu byť hierarchicky rozčlenené. Na toto slúžia tzv. *tagy* – značky, ktorými vyznačíme časť XML dokumentu.

Ukážeme si teraz príklad XML dokumentu:

```
<kviz>
  toto je text len tak
  <zaznam>
    <otazka>Kolky rocnik OI prave prebieha?</otazka>
    <odpoved>23</odpoved>
  </zaznam>
  <zaznam><otazka>2+2?</otazka><odpoved>4</odpoved></zaznam>
</kviz>
```

Tagy sú uzavreté v špicatých zátvorkách. Tento dokument teda obsahuje napr. tagy `<kviz>`, `<otazka>` a `</kviz>`.

Všimnite si, že tagy musia byť *párové* – jeden (otvárací) na začiatku vyznačenej oblasti, druhý (zatvárací) na jej konci. Zatvárací tag má vždy rovnaké meno ako otvárací a navyše začína znakom `/`.

Tiež si všimnite, že tagy sú v sebe vnorené – keď napríklad vo vnútri tagu `<zaznam>` otvoríme tag `<otazka>`, musíme najskôr použiť zatvárací tag `</otazka>`, až potom môžeme použiť tag `</zaznam>`.

Súťažná úloha:

Vašou úlohou bude napísať program, ktorý dostane na vstupe text a skontroluje, či je to korektný XML dokument.

Formálnejšie si definujeme, ako (pre účely tejto úlohy) vyzerá korektný XML dokument. Dokument sa skladá z tagov a textu medzi nimi. Text medzi tagmi neobsahuje znaky `<` a `>`. Názov každého tagu je tvorený len malými písmenami, a to aspoň jedným a nanajvýš ôsmimi.

Za každým otváracím tagom musí nasledovať jemu zodpovedajúci zatvárací, pričom musí byť dodržané vnáranie tagov do seba. (Inými slovami, pre ľubovoľný otvárací tag musí platiť, že keď zoberieme všetky tagy a text za ním, až po jemu zodpovedajúci zatvárací tag, dostaneme opäť korektný XML dokument.)

Formát vstupu a výstupu:

Vstup obsahuje niekoľko (nanajvýš 1000) riadkov textu, každý z nich obsahuje najviac 100 znakov.

Na výstup vypíšte text „`ano`“ alebo „`nie`“ podľa toho, či text na vstupe predstavuje korektný XML dokument.

Príklady:

Vstup

```
<srnka><paroh>hnedy</paroh>
</srnka><jelen></jelen>
```

Výstup

```
ano
```

Vstup

```
<a> <b> </a> </b>
```

Výstup

```
nie
```

Tagy nie sú v sebe vnorené.

Vstup

```
<body> <zly/zly tag> </body>
```

Výstup

```
nie
```

Nekorektný tag.

Vstup

```
<p><data>47</data> 94 <b><q>
</q> <q>q</q> <a><cc></cc>
</a></b> nejaky-text
</p> a este aj na konci
```

Výstup

```
ano
```

Vstup

```
a < b + c
```

Výstup

```
nie
```

Znak `<` *nesmie byť použitý.*

Vstup

```
<a> <b> <b> </b>
</b> </a> </a>
```

Výstup

```
nie
```

Posledný zatvárací tag nemá zodpovedajúci otvárací.

Vstup

```
<srnka><paroh></paroh>
```

Výstup

```
nie
```

Tag `<srnka>` *zostal otvorený.*

B-I-4 Rákosie

V tomto ročníku olympiády sa budeme v teoretickej úlohe stretávať so zvláštnym druhom rákosia. V študijnom texte uvedenom za zadaním tejto úlohy je popísané, ako toto rákosie vyzerá.

Súťažná úloha:

Nájdite nejakú odrodu rákosia, ktorej mŕtve steblá sú tvorené len bunkami a, a navyše počet týchto buniek je mocnina dvoch.

Vo vašej odrode teda musia ísť „vyrobiť“ steblá a, aa, $aaaa$, a^8, a^{16}, ... Naopak, nesmú sa dať vyrobiť steblá aaa ani abc.

Študijný text

Programátorské rákosie je zvláštna rastlina. Každý exemplár sa skladá z niekoľkých buniek, ktoré rastú v rade za sebou. Každá bunka je buď živá, alebo mŕtva. Živé bunky budeme značiť veľkými písmenami abecedy, mŕtve bunky malými. Keďže bunky rastú v rade, každé steblo rákosia vieme jednoducho popísať tak, že vypíšeme zaradom (od koreňa) typy jeho buniek.

Každé steblo rákosia sa počas svojho života vyvíja, až do okamihu, keď už obsahuje samé mŕtve bunky. Počas tohto vývoja sa môže meniť počet aj druh buniek, ktoré obsahuje.

Existuje veľa rôznych odrôd rákosia. Každá odroda má v génoch „naprogramovanú“ svoju sadu pravidiel, podľa ktorých sa steblá danej odrody môžu meniť.

Mŕtve bunky sú pekné farebné. Pestovatelia preto často šľachtia rôzne odrody, ktoré by vytvárali z mŕtvych buniek rôzne zaujímavé vzory.

Formálnejšie definície:

Steblo rákosia je ľubovoľná (aj prázdna) postupnosť veľkých a malých písmen, predstavujúcich živé a mŕtve bunky. Ak steblo neobsahuje žiadne živé bunky, hovoríme, že je to mŕtve steblo.

Aby sme prázdnu postupnosť buniek videli, budeme ju značiť ε. Ak ide v zápise stebla k rovnakých písmen x za sebou, môžeme to skrátene zapísať x^k. Teda napríklad $baCDGk^3v^{47}XX$ je steblo rákosia, ktoré je tvorené 57 bunkami, z nich je 5 živých.

Všimnite si, že steblo vypisujeme vždy od koreňa nahor. Teda steblo abc je iné ako steblo cba.

Genetické pravidlo je pravidlo, ktoré určuje jednu možnú zmenu stebla. Každé pravidlo je tvaru: „ak sa na steble vyskytne postupnosť buniek *vstup*, môže sa zmeniť na postupnosť buniek *výstup*“. Postupnosti *vstup* aj *výstup* môžu obsahovať ľubovoľne veľa (aj nula) živých aj mŕtvych buniek. Jediné obmedzenie je, že *vstup* musí obsahovať aspoň jednu živú bunku.

Pravidlo zapisujeme *vstup* $\rightarrow$ *výstup*. Ak máme viac pravidiel, ktoré majú rovnaký vstup, môžeme ich skrátene zapísať ako
vstup $\rightarrow$ *výstup*$_1$ | *výstup*$_2$ | *výstup*$_3$.

Napríklad zo stebla $aBacaBd$ sa použitím pravidla $aB \rightarrow X$ môže stať buď steblo $XacaBd$ alebo steblo $aBacXd$.

Použitie pravidla budeme značiť $\Rightarrow$. (Všimnite si, že ide o dvojitú šípku, zatiaľ čo na zápis pravidla používame jednoduchú.) Teda napríklad druhú zmenu z predchádzajúceho príkladu by sme zapísali $aBacaBd \Rightarrow aBacXd$.

Keď vznikne nové steblo rákosia, je vždy tvorené práve jednou bunkou Z. **Odroda rákosia** je jednoznačne určená *konečnou* sadou genetických pravidiel. Tá jednoznačne popisuje, aké všetky druhy stebiel môžu zo začiatočnej bunky Z narásť.

Súbor vzorov, ktoré vie odroda rákosia vyrobiť, je množina *všetkých mŕtvych* stebiel, ktoré môžu v rákosí danej odrody narásť.

Príklad 1:

Pestovateľ Ferko chcel, aby mu na rybníku vyrástlo rákosie, ktoré bude mať pekné strakaté steblá. Presnejšie, chcel, aby súbor vzorov jeho rákosia tvorili práve *všetky* steblá, na ktorých sa striedajú bunky a a b.

Ferko teda chce odrodu, v ktorej môžu narásť mŕtve steblá *ababa*, *abab* aj *babab*, ale nie *aab* ani *klmn*.

Riešenie príkladu 1:

Existuje veľa rôznych odrôd, ktoré majú túto vlastnosť. Ferko vyšľachtil odrodu, ktorá mala nasledujúce genetické pravidlá:

$$Z \rightarrow X, \quad Z \rightarrow bX, \quad X \rightarrow abX, \quad X \rightarrow \varepsilon \quad \text{a} \quad X \rightarrow a.$$

Ukážeme, ako môže narásť niekoľko dobrých stebiel. Ku každému použitiu pravidla $\Rightarrow$ napíšeme pre názornosť poradové číslo pravidla, ktoré sme použili. Teda napríklad $\Rightarrow_3$ znamená, že bolo použité pravidlo $X \rightarrow abX$.

$$\begin{array}{llllllll} Z & \Rightarrow_1 & X & \Rightarrow_3 & abX & \Rightarrow_3 & ababX & \Rightarrow_3 \quad ababa \\ Z & \Rightarrow_1 & X & \Rightarrow_3 & abX & \Rightarrow_3 & ababX & \Rightarrow_4 \quad abab \\ Z & \Rightarrow_2 & bX & \Rightarrow_3 & babX & \Rightarrow_3 & bababX & \Rightarrow_4 \quad babab \end{array}$$

Všimnite si, že použitie pravidla 4 zmaže zo stebla bunku X.

Príklad 2:

Pestovatelia Jožko a Mirko chceli, aby ich rákosie „vyrábalo“ mŕtve steblá, ktoré budú mať rovnako veľa buniek a aj b (a žiadne iné), ale na rozdiel od Ferka nech tieto bunky môžu byť ľubovoľne rozmiestnené.

Chcú teda odrodu, kde môžu narásť steblá ab, $abba$ aj $ba^{47}b^{46}$, ale nie $ababa$ ani bac.

Riešenie príkladu 2:

Jožko vyšľachtil rákosie s pravidlami:

$$Z \to aZbZ \mid bZaZ \mid \varepsilon.$$

Mirko vyšľachtil rákosie s pravidlami:

$$Z \to ABZ \mid \varepsilon, \quad AB \to BA, \quad BA \to AB, \quad A \to a, \quad B \to b.$$

Hlavný rozdiel medzi týmito dvoma odrodami je v tom, že Mirkovi sa bude ľahšie presviedčať návštevy, že jeho rákosie naozaj „funguje“.

V prípade Mirkovho rákosia je to naozaj jednoduché. Použitím prvého pravidla k-krát dostaneme steblo, na ktorom je k živých buniek A a k živých buniek B. Použitím pravidiel 2 a 3 sa tieto bunky môžu ľubovoľne premiešať a keď následne použijeme pravidlá 4 a 5, vyhrali sme.

U Jožkovho rákosia je zjavné, že počty a a b budú rovnaké, ale vieme naozaj vyrobiť všetky možné dobré steblá? Matematickou indukciou podľa dĺžky stebla dokážeme, že áno.

Steblá dĺžky 0 (ε) aj 2 (ab a ba) vyrobiť vieme. Nech teraz vieme vyrobiť všetky dobré steblá dĺžky $\leq 2n$. Zoberme si ľubovoľné dobré steblo dĺžky $2n+2$. Bez ujmy na všeobecnosti nech začína bunkou a. Budeme postupne čítať steblo a počítať si, o koľko viac a ako b sme videli. Na začiatku je táto hodnota 1 (začíname bunkou a), na konci bude 0 (lebo a a b v celom steble je rovnako).

Všimnime si okamih, kedy prvýkrát klesne táto hodnota na 0. Muselo sa to stať tak, že sme práve prečítali nejaké b. Napríklad pre steblo $aabaabbabbbaab$ sa tak stane po prečítaní desiateho znaku, pre $aabb$ po štvrtom.

Teraz naše slovo môžeme schematicky zapísať ako $a\beta b\gamma$, kde b je to b, ktoré sme si práve našli. Vieme, že v časti $a\beta b$ je rovnako veľa a aj b. Preto musí byť rovnako veľa a aj b v časti β, a teda aj v časti γ. Podľa indukčného predpokladu vieme teda zo Z vyrobiť aj β, aj γ. No a celé naše steblo teda vyrobíme tak, že najskôr použijeme pravidlo $Z \to aZbZ$, potom z prvého Z vyrobíme β, a nakoniec z druhého Z vyrobíme γ.

Zadania krajského kola kategórie A

A-II-1 Parkovanie kočov

Kráľ Drozdia brada vydáva dcéru. Pri tej sláve (a jedle zdarma) nemôže chýbať nijaký šľachtic z okolia. A ako sa tak šľachtici na svojich kočoch vezú na svadbu, vôbec netušia, že sluhom kráľa Drozdej brady spôsobia zaujímavý problém.

Všetky koče totiž treba zaparkovať, a to nie len tak hocijako. Koče treba parkovať do radov. A aby kráľovi nezničili trávnik, musí tých radov byť čo najmenej.

Dvorná etiketa káže, že keď budú hostia odchádzať, musia odchádzať zoradení podľa dôležitosti, najdôležitejší hosť ako posledný.

A to ešte nie je všetko. Koče, ktoré stoja v tom istom rade, musia samozrejme odchádzať v poradí, v akom stoja. A aby predišli zrážkam kočov, chcú ich sluhovia zaparkovať tak, aby pri odchádzaní vždy najskôr odišiel celý jeden rad, až potom začal odchádzať ďalší, a tak ďalej.

Súťažná úloha:

Sluhovia presne vedia poradie, v akom budú prichádzať hostia, aj dôležitosť každého z nich. Keď parkujú koč, môžu ho zaparkovať *na začiatok alebo na koniec* hociktorého z už stojacich radov, prípadne ho postaviť do nového radu. Koče musia sluhovia zaparkovať v poradí, v akom hostia prichádzajú.

Nájdite najmenší počet radov, ktorý stačí na to, aby po správnom zaparkovaní vedeli koče odísť v predpísanom poradí.

Formát vstupu:

Vstup začína celým číslom N ($1 \leq N \leq 100\,000$) – počet hostí. Nasleduje N *rôznych* kladných celých čísel d_i ($1 \leq d_i \leq 10^9$), udávajúcich dôležitosť hostí v poradí, v akom prichádzajú (väčšie číslo je dôležitejší hosť).

Formát výstupu:

Výstup má obsahovať jediný riadok a v ňom jediné celé číslo – potrebný počet radov.

Príklady:

Vstup

```
10
10 9 11 12 13 8 14 7 6 100
```

Výstup

```
1
```

V tomto prípade sa stačí držať pravidla „koče menej dôležité ako 10 idú na začiatok, ostatné na koniec“.

Vstup

```
6
12 17 9 23 16 14
```

Výstup

```
2
```

Jedno možné riešenie: Koče 12 a 17 pôjdu do rôznych radov. Koč 9 postavíme pred koč 12, koč 23 za koč 17, koč 16 pred koč 17 a nakoniec koč 14 pred koč 16.

Vstup

```
12
1 3 2 5 4 7 6 9 8 11 10 12
```

Výstup

```
6
```

V jedinom optimálnom riešení budú po zaparkovaní rady: (1 2), (3 4), (5 6), (7 8), (9 10) a (11 12).

Pomalšie riešenia:
Maximálne 8 bodov: Riešte tú istú úlohu pre $N \leq 1000$.
Maximálne 6 bodov: Riešte tú istú úlohu pre $N \leq 100$.
Maximálne 4 body: Riešte tú istú úlohu pre $N \leq 10$.

A-II-2 Dlhopisy

Kleofáš nedávno zdedil po bohatej tetičke Anastázii kopu peňazí. Nevedel, čo s nimi, tak sa rozhodol investovať ich do dlhopisov.

Pre jednoduchosť budeme predpokladať nasledujúce skutočnosti:

- Každý typ dlhopisu má svoju pevnú cenu, rovnakú pri kúpe aj predaji.
- Každý typ dlhopisu má pevne daný ročný výnos.

- Každého typu dlhopisu sa dá nakúpiť ľubovoľne veľa.

Uvažujme napríklad nasledujúcu situáciu: Banka ponúka dva typy dlhopisov: Dlhopisy za 4000 korún s ročným výnosom 400 a dlhopisy za 3000 korún s ročným výnosom 250.

Ak má Kleofáš 10 000 korún, najlepšie, čo s nimi môže spraviť, je kúpiť dva dlhopisy po 3000 a jeden za 4000, čím dostane ročný výnos 900 korún.

Keď prejdu dva roky a Kleofáš dostane dvakrát výnosy, bude mať dokopy kapitál 11 800 korún. V tejto chvíli sa mu oplatí jeden dlhopis za 3000 predať a namiesto neho kúpiť dlhopis za 4000. Po treťom roku takto jeho kapitál narastie na 12 850.

Súťažná úloha:

Napíšte program, ktorý načíta Kleofášov začiatočný kapitál, ceny a výnosy dlhopisov a počet rokov, a spočíta, koľko najviac peňazí vie mať Kleofáš po uplynutí daného počtu rokov.

Formát vstupu:

V prvom riadku je celé číslo K ($1 \leq K \leq 1\,000\,000$) udávajúce Kleofášov štartovný kapitál.

V druhom riadku je počet typov dlhopisov D ($1 \leq D \leq 100$).

V treťom riadku je D dvojíc celých čísel c_i, v_i predstavujúcich ceny a výnosy dlhopisov ($0 < c_i \leq 10^9$, $0 \leq v_i \leq c_i/10$, c_i je násobkom $T = 1000$).

V poslednom riadku je počet rokov R ($1 \leq R \leq 40$).

Časovú zložitosť svojho algoritmu vyjadrite pomocou K, D, T a R.

Formát výstupu:

Vypíšte jeden riadok a v ňom maximálnu hodnotu Kleofášovho kapitálu po R rokoch. (Môžete predpokladať, že sa táto hodnota zmestí do bežnej celočíselnej premennej.)

Príklady:

Vstup	Výstup
10000 2 4000 400 3000 250 4	14050

Príklad zo zadania. Vo štvrtom roku Kleofáš bude vlastniť 3 dlhopisy po 4000, čím zarobí ďalších 1200.

Vstup

```
100000
3
103000 9001 47000 7 83000 100
31
```

Výstup

```
112001
```

Kleofáš kúpi jeden dlhopis za 83 000. *Tým za 30 rokov zarobí 3000 korún. Na posledný rok si konečne môže kúpiť prvý dlhopis za* 103 000. *V poslednom roku teda zarobí ďalších* 9001.

Vstup

```
100000
3
103000 9001 47000 7 83000 100
37
```

Výstup

```
166014
```

Pokračovanie z predchádzajúceho príkladu. Po roku 36 Kleofáš dokúpi dlhopis za 47 000, *čím v poslednom roku zarobí o 7 korún viac.*

Pomalšie riešenia:
Maximálne 8 bodov: Riešte pôvodnú úlohu pre $D \leq 10$ a $R \leq 10$.
Maximálne 7 bodov: Riešte pôvodnú úlohu pre $D \leq 10$, $R \leq 10$, $c_i \leq 10^6$.
Maximálne 4 body: Riešte pôvodnú úlohu pre $K \leq 45\,000$ a $R = 1$.

A-II-3 O víťazovi turnaja

Samko sa rozhodol, že zorganizuje programátorský turnaj. Vyzval teda svojich priateľov, aby mu poslali programy, ktoré budú hrať piškvorky. Priatelia nelenili, programy poslali a Samko všetkých pozval na veľký turnaj.

Samko vymyslel pre svoj turnaj jednoduché pravidlá: V každom kole sa náhodne vyžrebujú dva programy, zahrajú si, a ten z nich, ktorý prehrá, z turnaja vypadne.

V noci pred turnajom Samka premohla zvedavosť a pozrel si súboje pre niektoré dvojice programov. Ráno však zistil, že sa tým pripravil o časť prekvapenia. Keďže všetky programy jeho priateľov sú deterministické (t. j. nepoužívajú pri rozhodovaní náhodu), dopadne súboj každých dvoch programov vždy rovnako. Na základe súbojov, ktoré v noci videl, teraz už Samko o niektorých programoch vedel, že turnaj vyhrať nemôžu.

Súťažná úloha:

Pre dané výsledky zápasov, ktoré si Samko v noci pozrel, zistite, ktoré programy ešte môžu turnaj vyhrať.

Formát vstupu:

V prvom riadku vstupu je jedno celé číslo N ($1 \leq N \leq 100\,000$) – počet programov v turnaji. Programy sú očíslované od 1 do N.

Nasleduje N riadkov, pričom i-ty z nich popisuje zápasy, ktoré vyhrá program i. Presnejšie, i-ty z týchto riadkov začína číslom d_i, ktoré hovorí, koľko zápasov program i v noci vyhral. Nasleduje d_i čísel programov, nad ktorými vyhral. Tieto čísla sú utriedené podľa veľkosti.

Označme $M = d_1 + \cdots + d_n$ počet hier, ktoré si Samko v noci pozrel. Môžete predpokladať, že $0 \leq M \leq 1\,000\,000$. Tiež môžete predpokladať, že vstup je korektný (ak nejaký program x vyhral nad y, tak y nevyhral nad x).

Formát výstupu:

Vypíšte jeden riadok a v ňom zoznam čísel programov, ktoré ešte môžu vyhrať turnaj.

Príklady:

Vstup

```
4
2  2 3
0
1  2
1  2
```

Výstup

```
1 3 4
```

Program 1 určite vyhrá nad programmi 2 a 3. Program 3 určite vyhrá nad programom 2. Program 4 určite vyhrá nad programom 2.

Ako príklad ukážeme, ako môže program 3 vyhrať turnaj. Jedna možnosť je, že v prvom kole vyhrá 3 nad 2, v druhom 4 nad 1 a nakoniec v treťom kole vyhrá 3 nad 4.

Vstup

```
5
2  2 3
0
1  2
1  2
0
```

Výstup

```
1 2 3 4 5
```

Tentokrát môže turnaj vyhrať ľubovoľný program.
Program 2 vie vyhrať napr. takto: V prvom kole 3 porazí 4, v druhom 5 porazí 3, v treťom 5 porazí 1 a vo štvrtom 2 porazí 5.

Pomalšie riešenia:
Maximálne 7 bodov: Riešte pôvodnú úlohu pre $N \leq 1\,000$.
Maximálne 5 bodov: Riešte pôvodnú úlohu pre $N \leq 100$.
Maximálne 4 body: Riešte pôvodnú úlohu pre $N \leq 20$.
Maximálne 3 body: Riešte pôvodnú úlohu pre $N \leq 10$.

A-II-4 Prekladacie stroje

Študijný text nájdete za zadaniami súťažnej úlohy A-I-4 z domáceho kola.

Súťažná úloha:

a) *(6 bodov)* Na začiatku máme množinu M_1, ktorá obsahuje práve všetky reťazce z písmen a, b, ktoré obsahujú rovnako veľa písmen a ako b. Teda napr. $abbbaa \in M_1$, ale $aabab \notin M_1$.

Nové množiny môžeme zostrojovať nasledovnými operáciami:
– prekladom nejakej už zostrojenej množiny nejakým strojom (môžeme použiť zakaždým iný stroj)
– zjednotením dvoch už zostrojených množín
– prienikom dvoch už zostrojených množín

Na čo najmenej operácií zostrojte množinu G, ktorá obsahuje práve všetky reťazce z písmen a, b, c, ktoré obsahujú rovnako veľa písmen každého druhu. Teda napr. $aabbcc \in G$, $bac \in G$, ale $abcc \notin G$.

b) *(4 body)* Množina X obsahuje desiatkové zápisy tých kladných celých čísel, ktoré obsahujú rovnako veľa cifier 1 a 2. Teda napr. $1122 \in X$, $21231 \in X$, $47 \in X$, ale $112 \notin X$ a $031221 \notin X$ (desiatkový zápis nemôže obsahovať nulu na začiatku).

Množina Y obsahuje desiatkové zápisy kladných celých čísel deliteľných 7. Teda napr. $140 \in Y$, $7707 \in Y$, ale $47 \notin Y$ a $07 \notin Y$.

Zostrojte prekladací stroj, ktorý preloží množinu X na Y, alebo dokážte, že takýto prekladací stroj neexistuje.

Zadania krajského kola kategórie B

B-II-1 Zberateľky

Danka a Janka sú dvojičky. Obe zbierajú cudzokrajné známky. Ešte donedávna mali obe presne rovnaké zbierky, každá tú svoju pekne spôsobne utriedenú vo svojom albume.

Lenže včera cestou zo školy našla Janka na zemi obálku s dvoma nádhernými známkami, ktoré ešte vo zbierke nemali. Danke ich však odmietla ukázať, a doma si ich rýchlo založila do svojho albumu.

Danku teraz zožiera zvedavosť – akéže to známky Janka má a ona nie? A tak, keď Janka odišla vyniesť smeti, Danka schytila oba albumy a začala nové známky hľadať.

Súťažná úloha:

Dané je celé číslo N a dve polia celých čísel $A[1 \ldots N]$ a $B[1 \ldots N+2]$. Polia A aj B sú utriedené v rastúcom poradí. Pole B obsahuje presne to isté ako pole A, a navyše dva prvky x, y (zaradené na správnom mieste). Všetky čísla v poli B (a teda aj v poli A) sú navzájom rôzne.

Napíšte procedúru/funkciu, ktorá prvky x a y čo najrýchlejšie nájde.

(Polia už existujú v pamäti pri zavolaní vašej procedúry, teda čas potrebný na ich načítanie nerátame do časovej zložitosti procedúry.)

Formát vstupu:

Vaša procedúra dostane ako parametre číslo N a polia A a B. (Prípadne, ak je vám to pohodlnejšie, považujte N, A a B za globálne premenné.)

Formát výstupu:

Vaša procedúra má vrátiť dve celé čísla – hodnoty prvkov x a y, ktoré sú v poli B navyše.

Príklady:

Vstup	Výstup
$N = 5$ $A = (1, 3, 4, 7, 110)$ $B = (1, 2, 3, 4, 7, 47, 110)$	2 47

Vstup	Výstup
$N = 4$ $A = (2, 3, 7, 110)$ $B = (2, 3, 7, 110, 111, 123)$	`111 123`

Hodnotenie riešení:
Celých 10 bodov môžu dostať len riešenia, ktoré majú lepšiu ako lineárnu časovú zložitosť, teda nájdu riešenie bez toho, aby videli väčšinu prvkov v poliach A a B.
Maximálne 6 bodov dostanú riešenia, ktoré vyriešia v lepšom ako lineárnom čase ľahšiu verziu úlohy: stačí, aby fungovali, ak čísla x a y nasledujú v poli B bezprostredne za sebou.
Maximálne 5 bodov dostanú riešenia (pôvodnej úlohy) s lineárnou časovou zložitosťou, teda také, ktorých čas behu je priamo úmerný číslu N.
Maximálne 3 body dostanú pomalšie riešenia.

B-II-2 Krtkovou norou tam a späť

Kdesi v okolí Brna žijú prúdoví krtkovia. Žijú pod zemou, majú tam svoje brlôžky a medzi nimi vyrazené chodbičky. Chodbičky sú rôzne dlhé, priamo úmerne tomu, koľko dní danú chodbičku krtkovia razili. A že sú krtkovia prúdoví, chodbičky razia rýchlo, a teda im to nikdy dlho netrvá.

Krtek Vítek býva v brlôžku 1. V brlôžku N býva jeho kamarátka, ku ktorej chodí na návštevu obzvlášť rád. A keďže sa k nej veľmi teší, ide k nej vždy najkratšou možnou cestou. To však neznamená, že ide vždy presne rovnakou cestou – najkratších ciest môže byť viac a Vítek si vždy zvolí, ktorou z nich pôjde.

Súťažná úloha:
Napíšte program, ktorý načíta popis siete brlôžkov, a nájde všetky brlôžky, ktoré môže Vítek cestou ku svojej kamarátke navštíviť.

Formát vstupu a výstupu:
V prvom riadku vstupu sú dve celé čísla N a M ($1 \leq N \leq 10\,000$, $0 \leq M \leq 100\,000$), kde N je počet brlôžkov a M počet chodbičiek medzi nimi. Brlôžky sú očíslované od 1 do N. Krtek Vítek býva v brlôžku číslo 1, jeho kamarátka v brlôžku N.

Nasleduje M riadkov. V každom z nich sú tri kladné celé čísla a, b a d, kde a a b ($1 \leq a, b \leq N$) sú čísla brlôžkov spojených chodbičkou a d ($1 \leq d \leq 7$) udáva, koľko dní krtkovia chodbičku razili.

Na výstup vypíšte jeden riadok a v ňom medzerami oddelené čísla všetkých brlôžkov, ktoré ležia na niektorej najkratšej ceste z 1 do N.

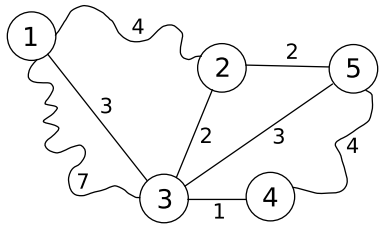
Príklad:

Vstup

```
5 8
1 2 4
1 3 7
1 3 3
2 3 2
2 5 2
3 4 1
3 5 3
4 5 4
```

Výstup

```
1 2 3 5
```



Najkratšie cesty sú dve: 1-2-5 a 1-3-5.

Hodnotenie riešení:
Plný počet bodov môžu získať riešenia, ktorých časová zložitosť závisí lineárne od veľkosti vstupu (od hodnoty $M + N$).
Riešenia, ktoré spravia počet krokov úmerný N^2, získajú najviac 8 bodov.
Riešenia, ktorých časová zložitosť rastie exponenciálne v závislosti od N, získajú najviac 4 body.

B-II-3 Opravovanie XML

Pripomeňme si najskôr, ako vyzerajú korektné XML dokumenty. Dokument sa skladá z tagov a textu medzi nimi. Za každým otváracím tagom musí nasledovať jemu zodpovedajúci zatvárací, pričom musí byť dodržané vnáranie tagov do seba. (Inými slovami, pre ľubovoľný otvárací tag musí platiť, že keď zoberieme všetky tagy a text za ním, až po jemu zodpovedajúci zatvárací tag, dostaneme opäť korektný XML dokument.)

Príklad XML dokumentu:

```
<kviz>
  toto je text len tak
```

```
  <zaznam>
    <otazka>Kolky rocnik OI prave prebieha?</otazka>
    <odpoved>23</odpoved>
  </zaznam>
  <zaznam><otazka>2+2?</otazka><odpoved>4</odpoved></zaznam>
</kviz>
```

Všimnite si, že tagy sú v sebe vnorené – keď napríklad vo vnútri tagu `<zaznam>` otvoríme tag `<otazka>`, musíme najskôr použiť zatvárací tag `</otazka>`, až potom môžeme použiť tag `</zaznam>`.

V domácom kole sme si napísali program, ktorý o danom texte vedel povedať, či je to korektný XML dokument alebo nie. V praxi však často potrebujeme viac. Príkladom sú rôzne prehliadače www stránok. Mnohé www stránky sú napísané nekorektne, prehliadač sa však napriek tomu snaží aj takúto stránku vám čo najlepšie zobraziť.

V tejto úlohe sa budeme zaoberať jednoduchým spôsobom opravy chýb, ktoré v XML dokumente objavíme. Ak vstupný dokument nie je korektný XML dokument, napravíme to tak, že niektoré tagy z neho vyškrtáme.

Tagy, ktoré vyškrtať, určíme nasledujúcim spôsobom. Tagy budeme spracúvať v poradí, v akom sa vyskytujú v dokumente. V každom okamihu si budeme pamätať postupnosť práve otvorených tagov.

Otvárací tag spracujeme vždy tak, že ho pridáme na koniec pamätanej postupnosti.

U zatváracieho tagu sú tri možnosti:

1. Pasuje k poslednému práve otvorenému tagu.
   V takomto prípade máme nájdený pár a jednoducho dotyčný otvárací tag vyhodíme z pamätanej postupnosti.
2. Pasuje k niektorému skoršiemu otvorenému tagu.
   V takomto prípade postupne (od konca) vyškrtávame otvorené otváracie tagy, až kým nedostaneme situáciu z bodu 1.
3. Nepasuje k žiadnemu otvorenému tagu.
   V takomto prípade vyškrtneme tento zatvárací tag.

Po spracovaní celého dokumentu ešte vyškrtáme tie otváracie tagy, ktoré ostali otvorené.

Súťažná úloha:

Napíšte program, ktorý čo najefektívnejšie odsimuluje tento algoritmus a vypíše počet vyškrtnutých tagov.

Formát vstupu a výstupu:

Vstup obsahuje niekoľko (nanajvýš 10 000) riadkov textu, každý z nich obsahuje najviac 100 znakov.

Môžete predpokladať, že znaky `<` a `>` sa vyskytujú len ako začiatky a konce tagov, a že názov každého tagu je tvorený 1 až 8 malými písmenami.

Na výstup vypíšte jedno celé číslo – počet vyškrtnutých tagov.

Príklady:

Vstup

```
<srnka><paroh>hnedy</paroh>
</srnka><jelen></jelen>
```

Výstup

```
0
```

Korektný XML dokument.

Vstup

```
<z> <a> <a> <b> </a>
<v> </v> </a>
```

Výstup

```
2
```

Pri spracúvaní prvého `</a>` *vyšktrneme* `<b>`*. Po spracovaní celého dokumentu vyškrtneme* `<z>`*.*

Vstup

```
<ahoj> <ako> </ako> </sa>
<mas>   </mas> </ahoj>
```

Výstup

```
1
```

Tag `</sa>` *vyšktrneme, lebo nemá čo zavrieť.*

B-II-4 Rákosie sa vracia

V tomto ročníku olympiády v teoretickej úlohe pestujeme zvláštne druhy rákosia. V študijnom texte, ktorý nájdete za zadaním úlohy B-I-4, je popísané, ako toto rákosie vyzerá.

Súťažná úloha:

a) *(4 body)* Nájdite nejakú odrodu rákosia, ktorej mŕtve steblá obsahujú len bunky a, b a c, pričom buniek c je toľko isto ako buniek a a b dokopy. Samozrejme, chceme takú odrodu, kde môže narásť každé takéto steblo.

   Vo vašej odrode rákosia teda musí existovať spôsob, ako narastú napr. steblá ε, *bcbcbc*, *cabc* a $a^{47}b^3c^{50}$, a naopak nesmú narásť napr. steblá a, *baca* ani *dedo*.

b) *(6 bodov)* Nájdite nejakú odrodu rákosia, ktorej mŕtve steblá sú tvorené len bunkami a, a navyše počet týchto buniek je ľubovoľné zložené číslo.

   Vo vašej odrode teda musia vedieť narásť napr. steblá a^4, a^6, a^8, a^9, ... Naopak, nesmú narásť napr. steblá ε, a, aa, aaa, a^5, a^{47} ani *dedo*.

   Možno dobrá rada: Každé zložené číslo vieme zapísať v tvare $m \cdot n$, kde $m, n > 1$.

Zadania celoštátneho kola kategórie A

A-III-1 Najväčšia horúčava

„Tento rok bolo strašne horúco! V Hurbanove namerali dokonca rovných štyridsať sedem stupňov! Taká horúčava u nás ešte jakživ nebola!“ nechal sa počuť Tadeáš.

„Ále čoby,“ odvetil Kleofáš, „veď predsa pred trinástimi rokmi bolo vonku ešte oveľa väčšie peklo.“

„Tak potom je toto najväčšie teplo odvtedy.“ zahlásil Tadeáš, no Kleofáš ho opäť sklamal: „Aleba. Nemáš pravdu, aj pred šiestimi rokmi bolo teplejšie.“

Súťažná úloha:
Dané sú údaje tvaru „v roku r bola najvyššia nameraná teplota t_r“.
Ďalej sú dané výroky tvaru „v roku q_1 bolo najväčšie teplo od roku q_2“.
Vašou úlohou bude o každom z výrokov rozhodnúť, či je pravdivý, nepravdivý, alebo otázny.

Výrok je pravdivý, ak platí:

- poznáme teploty pre všetky roky od q_2 po q_1, vrátane
- v roku q_2 bolo aspoň tak teplo ako v roku q_1 (teda $t_{q_2} \geq t_{q_1}$)
- pre každý rok r medzi q_2 a q_1 platí, že v ňom bolo chladnejšie ako v q_1, a teda aj chladnejšie ako v q_2
  (teda $\forall r \in \{q_2 + 1, \ldots, q_1 - 1\};\ t_r < t_{q_1}$)

Výrok je otázny, ak pre niektoré roky medzi q_2 a q_1 nie je známa nameraná teplota, ale vieme tieto teploty doplniť tak, aby sa výrok stal pravdivým.
V ostatných prípadoch je výrok nepravdivý.

Formát vstupu:
Prvá časť vstupu, ktorá popisuje namerané teploty, začína celým číslom N, udávajúcim počet rokov, pre ktoré vieme nameranú teplotu. Nasleduje N dvojíc celých čísel $r_i\ t_{r_i}$. Môžete predpokladať, že údaje sú utriedené podľa čísla roku, teda pre každé i platí $r_i < r_{i+1}$.

Druhá časť vstupu, ktorá popisuje výroky, začína číslom M, udávajúcim počet výrokov. Nasleduje M dvojíc čísel $q_1\ q_2$.

Môžete predpokladať, že $0 \leq N, M \leq 100\,000$ a $0 \leq r_i, t_{r_i}, q_i \leq 1\,000\,000\,000$. Teploty, ktoré nie sú udané na vstupe, môžu nadobúdať aj neceločíselné hodnoty, aj hodnoty mimo uvedeného rozsahu. Riešenie, ktoré funguje za predpokladu $0 \leq N, M \leq 1\,000$, získa aspoň 5 bodov.

Formát výstupu:

Pre každý výrok vypíšte jeden riadok, a v ňom jeho pravdivostnú hodnotu.

Príklad:

Vstup

```
5
2003 49
2004 59
2005 28
2006 38
2008 47
4
2006 2003
2006 2004
2008 2004
2008 2006
```

Výstup

```
nepravdivy
pravdivy
otazny
nepravdivy
```

Tretí výrok môže byť pravdivý napr. ak $t_{2007} = 11$.
Štvrtý výrok nemôže byť pravdivý, lebo $t_{2008} > t_{2006}$.

A-III-2 Tobogany

V Nalomenej Trieske otvorili včera nový vodný zábavný park. Hlavnou atrakciou je obrovská sieť toboganov.

V sieti je N bazénov, ktoré sa nachádzajú v rôznych výškach nad zemským povrchom. Jeden z nich je horný (tam jazda toboganmi začína) a jeden je dolný (tam každá jazda končí).

Bazény sú troch rôznych tvarov: štvorcový, šesťuholníkový a osemuholníkový. Bazény rovnakého tvaru sú navzájom nerozlíšiteľné. Z každého bazénu okrem dolného vedú niekam dodola práve tri tobogany, ktoré sú označené číslami 1, 2 a 3.

Jazda na atrakcii vyzerá tak, že začneme v hornom bazéne a postupne si v každom bazéne, kde sa ocitneme, vyberieme jeden z troch toboganov a pustíme sa ním ďalej dole. Takto pokračujeme, až kým sa nedostaneme do dolného

bazénu. Sieť toboganov je navrhnutá tak, že bez ohľadu na to, ako sa budeme rozhodovať, do dolného bazénu sa časom určite dostaneme.

Do každého bazénu môže ústiť ľubovoľne veľa toboganov. Tie sa však spájajú ešte pred tým, ako vás vypľujú do bazéna, takže tento počet neviete zistiť. Jediné, čo na konci jazdy viete, sú čísla toboganov, ktoré ste si vyberali, a tvary bazénov, cez ktoré ste išli.

Zábavný park má pri otvorení veľkú akciu: Kto prvý nakreslí správne mapu celej atrakcie, dostane ročné vstupné zdarma. Malý Ľuboško si zaumienil, že cenu vyhrá. Celý včerajší deň jazdil na toboganoch, až nakoniec nakreslil mapu, ktorá všetkým jeho jazdám zodpovedala. No keď ju skúsil odovzdať, dozvedel sa, že síce má na nej zakreslené všetky bazény, ale bohužiaľ mnohé z nich viackrát.

Súťažná úloha:

Na vstupe je daná mapa siete toboganov a bazénov, ktorú nakreslil Ľuboško. Viete, že správna mapa je od Ľuboškovej nerozlíšiteľná. To znamená, že ak pôjdete po jednej z máp a budete si voliť čísla toboganov ako len chcete, tak v druhej mape bude vždy taká cesta existovať tiež, a pôjde cez bazény rovnakých tvarov.

Tiež viete, že počet bazénov na správnej mape je najmenší možný. Napíšte program, ktorý zistí, koľko bazénov je na správnej mape.

Formát vstupu:

Prvý riadok vstupu obsahuje jediné celé číslo N ($2 \leq N \leq 5\,000$) udávajúce počet bazénov na Ľuboškovej mape, vrátane horného a dolného. Bazény sú očíslované od 1 po N tak, že bazén s vyšším číslom je bližšie pri zemi. Teda špeciálne bazén 1 je horný a bazén N je dolný bazén.

Nasleduje $N-1$ riadkov, jeden pre každý bazén okrem dolného. Každý riadok obsahuje štyri záznamy: začína číslom t_i ($t_i \in \{4, 6, 8\}$) udávajúcim tvar i-teho bazénu. Nasledujú tri čísla bazénov, do ktorých vedú z tohto bazéna tobogany s číslami 1, 2 a 3.

Formát výstupu:

Výstup má obsahovať jediné číslo – počet bazénov na správnej, najmenšej možnej mape.

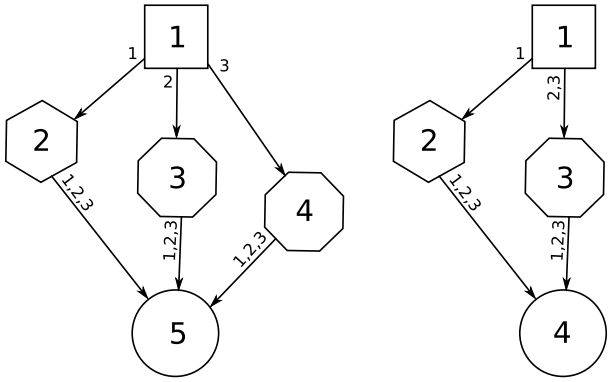
Príklady:

Vstup

```
5
4 2 3 4
6 5 5 5
8 5 5 5
8 5 5 5
```

výstup

```
4
```



Zadaná mapa je znázornená na obrázku vľavo. (Všimnite si, že na tvare dolného bazéna nezáleží, preto nie je ani udaný na vstupe. V obrázku ho značíme kruhom.)
Najmenšia mapa, ktorá je nerozoznateľná od zadanej, je na obrázku vpravo.

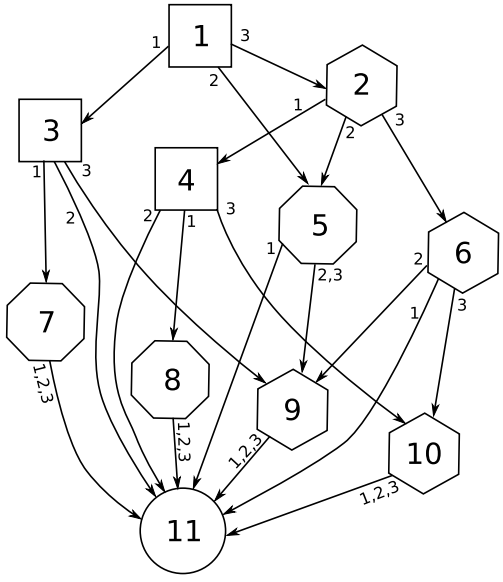
Vstup

```
11
4 3 5 2
6 4 5 6
4 7 11 9
4 8 11 10
8 11 9 9
6 11 9 10
8 11 11 11
8 11 11 11
6 11 11 11
6 11 11 11
```

Výstup

```
8
```

Mapa znázorňujúca vstup:



A-III-3 Prekladacie stroje

Študijný text nájdete za zadaním súťažnej úlohy A-I-4.

Súťažná úloha:

a) *(6 bodov)* Množina N obsahuje desiatkové zápisy kladných celých čísel.

Zostrojte prekladací stroj A **s čo najmenej stavmi**, pre ktorý platí: Nech X je ľubovoľná podmnožina N. Potom $A(X)$ je množina zápisov čísel, ktoré sú v X a zároveň nie sú deliteľné číslom 105.

Teda napríklad pre $X = \{47, 315, 411, 1050\}$ musí byť $A(X) = \{47, 411\}$.

Ak si myslíte, že takýto prekladací stroj neexistuje, dokážte to.

b) *(4 body)* Množina X obsahuje jediný reťazec ε.

Množina Y obsahuje práve všetky reťazce, ktoré sú tvorené písmenami a a b a písmen a je v nich rovnako ako písmen b. Teda napríklad $\varepsilon \in Y$, $abba \in Y$, $aababb \in Y$, ale $a \notin Y$ a $bbab \notin Y$.

Zostrojte prekladací stroj **s čo najmenej stavmi**, ktorý preloží množinu X na Y.

Ak si myslíte, že takýto prekladací stroj neexistuje, dokážte to.

A-III-4 O lenivých prasiatkach

Na lúke si polihovalo P lenivých prasiatok. Až sa priblížil večer a prasiatka sa rozhodli, že keďže sa schyľuje k búrke, pôjdu prespať do senníkov. Na lúke je práve P senníkov. Je jedno, v ktorom senníku bude ktoré prasiatko spať, ale do každého sa zmestí len jedno.

Prasiatka sú ale lenivé, a tak by sa chceli schovať s čo najmenšou celkovou námahou.

Lúku si môžeme predstaviť ako veľký štvorec, zložený z $N \times N$ jednotkových štvorcov, ktoré budeme volať políčka. Na niektorých políčkach rastú stromy, tadiaľ prasiatka ísť nemôžu. Ostatné políčka sú voľné. Na niektorých P voľných políčkach stoja senníky. Na niektorých P voľných políčkach stoja prasiatka. (Je možné, že niektoré prasiatka už sú na políčkach, kde sú senníky.) V každom okamihu môže byť na každom políčku najviac jedno prasiatko.

Námahu presunu budeme merať na kroky. V jednom kroku sa môže jedno prasiatko presunúť na jedno zo štyroch políčok, ktoré stranou susedia s políčkom, kde práve stojí. (Samozrejme, na políčku nesmie byť strom ani iné prasiatko, a prasiatko nesmie opustiť lúku.)

Navyše vedia naše prasiatka skákať cez seba. Ak by prasiatko A chcelo ísť smerom, kde stojí prasiatko B, môže v jednom kroku prasiatko B preskočiť a dopadnúť až na políčko za ním. (Takto sa vlastne prasiatko A v jednom kroku posunulo až o dve políčka.)

Súťažná úloha:

Na vstupe je mapa lúky, polohy prasiatok a senníkov.

Zistite, s akou najmenšou námahou sa vedia všetky prasiatka schovať do senníkov.

Formát vstupu:

V prvom riadku vstupu je celé číslo N ($1 \leq N \leq 20$) – rozmer lúky.

Nasleduje N riadkov, ktoré obsahujú mapu lúky. Každý z nich obsahuje N znakov, a každý znak je buď „`.`“ (bodka) alebo veľké písmeno „`X`“. Bodky predstavujú voľné políčka a písmená sú stromy.

V nasledujúcom riadku je celé číslo P ($1 \leq P \leq 4$) – počet prasiatok.

Nasleduje P riadkov, ktoré popisujú polohu prasiatok. Každý z nich obsahuje dve celé čísla od 1 do N – riadok a stĺpec políčka, kde prasiatko stojí. (Riadky aj stĺpce sú číslované v poradí v akom sú na vstupe.) Všetkých P riadkov je navzájom rôznych, a všetky popísané políčka sú voľné.

Posledných P riadkov rovnako popisuje polohu senníkov.

Formát výstupu:

Vypíšte jediný riadok a v ňom jedno celé číslo – najmenšiu námahu, ktorá stačí na to, aby sa všetky prasiatka schovali do senníkov.

Môžete predpokladať, že pre každý použitý testovací vstup bude existovať riešenie.

Horšie riešenia:

V 12 z 15 sád testovacích vstupov bude $N \leq 10$. V 10 z týchto 12 sád bude dokonca $N \leq 8$.

V aspoň 5 testovacích sadách bude $P \leq 2$. V aspoň 2 testovacích sadách bude existovať optimálne riešenie, pri ktorom sa ani raz neskočí.

Príklady:

Vstup

```
3
X.X
...
X.X
2
2 1
2 2
3 2
1 2
```

Výstup

```
3
```

Najskôr prejde druhé prasiatko z $[2, 2]$ *do jedného senníka, napríklad na* $[3, 2]$*, následne môže prvé prasiatko prejsť na dva kroky do druhého senníka.*

Vstup

```
6
......
.X.X..
.X.XX.
.X..X.
.XXXX.
......
1
6 5
4 4
```

Výstup

```
13
```

Toto prasiatko je na lúke samo. Nemá ako skákať, preto sa proste vyberie najkratšou cestou do senníka. V tomto prípade je výhodnejšie stenu obísť sprava.

Vstup

```
5
X.XXX
X.XXX
.....
X.XXX
X.XXX
2
1 2
3 1
3 5
5 2
```

Výstup

```
7
```

Najlepšie riešenie je, aby jedno prasiatko prišlo na križovatku na súradniciach $[3, 2]$ *a nechalo druhé cez seba preskočiť.*

Vstup

```
7
.......
.......
.......
.......
.......
.XX.XX.
.......
2
1 1
1 7
7 3
7 5
```

Výstup

```
15
```

Prasiatka sa v hornom riadku vyberú k sebe, stredným stĺpcom sa presunú dole (pričom striedavo cez seba skáču), a tam sa opäť rozídu každé na inú stranu.

Vstup

```
7
.......
.......
XXXXX..
....X..
....X..
....X..
....X..
4
1 1
1 2
2 1
2 2
6 6
6 7
7 6
7 7
```

Výstup

```
20
```

Prasiatka začínajú v ľavom hornom rohu mapy, rozostavené do štvorca. Pri optimálnom presune v každom kroku niektoré prasiatko skáče – kým to ide, skáču vodorovne, keď už to nejde, tak zvisle.
Po prvých 10 krokoch budú prasiatka na štyroch políčkach v pravom hornom rohu, po ďalších 10 sa dostanú všetky do senníkov.

A-III-5 Posledná dobrota

Na stole je obrovská kopa dobrôt: koláčov aj ovocia. Julka a Monika sa s nimi hrajú nasledovnú hru.

Hra prebieha v ťahoch. Prvá ťahá Julka, potom sa striedajú. Hráčka na ťahu si buď môže vziať nejaké koláče, alebo niekoľko kusov ovocia, alebo oboje. Ale ak chce brať oboje, musí si koláčov zobrať toľko isto ako kusov ovocia.

Aby sa neprejedli, dohodli sa navyše na tom, že v jednom ťahu si môže každá z nich vziať najviac K koláčov a najviac K kusov ovocia.

Hru vyhrá tá, ktorá zoberie zo stola úplne poslednú dobrotu.

Súťažná úloha:

Vašou úlohou bude napísať knižnicu, ktorá bude túto hru hrať namiesto Julky, proti nášmu programu, ktorý bude hrať namiesto Moniky. Body dostanete za každú hru, ktorú vaša knižnica celú korektne odohrá a vyhrá.

Dostanete od nás k dispozícii kostru knižnice, ktorú máte napísať, a ukážkový program protihráča. Tento protihráč však na rozdiel od toho, ktorého použijeme pri testovaní, nebude hrať optimálne.

Popis knižnice:

Vaša knižnica má obsahovať nasledujúce dve procedúry / funkcie:

```
procedure zaciatok(A, B, K: longint);
void zaciatok(int A, int B, int K);
```

V tejto funkcii si vaša knižnica môže inicializovať premenné a štruktúry, ktoré potom bude využívať pri hre. Náš program túto funkciu zavolá práve raz, na začiatku celej hry. Pomocou parametrov oznámi náš program vašej knižnici počet A koláčov na stole, počet B kusov ovocia na stole a limit na ťah K.

Vo všetkých testovacích vstupoch bude $1 \leq A, B \leq 10\,000\,000$ a $1 \leq K \leq 1\,000\,000$.

```
procedure tahaj(A, B: longint; var berA, berB: longint);
void tahaj(int A, int B, int *berA, int *berB);
```

Túto funkciu náš program zavolá vždy, keď je vaša knižnica na ťahu. (Prvýkrát teda hneď po zavolaní funkcie `zaciatok`.) V premenných A a B vám oznámi aktuálne počty koláčov a ovocia. Pred skončením musí vaša funkcia nastaviť hodnoty premenných `berA` a `berB` na počty koláčov a ovocia, ktoré chcete v tomto ťahu zo stola zobrať.

Príklad hry:

- `zaciatok(7,2,3)`

  Na stole je 7 koláčov, 2 kusy ovocia, v jednom ťahu sa dajú vziať max. 3 rovnaké veci.

- `tahaj(7,2,berA,berB)`

  Prvý ťah. Julka sa rozhodne zobrať 1 koláč a 1 kus ovocia, preto nastaví `berA` aj `berB` na 1.

  Monika berie 3 koláče, ostávajú 3 koláče a 1 ovocie.

- `tahaj(3,1,berA,berB)`

  Julka sa dozvie aktuálny stav hry, zoberie 1 koláč.

  Monika berie po jednom koláči aj ovocí.

- `tahaj(1,0,berA,berB)`

  Julka berie posledný koláč a vyhráva.

Riešenia domáceho kola kategórie A

A-I-1 O zdanlivom kopci

Zadanou úlohou je vybrať z danej postupnosti čo najdlhšiu podpostupnosť, ktorá by najskôr rástla a potom klesala.

Keby sme vedeli, ktorý prvok je v našej podpostupnosti ten najväčší („vrchol kopca"), mali by sme ľahšiu úlohu: z časti postupnosti od začiatku po vrchol vybrať čo najdlhšiu rastúcu podpostupnosť končiacu „vrcholom", a zo zvyšku zase vybrat čo najdlhšiu klesajúcu podpostupnosť. No a vyberať klesajúcu podpostupnosť je to isté ako vyberať rastúcu idúc sprava doľava.

Preto nám stačí riešiť nasledujúcu jednoduchšiu úlohu: K danej postupnosti pre každé i spočítame dĺžku d_i najdlhšej rastúcej podpostupnosti, ktorá končí členom a_i.

Ukážeme si najskôr pomalšie riešenie tejto úlohy, potom jeho efektívnejšiu verziu.

Pomalšie riešenie:

Zjavne $d_1 = 1$. Ak už vieme hodnoty d_1 až d_k pre nejaké k, hodnotu d_{k+1} spočítame nasledovne:

Predstavme si, že už máme nájdenú najdlhšiu podpostupnosť končiacu prvkom a_{k+1}. Pozrime sa na ňu a zakryme si posledný člen. To, čo teraz vidíme, musí byť opäť nejaká rastúca podpostupnosť. A nie len tak hocijaká. Nech jej posledný člen je a_x. Potom to, čo vidíme, musí byť (jedna možná) najdlhšia rastúca podpostupnosť končiaca a_x. Jej dĺžka je teda d_x, a dĺžka našej pôvodnej podpostupnosti je $d_x + 1$.

My síce nepoznáme x, ale tu je ľahká pomoc: Vyskúšame všetky možné x a vyberieme si tú najlepšiu možnosť. A ktoré sú to tie „všetky možné" x? V prvom rade musí byť $1 \leq x \leq k$, no a navyše musí platiť $a_x < a_{k+1}$, aby bola nová podpostupnosť naďalej rastúca. Dostávame teda:

$$d_{k+1} = \max_{1 \leq x \leq k,\ a_x < a_{k+1}} d_x + 1$$

Algoritmus, ktorý spočíta hodnoty d_i použitím tohoto vzťahu, má časovú zložitosť $O(N^2)$, kde N je počet členov spracúvanej postupnosti. Takéto riešenie mohlo získať nanajvýš 7 bodov.

Lepšie riešenie:

Na zrýchlenie vyššie popísaného algoritmu použijeme nasledujúce pozorovanie: Ak vieme vybrať dve rastúce podpostupnosti rovnakej dĺžky, „lepšia“ je tá, ktorá končí menšou hodnotou. Totiž ak vieme po pridaní nasledujúcich členov postupnosti nejako predĺžiť tú „horšiu“, vieme rovnako predĺžiť aj tú „lepšiu“.

V každom okamihu si teda stačí pre každú možnú dĺžku pamätať jednu rastúcu podpostupnosť – tú „najlepšiu“, čiže končiacu najmenšou možnou hodnotou.

Presnejšie, nech m_i je najmenšia hodnota, akou môže končiť i-prvková rastúca podpostupnosť vybratá z doteraz spracovaných prvkov.

Na začiatku je $m_0 = 0$ a $\forall i > 0;\ m_i = \infty$.

Teraz budeme postupne po jednom spracúvať prvky danej postupnosti. Ako sa hodnoty m_i zmenia po spracovaní jedného jej členu?

Všimnime si najskôr, že v každom okamihu platí, že hodnoty m_i (ktoré sú rôzne od ∞) sú rastúce. Totiž ak vieme spraviť rastúcu podpostupnosť dĺžky i, ktorá končí hodnotou m_i, tak jej prvých $i-1$ členov tvorí rastúcu podpostupnosť dĺžky $i-1$, ktorá končí členom menším ako m_i, preto nutne $m_{i-1} < m_i$.

Nech je práve spracúvaný člen postupnosti x. Potom zjavne existuje práve jedno a také, že $m_a < x \leq m_{a+1}$.

Čo toto znamená? V prvom rade vieme, že doteraz „najlepšia“ vybraná podpostupnosť dĺžky $a+1$ (a viac) končila číslom väčším alebo rovným x. Žiadnu takúto postupnosť nevieme predĺžiť hodnotou x, a teda hodnoty od m_{a+2} ďalej sa meniť nebudú.

Podobne sa nebudú meniť ani hodnoty od m_0 po m_a, vrátane. Tieto sú všetky už teraz menšie ako x, takže ich zlepšiť nevieme.

Jediné, čo sa teda zmení, je, že teraz vieme vybrať rastúcu postupnosť dĺžky $a+1$, ktorá končí hodnotou x. Odteraz teda bude $m_{a+1} = x$. A zároveň vieme, že $a+1$ je dĺžka najdlhšej vybranej rastúcej podpostupnosti končiacej práve spracúvaným prvkom.

No a už máme náš algoritmus skoro hotový. Teraz si stačí len uvedomiť, že hodnoty m_i sú utriedené, a teda vieme nájsť hodnotu a binárnym vyhľadávaním v čase $O(\log N)$. Zvyšné úpravy už spravíme v konštantnom čase.

Potrebujeme spracovať N prvkov, časová zložitosť teda bude $O(N \log N)$.

Listing programu:

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

```
#define NEKONECNO 987654321

vector<int> rastuce(const vector<int> &A) {
  int N = A.size();
  vector<int> result(N);
  vector<int> M(N+1,NEKONECNO); M[0] = 0;
  for (int i=0; i<N; i++) {
    // binarnym vyhladavanim najdeme "a"
    int a = lower_bound( M.begin(), M.end(), A[i] ) - M.begin() - 1;
    M[a+1] = A[i];
    result[i] = a+1;
  }
  return result;
}

int main() {
  // nacitame vstup
  int N; cin >> N; vector<int> A(N); for (int i=0; i<N; i++) cin >> A[i];

  // spocitame dlzky pre rastuce podpostupnosti
  vector<int> B = rastuce(A);

  // spocitame dlzky pre klesajuce podpostupnosti
  reverse(A.begin(),A.end());
  vector<int> C = rastuce(A);
  reverse(C.begin(),C.end());

  // spocitame vysledok
  int res = 0;
  for (int i=0; i<N; i++) res = max(res, B[i]+C[i]-1 );
  cout << res << endl;
  return 0;
}
```

A-I-2 Rezervácie miesteniek

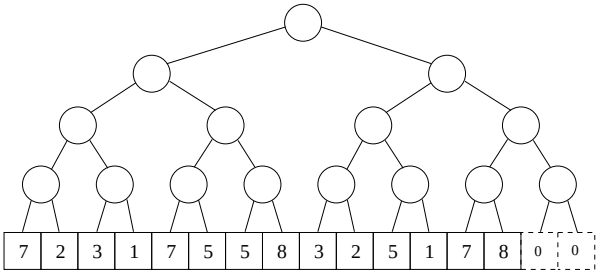
Naša železničná trať má N úsekov medzi stanicami. Keď nám príde nejaká požiadavka, potrebujeme sa pozrieť na úseky, ktoré obsahuje, a nájsť úsek, kde je voľného miesta najmenej. Ak je ho tam dosť, je ho dosť všade a požiadavku prijmeme. Naopak, ak ho tam dosť nie je, požiadavku musíme odmietnuť.

Na 4 body si stačilo pre každý úsek pamätať v poli počet obsadených miest. Na 7 bodov potrebujeme vedieť efektívnejšie odhaliť odmietané požiadavky (teda nájsť najplnší úsek), no a na 10 bodov budeme musieť efektívne spracúvať aj prijaté požiadavky.

Intervalový strom:

Pre jednoduchosť budeme predpokladať, že N je mocnina dvoch. (Ak by nebolo, zväčšíme ho na najbližšiu mocninu dvoch. Uvedomte si, že tým sa zväčší menej ako na dvojnásobok pôvodnej hodnoty.)

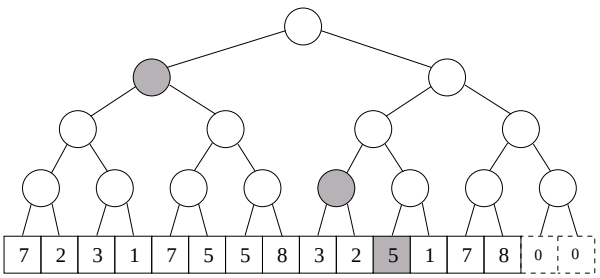
Použijeme dátovú štruktúru známu pod menom *intervalový strom*. Ten bude vyzerať takto:



Listy intervalového stromu zodpovedajú jednotlivým úsekom trate. Všimnite si, že vnútorný vrchol, ktorý je k úrovní nad listami, zodpovedá intervalu obsahujúcemu 2^k po sebe idúcich úsekov. Tie intervaly, ktoré zodpovedajú vrcholom nášho stromu, budeme volať *jednoduché*.

Načo je intervalový strom dobrý? Ukážeme, že ľubovoľný interval úsekov vieme šikovne „poskladať“ z jedoduchých intervalov.

Zaoberajme sa najskôr intervalom, ktorý obsahuje úseky od 1 po k. Tvrdíme, že tento vieme zložiť z najviac $\lg N$ jednoduchých intervalov. Toto dokážeme tak, že budeme z jeho ľavej strany odkrajovať čo najväčšie jednoduché intervaly, až kým sa neminie. Najjednoduchšie je to vidieť na príklade. Napr. interval „od 1 po 11“ vieme rozdeliť na „od 1 po 8“, „od 9 po 10“ a „od 11 po 11“.



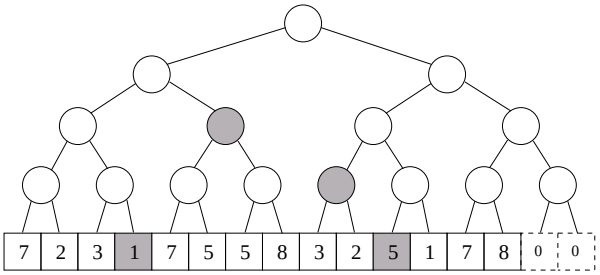
Vyznačené vrcholy zodpovedajú jednoduchým intervalom,
ktoré dokopy tvoria interval „od 1 po 11“.

Odhad počtu použitých intervalov vyplýva napríklad z toho, že v každom kroku odkrojíme viac ako polovicu intervalu.

Všimnite si, že postup krájania zodpovedá schádzaniu z koreňa dole po intervalovom strome. V každom vrchole sa pozrieme, či nerozkrájaná časť zadaného

intervalu leží celá v ľavom podstrome. Ak áno, nič nekrájame a zlezieme doň. Ak nie, odkrojíme interval zodpovedajúci ľavému podstromu a zlezieme do pravého.

Vo všeobecnej situácii, keď chceme naskladať interval úsekov od k po l, budeme na tom podobne, vystačíme si s $2 \lg N$ jednoduchými intervalmi. Dôkaz je podobný, pôjdeme dole intervalovým stromom. Akonáhle niekedy zistíme, že zadaný interval zasahuje do oboch podstromov, rozkrojíme ho na dve časti. No a každá z častí už teraz zodpovedá jednoduchšiemu prípadu, ktorý sme rozobrali vyššie.

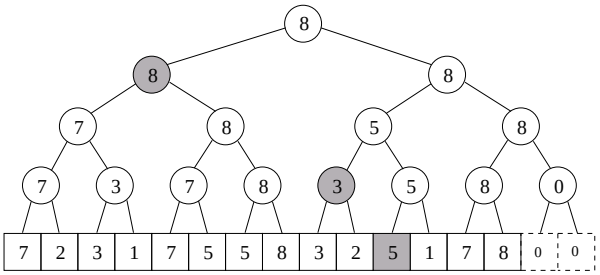


Všeobecný prípad: interval „od 4 po 11“.

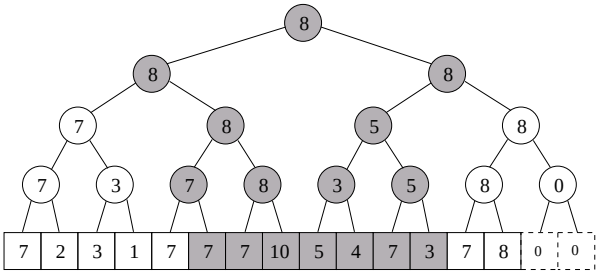
Ukázali sme teda, že v čase $O(\log N)$ vieme ľubovoľný interval rozdeliť na niekoľko častí, ktoré zodpovedajú vrcholom stromu.

Keby sme pre každý vrchol stromu vedeli, aké je maximum z hodnôt listov v jeho podstrome, vedeli by sme teda v čase $O(\log N)$ povedať maximum pre ľubovoľný interval úsekov. A v tom je celý trik intervalového stromu: Zvolili sme si niekoľko vhodných intervalov, z ktorých vieme efektívne naskladať odpoveď pre hocijaký iný interval.

Riešenie za 7 bodov je v tomto okamihu už triviálne. Spravíme si intervalový strom, v ktorom budú v listoch počty obsadených miest na jednotlivých úsekoch, a v každom vrchole si pamätáme maximum z hodnôt listov v jeho podstrome. Keď príde požiadavka „x y z“, v čase $O(\log N)$ zistíme maximum z hodnôt v intervale od $y + 1$ po z. Ak je väčšie ako $M - x$, požiadavku odmietneme. V opačnom prípade potrebujeme strom upraviť. Zväčšíme hodnoty v listoch od $y + 1$ po z a opravíme všetky vrcholy nad nimi. Toto vieme spraviť v čase $O(z - y)$, čo vieme zhora odhadnúť ako $O(N)$.



Intervalový strom s maximami pre jednoduché intervaly.
Maximum v intervale „od 1 po 11“ je rovné maximu zvýraznených políčok.

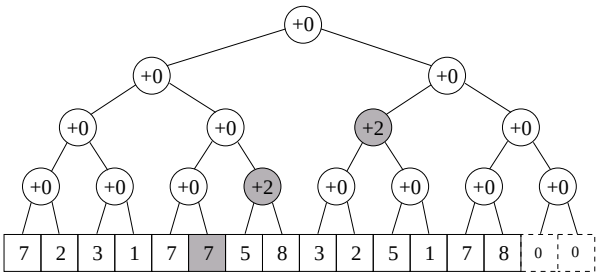


Situácia počas spracúvania prijatej požiadavky „2 5 12“:
Vo vyznačenom intervale (sivé štvorčeky) pribudli dvaja cestujúci.
Sivé krúžky označujú vrcholy, ktoré ešte treba (zdola nahor) prepočítať.

Celkovo má toto riešenie časovú zložitosť $O(P \log N + AN)$, kde P je počet všetkých a A je počet prijatých požiadaviek.

Efektívne spracovanie prijatej požiadavky:

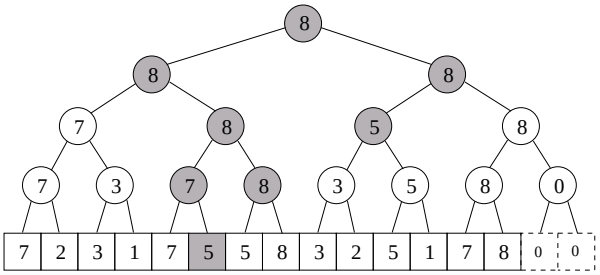
Zostáva ukázať, ako šikovnejšie upravovať informáciu o počtoch cestujúcich v prípade prijatia požiadavky. Trik je jednoduchý. Nebudeme ukladať informácie o počte cestujúcich len v listoch, ale v celom strome. Keď teda máme zväčšiť hodnoty v nejakom intervale, zväčšíme hodnoty v zodpovedajúcich jednoduchých intervaloch. V každom vrchole intervalového stromu si teda namiesto doterajšej jednej hodnoty (maxima) budeme pamätať hodnoty dve (maximum a zmenu počtu cestujúcich v ňom).



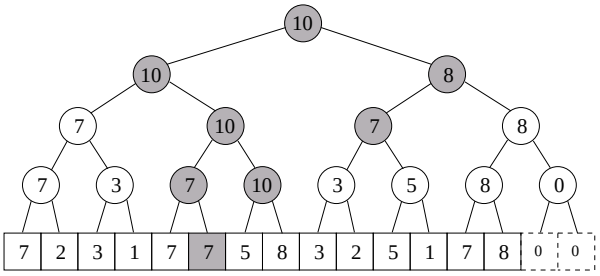
Pridanie 2 ľudí do úsekov 6 až 12.
Čísla predstavujú **počty cestujúcich**, sivé vrcholy sa menili.

Všimnite si, že pre každý list platí, že počet cestujúcich na zodpovedajúcom úseku dostaneme tak, že sčítame všetky počty cestujúcich na ceste z daného listu do koreňa stromu.

Upravili sme teda strom tak, že pamätané počty cestujúcich už sú správne. Zostáva upraviť pamätané maximá pre podstromy. Kde sa tie budú meniť? Nuž, vo vrcholoch, kde sa zmenili počty cestujúcich, a všade nad nimi:



Pridanie 2 ľudí do úsekov 6 až 12.
Čísla predstavujú **maximá**, sivé vrcholy treba prepočítať.



Pridanie 2 ľudí do úsekov 6 až 12.
Takto budú vyzerať maximá po prepočítaní.

Vrcholov, kde treba prepočítať maximum, je $O(\log N)$. Sú to totiž práve tie vrcholy, ktoré navštívime, keď delíme náš interval na jednoduché časti – a teda aj keď upravujeme počty cestujúcich. Vieme teda obe veci upraviť naraz v jednej jednoduchej rekurzívnej funkcii.

Celkovo vieme každú požiadavku spracovať v čase $O(\log N)$, preto celková časová zložitosť nášho riešenia je $O(P \log N)$.

Ešte jedna poznámka k implementácii: Intervalový strom si vieme pamätať v jednom statickom poli, podobne ako napríklad haldu. Koreň bude na políčku s indexom 1, synovia vrcholu x budú na políčkach $2x$ a $2x + 1$. Listy budú na políčkach N až $2N - 1$.

Listing programu:

```
#include <iostream>
using namespace std;

#define MAXN 1000000

class vrchol { public: int cestujuci, maximum; };
vrchol strom[2*MAXN + 47];
int N,M,P;

int najdiMaximum(int x, int y, int kde, int left, int right, int uz) {
  // "kde" je cislo vrcholu v ktorom prave sme
  // "left" a "right" su konce jemu zodpovedajuceho jednoducheho intervalu
  // "uz" je pocet cestujucich, o ktorych uz vieme, ze su v intervale
  //   (dozvedeli sme sa o nich vyssie)

  if (x<=left && y>=right) return uz + strom[kde].maximum;
  if (y<left || x>right) return 0;
  int dlzka = (right-left+1)/2;
  uz += strom[kde].cestujuci;
  return max(
    najdiMaximum(x,y, 2*kde,   left,       left+dlzka-1, uz),
    najdiMaximum(x,y, 2*kde+1, left+dlzka, right,        uz) );
}

void pridaj(int x, int y, int kolko, int kde, int left, int right) {
  if (x<=left && y>=right) {
    strom[kde].maximum += kolko;
    strom[kde].cestujuci += kolko;
    return;
  }
  if (y<left || x>right) return;
  int dlzka = (right-left+1)/2;
  pridaj(x,y,kolko, 2*kde,   left,       left+dlzka-1);
  pridaj(x,y,kolko, 2*kde+1, left+dlzka, right       );
  strom[kde].maximum =
    strom[kde].cestujuci + max( strom[2*kde].maximum, strom[2*kde+1].maximum );
}

int main() {
  int _N;
  cin >> _N >> M >> P;
  N=1; while (N<_N) N *= 2;
  while (P--) {
    int k,x,y;
    cin >> k >> x >> y; x++;
    int m = najdiMaximum(x,y,1,1,N,0);
    if (k > M-m) {
      cout << "odmietnuta" << endl;
    } else {
```

```
      cout << "prijata" << endl;
      pridaj(x,y,k,1,1,N);
    }
  }
}
```

A-I-3 Fibonacciho sústava

Jednoznačnosť pekného zápisu:

Nájsť pekný zápis prirodzeného čísla N vo Fibonacciho sústave vlastne znamená nájsť množinu Fibonacciho čísel, ktorých súčet je N, všetky sú navzájom rôzne a žiadne dve nenasledujú vo Fibonacciho postupnosti po sebe.

Na úvod si všimnime, že ak samotné N je Fibonacciho číslo, tak samo osebe tvorí hľadanú množinu. Tomu zodpovedá pekný zápis s práve jednou jednotkou: pre F_n (kde $n \geq 2$) je to „$1\underbrace{0\ldots0}_{n-2}$“.

Fibonacciho zápisy prirodzených čísel do 15 si môžeme ručne vypísať, aby sme odpozorovali nejakú zákonitosť. Dostaneme:

číslo	zápis
$F(2) = 1$	1
$F(3) = 2$	10
$F(4) = 3$	100
4	101
$F(5) = 5$	1000
6	1001
7	1010
$F(6) = 8$	10000
9	10001
10	10010
11	10100
12	10101
$F(7) = 13$	100000
14	100001
15	100010

Vidíme dve veci. V prvom rade sme si overili, že pre malé prirodzené čísla dokazované tvrdenie platí, v druhom rade začíname vidieť systém, akým pekný zápis funguje. Skúsime teraz dokázať, že to tak bude fungovať aj ďalej.

Matematickou indukciou dokážeme, že pre každé $n \geq 2$ platí tvrdenie $T(n)$: „Všetky prirodzené čísla z množiny $\{F_n, \ldots, F_{n+1} - 1\}$ majú práve jeden pekný zápis, a ten obsahuje práve $n - 1$ cifier. Navyše platí, že každé väčšie číslo už musí mať aspoň n-ciferný pekný zápis.“

Z tabuľky vidíme, že pre n do 6 toto tvrdenie platí. Zostáva teda dokázať indukčný krok. Nech pre nejaké n platia tvrdenia $T(2)$ až $T(n-1)$, dokážeme, že z toho vyplýva platnosť $T(n)$.

Zaujímajú nás teda zápisy čísel z množiny $M(n) = \{F_n, \ldots, F_{n+1} - 1\}$. Z tvrdenia $T(n-1)$ vieme, že ich zápis musí mať aspoň $n - 1$ cifier. No a viac cifier mať nemôže, najmenšie číslo s n-ciferným zápisom je predsa zjavne F_{n+1}. Pekný zápis každého z týchto čísel musí teda mať práve $n - 1$ cifier.

To znamená, že keď sa na pekný zápis dívame ako na množinu Fibonacciho čísel, táto množina musí obsahovať F_n. A keďže nemôžeme použiť dve po sebe idúce Fibonacciho čísla, táto množina nesmie obsahovať F_{n-1}. Inými slovami, pekný zápis čísla z $M(n)$ musí byť tvaru „$10\underbrace{?\ldots?}_{n-3}$“.

Zoberme si teraz nejaké číslo X z množiny $M(n)$. Ukážeme, že chýbajúce cifry pekného zápisu X sú určené jednoznačne. Prečo? Chýbajúce cifry zjavne musia tvoriť pekný zápis čísla $Y = X - F_n$. Platí $X < F_{n+1}$, preto $Y < F_{n+1} - F_n = F_{n-1}$. Z indukčného predpokladu teda vieme, že Y má práve jeden pekný zápis, a že ten má dostatočne málo cifier na to, aby sa zmestil na chýbajúce miesta. Preto má aj X práve jeden pekný zápis.

Posledný krok dôkazu: Najväčšie číslo s pekným zápisom dĺžky $n - 1$ má zjavne pekný zápis tvaru „$101010\ldots$“. Toto môžeme zapísať ako „$10Z$“, kde Z je maximálny pekný zápis dĺžky $n-3$. O tom už vieme z indukčného predpokladu, že zodpovedá číslu $F_{n-1} - 1$. Náš maximálny zápis teda zodpovedá číslu $F_n + F_{n-1} - 1 = F_{n+1} - 1$, q.e.d.

Nájdenie pekného zápisu:

Náš dôkaz nám priamo dáva metódu na nájdenie pekného zápisu čísla X: Nájdeme najväčšie n také, že $F_n \leq X < F_{n+1}$. Toto Fibonacciho číslo sa v zápise X bude vyskytovať. Zvyšok zápisu X je tvorený zápisom menšieho čísla $X - F_n$. Opakovaním postupu zostrojíme postupne celý zápis.

Listing programu:

```
var F : array[0..45] of longint;
    i,X,n : longint;
begin
```

```
   { spocitame Fibonacciho cisla }
   F[0] := 0;  F[1] := 1;
   for i:=2 to 100 do F[i] := F[i-1] + F[i-2];
   { nacitame vstup }
   read(X);
   { najdeme najvyssiu cifru }
   n:=2;
   while (F[n] <= X) do inc(n);
   dec(n);
   { postupne vypisujeme cifry }
   while (n >= 2) do begin
      if (X >= F[n]) then begin write(1); dec(X,F[n]); end else write(0);
      dec(n);
   end;
   writeln;
end.
```

Počítanie zaujímavých čísel:

Označme $R(k, A, B)$ riešenie zadanej úlohy, teda počet čísel z množiny $\{A, A+1, \ldots, B\}$, ktoré majú vo svojom peknom zápise práve k jednotiek.

Začneme tým, že si zadanú úlohu zjednodušíme. Zadanú úlohu stačí vedieť riešiť pre prípad $A = 1$, lebo zjavne platí $R(k, A, B) = R(k, 1, B) - R(k, 1, A-1)$. Slovne: Aby sme spočítali vhodné čísla v zadanej množine, spočítame vhodné čísla neprevyšujúce B, a od nich odpočítame vhodné čísla menšie ako A.

Poďme teda ukázať, ako spočítať hodnotu $R(k, 1, N)$ pre dané N. Začneme tým, že si prevedieme N do Fibonacciho sústavy a zistíme jeho počet cifier c. Teraz vieme, že všetky hľadané pekné zápisy budú mať najviac c cifier. Môžeme si pre jednoduchosť predstaviť, že tie z nich, ktoré sú kratšie, doplníme zľava nulami na dĺžku presne c.

Na zadaný problém sa teda môžeme pozerať nasledovne: Máme postupnosti núl a jednotiek, ktoré majú dĺžku c. Potrebujeme spočítať, koľko z nich má všetky nasledujúce vlastnosti:

- Obsahuje práve k jednotiek.
- Je pekným zápisom, teda nemá dve jednotky po sebe.
- Vo Fibonacciho sústave predstavuje číslo neprevyšujúce N.

Len prvá podmienka:

Spočítať postupnosti, ktoré spĺňajú prvú podmienku, je ľahké: Máme postupnosť dĺžky c, potrebujeme vybrať k miest, kde budú jednotky, toto sa dá spraviť $\binom{c}{k}$ spôsobmi.

Prvé dve podmienky:

Tu to nebude omnoho zložitejšie. Zoberme ľubovoľnú postupnosť, ktorá spĺňa prvé dve podmienky. Tesne za každou z prvých $k-1$ jednotiek je v nej určite nula. Keď týchto $k-1$ núl vyhodíme, dostaneme novú postupnosť dĺžky $c-k+1$, v ktorej je k jednotiek. A naopak, z novej postupnosti vieme tú pôvodnú jednoznačne zrekonštruovať, stačí za každú jednotku okrem poslednej vložiť jednu nulu.

Tým sme dokázali, že postupností dĺžky c, ktoré spĺňajú prvé dve podmienky, je rovnako ako postupností dĺžky $c-k+1$, ktoré spĺňajú prvú podmienku – čiže $\binom{c-k+1}{k}$.

Jednoduché rekurzívne riešenie:

Budeme rekurzívne zľava doprava generovať všetky možné postupnosti núl a jednotiek, ktoré spĺňajú prvé dve podmienky. Zároveň si budeme v každom okamihu pamätať, akému číslu zodpovedá práve vygenerovaná postupnosť, aby sme neprekročili N.

Toto riešenie je ľahko naprogramovateľné, ale má veľkú časovú zložitosť – počet krokov je aspoň tak veľký ako počet nájdených čísel. Presný odhad časovej zložitosti by bol náročný, uvedieme len náznak myšlienky: Každé číslo do N má v peknom zápise nanajvýš $\log_2 N$ jednotiek. Preto pre nejaké k ($1 \leq k \leq \log_2 N$) bude mať odpoveď veľkosť aspoň $N/\log_2 N$, a teda náš algoritmus má časovú zložitosť $\Omega(N/\log N)$.

Listing programu:

```
#include <iostream>
using namespace std;

long long F[100]; // fibonacciho cisla

int generuj(int c, int k, int poz, int lim) {
  // c je dlzka postupnosti, k je pocet zostavajucich jednotiek
  // poz je pozicia ktoru doplname, lim je horny limit na velkost cisla
  if (lim<0) return 0;
  if (poz>=c) return k==0;
  // !!! sem pride optimalizacia !!!
  int result = generuj(c,k,poz+1,lim); // umiestnime 0
  if (k>0) result += generuj(c,k-1,poz+2,lim - F[c-poz+1]); // umiestnime 1
  return result;
}

int rataj(int k, int N) {
  int c=1; while (F[c+2]<=N) c++; // zistime pocet cifier c
  return generuj(c,k,0,N);
```

```
}

int main() {
  F[0]=0; F[1]=1; for (int i=2; i<100; i++) F[i]=F[i-1]+F[i-2];
  int k,A,B;
  cin >> k >> A >> B;
  cout << (rataj(k,B) - rataj(k,A-1)) << endl;
  return 0;
}
```

Optimalizácia rekurzívneho riešenia:

Do programu pridáme dva riadky, ktoré ho priam zázračne urýchlia, napriek tomu, že oba budú veľmi jednoduché.

Prvé pozorovanie: Ak nám zostáva do konca použiť k jednotiek a máme už iba menej ako $2k-1$ pozícií, riešenie neexistuje a môžeme sa vrátiť o pozíciu späť.

Druhé pozorovanie: Ak máme doplniť posledných x miest, najväčšie číslo, ktoré vieme vyrobiť, je $F_{x+2}-1$. Pokiaľ vieme, že ešte ani týmto číslom neprekročíme hornú hranicu, nemusíme všetky možné čísla generovať, ale vieme ich rovno zarátať. Ako sme ukázali vyššie, je ich $\binom{x-k+1}{k}$.

V našej rekurzívnej funkcii teda pridáme nasledovné podmienky:

```
  if (c-poz < 2*k-1) return 0;
  if (lim >= F[c-poz+2]-1) return C[c-poz-k+1][k];
```

Kombinačné čísla si môžeme napríklad na začiatku jednoducho predrátať využitím známeho vzťahu $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$:

```
  long long C[100][100]; // kombinacne cisla
  for (int i=0; i<100; i++) for (int j=0; j<=i; j++)
    C[i][j] = (j==0||j==i) ? 1 : C[i-1][j-1]+C[i-1][j];
```

Prečo je takto vylepšené riešenie odrazu také efektívne? Preto, že situácia, kedy by sme nevedeli použiť žiadnu z podmienok, skoro nikdy nenastane.

Všimnime si jednu dôležitú vlastnosť pekného zápisu: Keď máme dva pekné zápisy rovnakej dĺžky, ten, ktorý predstavuje menšie číslo, je aj lexikograficky menší. (Hravo sa to dá dokázať indukciou.)

Všimnime si teraz ľubovoľnú situáciu počas behu nášho vylepšeného algoritmu. V tomto okamihu máme vygenerovanú nejakú postupnosť núl a jednotiek dĺžky najviac c, a ideme spočítať, koľkými spôsobmi sa dá doplniť. Porovnajme doteraz vygenerovanú postupnosť s rovnako dlhým prefixom pekného zápisu N.

Ak je naša postupnosť väčšia, znamená to, že už sme N prekročili, a naša funkcia teda okamžite vráti nulu. Ak je naša postupnosť menšia, vieme, že nech už doplníme čokoľvek, N neprekročíme, a preto môžeme rovno vrátiť počet všetkých doplnení. Jediný prípad, kedy musíme spraviť dve rekurzívne volania (a teda generovať postupnosť ďalej) je ten, keď nastala rovnosť – čiže keď je naša postupnosť prefixom pekného zápisu N.

Ukážeme si to na príklade: Nech má N pekný zápis 100101001000 a nech $k = 5$. Keď sme v situácii 101..., aj keby sme už doplnili samé nuly, bude výsledok väčší ako N, preto takéto riešenie neexistuje. Keď sme v situácii 100100..., môžeme na zvyšných 6 miest doplniť ľubovoľný pekný zápis s troma jednotkami. V tomto prípade máme teda 4 riešenia.

Rekurzívne volania bude teda náš algoritmus robiť len raz pre každú dĺžku prefixu, teda rádovo c-krát. Preto je časová zložitosť samotnej rekurzie $O(c)$, alebo ekvivalentne $O(\log N)$. Celý algoritmus je o niečo pomalší kvôli predrátaniu kombinačných čísel. Dalo by sa s tým ešte niečo robiť, ale neoplatí sa. Ani pre N rovné miliarde by sme už žiadne viditeľné zrýchlenie nespozorovali.

Matematické riešenie:

Na všetko je vzorec, a ani tento prípad nie je výnimkou. Naše riešenie bude vyzerať nasledovne: Prevedieme N na pekný zápis. Nájdeme najbližšie menšie alebo rovné číslo N', ktoré má v peknom zápise práve k jednotiek. (Rozmyslite si ako na to, nie je to také triviálne ako sa zdá na prvý pohľad.) Priamo z toho, ako tento zápis vyzerá, spočítame, koľký v poradí zaujímavý zápis to je.

Využijeme dve skutočnosti. Prvou bude pozorovanie z predchádzajúcej časti: Keď máme dva pekné zápisy rovnakej dĺžky, ten, ktorý predstavuje menšie číslo, je aj lexikograficky menší.

Druhou bude trik, ktorý sme použili, keď sme rátali pekné postupnosti: Keď zoberieme pekné postupnosti dĺžky c s k jednotkami a v každej za každou jednotkou okrem poslednej škrtneme nulu, dostaneme práve všetky postupnosti dĺžky $c - k + 1$ s k jednotkami.

Všimnime si teraz, že keď sme zobrali dve postupnosti a z oboch takto vyškrtali nuly, tak tá, ktorá bola menšia pred škrtaním, musela zostať menšia aj po ňom.

Takže pokojne môžeme zobrať zápis N', týmto spôsobom ho upraviť, a následne zodpovedať jednoduchšiu otázku: Koľká v poradí postupností s k jednotkami je toto?

No a toto vieme ľahko spočítať. V každom okamihu stačí rozlíšiť medzi dvoma prípadmi. Ak začína nulou, stačí túto nulu zahodiť. Ak začína jednotkou,

sú pred ňou všetky postupnosti, ktoré začínajú nulou. No a tých je $\binom{d-1}{k}$, kde d je aktuálna dĺžka. Všetky tieto zarátame, jednotku zo začiatku zahodíme a zmenšíme k o jedna.

Skúste si uvedomiť, že toto riešenie, ktoré sme dostali, je takmer identické s riešením, ku ktorému sme sa z úplne opačného prístupu dopracovali v predchádzajúcej časti.

Listing programu:

```
#include <iostream>
using namespace std;

long long F[100]; // fibonacciho cisla
long long C[100][100]; // kombinacne cisla
int jednotky[100];
int N;

long long rataj(int k, int N) {
  if (N==0) return 0;
  // spocitame zapis N
  int co=2; while (F[co+1]<=N) co++;
  int J=0; while (co>=2) { if (N>=F[co]) { jednotky[J++]=co-2; N-=F[co]; } co--;
}

  // zostrojime zapis N'
  if (J<k) {
    if (jednotky[0]<=2*k-2) return 0;
    int skus;
    for (skus=J-1; skus>=0; skus--) {
      int ostava=(jednotky[skus]-1)/2, treba=k-(skus+1);
      if (ostava >= treba) break;
    }
    jednotky[skus]--;
    for (int i=skus+1; i<k; i++) jednotky[i] = jednotky[i-1]-2;
  }
  J=k;

  // odstranime zo zapisu N' nuly a zostrojime si ho
  for (int i=0; i<J; i++) jednotky[i] -= J-1-i;
  int zapis[100];
  memset(zapis,0,sizeof(zapis));
  for (int i=0; i<J; i++) zapis[jednotky[i]]=1;

  // spocitame vysledok
  long long res = 1;
  for (int i=jednotky[0]; i>=0; i--) if (zapis[i]) { res += C[i][k]; k--; }
  return res;
}
```

```
int main() {
  F[0]=0; F[1]=1; for (int i=2; i<100; i++) F[i]=F[i-1]+F[i-2];
  for (int i=0; i<100; i++) for (int j=0; j<=i; j++)
    C[i][j] = (j==0||j==i) ? 1 : C[i-1][j-1]+C[i-1][j];
  long long k,A,B;
  cin >> k >> A >> B;
  cout << (rataj(k,B) - rataj(k,A-1)) << endl;
  return 0;
}
```

A-I-4 Prekladacie stroje

Podúlohy a+b:

V prvej časti tohto vzorového riešenia ukážeme, že stroje B a C nerobia navzájom úplne presne inverzné operácie. Problém bude v tom, že dekódovanie morzeovky bez oddeľovačov nemusí byť jednoznačné – niektoré reťazce čiariek a bodiek sa dajú preložiť viacerými spôsobmi, a iné naopak vôbec.

Zoberme si napríklad jednoslovnú množinu $M_1 = \{i\}$. Keď túto preložíme strojom B do morzeovky, dostaneme $B(M_1) = \{\bullet\bullet\}$. Reťazec $\bullet\bullet$ je ale aj morzeovkovým zápisom reťazca ee. Preto keď $B(M_1)$ preložíme späť strojom C, dostávame $C(B(M_1)) = \{i, ee\} \neq M_1$, a teda prvé tvrdenie neplatí.

Podobne ľahko zistíme, že pre $M_2 = \{-\}$ je $B(C(M_2)) = \emptyset \neq M_2$, a teda ani druhé tvrdenie neplatí.

Podúloha c:

Prekladať budeme len niektoré reťazce, a to reťazce, v ktorých sú najskôr všetky a, a potom všetky b. Každú dvojicu a prepíšeme na jedno a, a každé b na tri b.

Formálne, náš prekladací stroj A bude pätica $(K, \Sigma, P, 0, F)$, kde $K = \{0, 1\}$, $F = \{1\}$ a prekladové pravidlá vyzerajú nasledovne:

$$P \;=\; \Big\{\ (0, aa, a, 0),\ (0, \varepsilon, \varepsilon, 1),\ (1, b, bbb, 1)\ \Big\}$$

Podúloha d:

Najlepšie riešenie potrebuje tri operácie. Jedno takéto riešenie si ukážeme.

Čo všetko vieme dosiahnuť jedným prekladom? V prvom rade vieme z množiny M_1 vybrať len pre nás zaujímavé reťazce, teda tie, kde idú najskôr a a potom b. Keď sa dočítame na koniec slova, vieme ešte napísať aj nejaké c, už však nijak nevieme zabezpečiť, aby ich počet bol rovný počtu a a b.

Formálne, definujme prekladací stroj $A_1(K_1, \Sigma, P_1, A, F_1)$, kde $K_1 = \{A, B, C\}$, $F_1 = \{C\}$ a prekladové pravidlá vyzerajú nasledovne:

$$P_1 \quad = \quad \Big\{ \ (A, a, a, A),\ (A, \varepsilon, \varepsilon, B),\ (B, b, b, B),\ (B, \varepsilon, \varepsilon, C),\ (C, \varepsilon, c, C) \ \Big\}$$

Zjavne až na počet písmen c je $M_2 = A_1(M_1)$ presne to, čo hľadáme. Ako ale zabezpečiť, aby sa počty a, b aj c museli rovnať?

Trik je v tom, že rovnako, ako sme si práve vyrobili množinu reťazcov s rovnakým počtom a a b, vieme vyrobiť aj množinu reťazcov, ktoré budú mať rovnako b a c.

Formálne, definujme prekladací stroj $A_2(K_2, \Sigma, P_2, A, F_2)$, kde $K_2 = \{A, B, C\}$, $F_2 = \{C\}$ a prekladové pravidlá vyzerajú nasledovne:

$$P_2 \quad = \quad \Big\{ \ (A, \varepsilon, a, A),\ (A, \varepsilon, \varepsilon, B),\ (B, a, b, B),\ (B, \varepsilon, \varepsilon, C),\ (C, b, c, C) \ \Big\}$$

Aj tento prekladací stroj vyrobí z M_1 takmer presne to, čo treba: $M_3 = A_2(M_1)$ má nasledovné vlastnosti: Písmená v reťazcoch idú v správnom poradí a písmen b a c je rovnako veľa.

No a posledný krok je jednoduchý, $G = M_2 \cap M_3$. Slovne: V prieniku množín sú práve tie reťazce, ktoré majú obe dobré vlastnosti – počet a je rovný počtu b (lebo je to reťazec z M_2) a počet b je rovný počtu c (lebo je to zároveň reťazec z M_3).

Riešenia domáceho kola kategórie B

B-I-1 O spájaní polí

Pomalšie riešenie:

Prvé, čo nás môže napadnúť, je začať tým, že presunieme prvky z B do A, a následne sa pokúsime pole A utriediť.

Ak by sme na to použili nejaký štandardný algoritmus na triedenie (napr. QuickSort), najlepšia časová zložitosť, akú môže náš program mať, by bola $O\Big((M+N)\cdot\log(M+N)\Big)$. A navyše by sme si museli pri implementácii triedenia poradiť bez pomocnej pamäte. V našom prípade však existujú aj efektívnejšie riešenia.

Vzorové riešenie:

S úlohou „spojte dve utriedené postupnosti do jednej“ sa stretáme napríklad pri známom triediacom algoritme MergeSort.

Keby sme mohli použiť nové pole, bolo by to jednoduché. Predstavme si triedené prvky ako ľudí usporiadaných podľa výšky. Máme teda dva rady, z každého najmenší stojí úplne vpredu. Kto je teda najmenší celkovo? Predsa menší z tých dvoch predných. Toho teda pošleme, nech sa ide postaviť do výsledného poradia na prvé miesto. A pokračujeme: kto je najmenší z tých, čo zostali? No predsa jeden z tých dvoch, ktorí teraz stoja ako prví v radoch. Tak ich opäť porovnáme, menšieho pošleme „na výstup“, a tak dokola, až kým sa nám niektorý rad neminie. Potom už len pridáme zvyšok druhého radu na koniec výslednej postupnosti.

Takéto riešenie by malo časovú zložitosť $O(M+N)$, teda lineárnu od počtu spracúvaných prvkov. (To preto, že každý krok vieme spraviť v konštantnom čase, a v každom kroku pošleme jeden prvok na výstup.)

Ak by sme ale chceli tento postup použiť na našu úlohu, narazíme na problém – nemáme nové pole, kam by sme výstup dávali.

Máme ale nejaké voľné miesto, a to na konci poľa A. Pomôžeme si teda tak, že výslednú postupnosť budeme zostrojovať od konca. Namiesto toho, aby sme porovnávali dva najmenšie (ešte nespracované) prvky, budeme vždy porovnávať dva najväčšie, a „výhercov“ budeme ukladať do poľa A od konca.

Uvedomte si, že voľné miesto v poli A sa nám pri takomto postupe predčasne neminie. Dokonca to vieme povedať aj presnejšie: V každom okamihu bude v

poli A toľko voľného miesta, koľko ešte ostáva v poli B nespracovaných prvkov.

Poznámka:

Úloha by bola riešiteľná v lineárnom čase dokonca aj vtedy, ak by sme nemali k dispozícii ani len to voľné miesto v poli A. Teda zadanie by bolo: „v poli A sú za sebou dve utriedené postupnosti, prvá má M prvkov a druhá N, bez použitia pomocných polí ich spojte do jednej“.

Tento problém je známy pod názvom Merge-in-place a je známych viacero rôznych algoritmov, ktoré ho riešia v lineárnom čase. Kvôli prísnejšiemu obmedzeniu na použitú pamäť však vo všetkých prípadoch však ide o algoritmy neporovnateľne komplikovanejšie ako naše riešenie.

Listing programu:

```
procedure merge(var A,B : array of longint; M,N : longint);
var kdeA,kdeB,kdeC : longint;
begin
   kdeA := M-1; kdeB := N-1; kdeC := M+N-1;
   { kym mame co porovnavat, porovnavame }
   while (kdeA >= 0) and (kdeB >= 0) do begin
      if (A[kdeA] > B[kdeB]) then begin
         A[kdeC] := A[kdeA]; dec(kdeA); dec(kdeC);
      end else begin
         A[kdeC] := B[kdeB]; dec(kdeB); dec(kdeC);
      end;
   end;
   { ked uz nemame co porovnavat, dopresuvame zvysok B }
   while (kdeB >= 0) do begin
      A[kdeC] := B[kdeB]; dec(kdeB); dec(kdeC);
   end;
end;

{ pre nazornost uvedieme aj program, ktory nasu proceduru pouziva }
var
   _A,_B:array[0..200000] of longint;
   i,_M,_N:longint;
begin
   read(_M); for i:=0 to _M-1 do read(_A[i]);
   read(_N); for i:=0 to _N-1 do read(_B[i]);
   merge(_A,_B,_M,_N);
   for i:=0 to _M+_N-1 do writeln(_A[i]);
end.
```

B-I-2 Krtkovou norou

Ak nevieme riešiť zadanú úlohu, skúsme najskôr vyriešiť jednoduchšiu. Zaoberajme sa teda najskôr prípadom, kde sú všetky chodbičky rovnako dlhé.

Riešenie pre chodbičky rovnakej dĺžky:

V takomto prípade môžeme na nájdenie najvzdialenejších brlôžkov použiť postup známy pod menom *prehľadávanie do šírky*.

Myšlienka tohoto postupu je jednoduchá. Najskôr postupne spracujeme všetky brlôžky, ktoré sú od začiatočného vo vzdialenosti 1, potom všetky, ktoré sú vo vzdialenosti 2, a tak ďalej.

Presnejšie to bude fungovať takto: Budeme mať jedno pole, v ktorom budeme o každom brlôžku mať zaznamenané, či už vieme, ako je ďaleko. Okrem toho si budeme pamätať zoznam brlôžkov, ktoré čakajú na spracovanie.

Na začiatku si zaznamenáme, že do brlôžku krtka Vítka je vzdialenosť 0 a zaradíme tento brlôžok do zoznamu na spracovanie.

Teraz dokola opakujeme nasledujúci postup: Vyberieme zo zoznamu brlôžok, ktorý je v ňom najdlhšie, a ideme ho spracovať. Nech je tento brlôžok vo vzdialenosti d od brlôžku, kde sme začínali. Jeho spracovanie bude vyzerať nasledovne: Pozrieme sa postupne na všetky brlôžky, ktoré s ním susedia. Pre niektoré z nich už vzdialenosť vieme, tie necháme na pokoji. Ostatným brlôžkom (t.j. tým, do ktorých sme sa pri prehľadávaní doteraz nedostali) nastavíme vzdialenosť na $d + 1$ a zaradíme ich do zoznamu na spracovanie.

Zoznam brlôžkov na spracovanie si môžeme pamätať napríklad ako spájaný zoznam, ale najjednoduchšie je uložiť si ich v poli ako súvislý úsek. Dátová štruktúra, ktorá funguje ako náš zoznam brlôžkov (t.j. vždy vyberáme prvok, ktorý je v nej najdlhšie), sa volá *fronta*. Jej implementáciu pomocou poľa nájdete v listingu vzorového programu.

Aké efektívne je toto riešenie? Každý z N brlôžkov práve raz zaradíme do zoznamu, raz ho odtiaľ vyberieme a raz spracujeme. A každú z M chodbičiek spracujeme dvakrát (po jednom raze v každom koncovom brlôžku). Preto je časová zložitosť tohoto riešenia $O(N + M)$.

Listing programu:

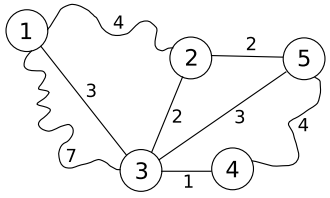
```
var G : array[1..1000,1..1000] of longint;
    stupne,fronta,vzdialenost : array[1..1000] of longint;
    bol : array[1..1000] of boolean;
    N,M,dalsi,i,a,b,h,kde,kam,zac,kon,max : longint;

begin
```

```
   readln(N,M);
   for i:=1 to M do begin
      readln(a,b);
      inc(stupne[a]); inc(stupne[b]);
      G[a][ stupne[a] ] := b;
      G[b][ stupne[b] ] := a;
   end;
   { pustime prehladavanie do sirky }
   { v kazdom okamihu plati, ze brlozky na spracovanie su v poli fronta[]
     na poziciach zac az (kon-1) }
   zac:=1; kon:=2; fronta[1]:=1; bol[1]:=true; vzdialenost[1]:=0;
   while (zac < kon) do begin
      kde := fronta[zac]; inc(zac);
      for i:=1 to stupne[kde] do begin
         kam := G[kde][i];
         if (not bol[kam]) then begin
            bol[kam]:=true; vzdialenost[kam]:=vzdialenost[kde]+1;
            fronta[kon]:=kam; inc(kon);
         end;
      end;
   end;
   { najdeme a vypiseme najvzdialenejsie brlozky }
   max:=0; for i:=1 to N do if (vzdialenost[i]>max) then max:=vzdialenost[i];
   for i:=1 to N do if (vzdialenost[i]=max) then writeln(i);
end.
```

Riešenie pôvodnej úlohy:

V pôvodnej úlohe môžu mať chodbičky rôzne dĺžky, a to od 1 do 7. Tým sa ale nedáme zastrašiť. Presvedčíme krtkov, aby na každej chodbičke, ktorá je dlhšia ako 1, vyhrabali nové brlôžky, ktoré ju rozdelia na úseky dĺžky 1.

Napríklad pre sieť z príkladu v zadaní by výsledok snaženia krtkov vyzeral takto:



Takto sme dostali novú sieť chodbičiek, kde už majú všetky chodbičky rovnakú dĺžku. A pre túto sieť už zadanú úlohu vieme riešiť. (Len si treba dať pozor na to, že vypisovať sa majú len pôvodné komôrky, nie tie nové.)

Posledná otázka znie: nepokazili sme týmto trikom časovú zložitosť riešenia? Nuž, našťastie nie. Prečo? Nová sieť má nanajvýš 7× toľko chodbičiek ako pôvodná. A na každej chodbičke pribudlo najviac 6 nových brlôžkov.

Brlôžkov je teda dokopy nanajvýš $N+6M$ a chodbičiek $7M$. Náš algoritmus teda bude mať časovú zložitosť $O(N+13M) = O(N+M)$.

Listing programu:

```
var G : array[1..1000,1..1000] of longint;
    stupne,fronta,vzdialenost : array[1..1000] of longint;
    bol : array[1..1000] of boolean;
    N,M,dalsi,i,a,b,h,kde,kam,zac,kon,max : longint;

begin
   readln(N,M);
   dalsi := N+1; { cislo, ktore dostane dalsi brlozok }
   for i:=1 to M do begin
      readln(a,b,h);
      kde := a;
      while (h>1) do begin { kopeme novy brlozok }
         inc(stupne[kde]); inc(stupne[dalsi]);
         G[kde][ stupne[kde] ] := dalsi;
         G[dalsi][ stupne[dalsi] ] := kde;
         kde := dalsi; inc(dalsi);
         dec(h);
      end;
      inc(stupne[kde]); inc(stupne[b]);
      G[kde][ stupne[kde] ] := b;
      G[b][ stupne[b] ] := kde;
   end;
   { pustime prehladavanie do sirky }
   zac:=1; kon:=2; fronta[1]:=1; bol[1]:=true; vzdialenost[1]:=0;
   while (zac < kon) do begin
      kde := fronta[zac]; inc(zac);
      for i:=1 to stupne[kde] do begin
         kam := G[kde][i];
         if (not bol[kam]) then begin
            bol[kam]:=true; vzdialenost[kam]:=vzdialenost[kde]+1;
            fronta[kon]:=kam; inc(kon);
         end;
      end;
   end;
   { najdeme a vypiseme najvzdialenejsie brlozky }
   max:=0; for i:=1 to N do if (vzdialenost[i]>max) then max:=vzdialenost[i];
   for i:=1 to N do if (vzdialenost[i]=max) then writeln(i);
end.
```

B-I-3 Kontrola XML

V prvom rade nás bude zaujímať len postupnosť tagov na vstupe, text medzi nimi môžeme pokojne ignorovať.

Na otváracie a zatváracie tagy sa môžeme dívať ako na ľavé a pravé zátvorky rôznych typov. Potom podmienka, že tagy sa nesmú prekrývať, nadobudne jasnejší význam: XML dokumenty sa podobajú dobre uzátvorkovaným výrazom.

Ako teda budeme kontrolovať, či je náš XML dokument korektný? Tagy budeme spracúvať v poradí, v akom sa v ňom vyskytujú. V každom okamihu si budeme pamätať, ktoré tagy sme už otvorili a ešte nezavreli. Ak je nasledujúci spracúvaný tag otvárací, len ho pridáme na koniec pamätaného zoznamu. Naopak, ak je zatvárací, môže nastať hneď niekoľko problémov.

V prvom rade ten najbežnejší problém: posledný otvorený tag bol iný ako ten, ktorý práve spracúvame (`<a></b>`).

Mohlo sa tiež stať, že nie je otvorený vôbec žiaden tag (`<a></a></x>`).

Jediná dobrá situácia je, ak posledný otvorený a ešte nezavretý tag sedí so zatváracím tagom, ktorý práve spracúvame. Vtedy tento tag odstránime zo zoznamu, ktorý si pamätáme (`<a><b><c></c></b>`).

Ak sa takto dočítame až na koniec XML dokumentu, ešte potrebujeme skontrolovať jednu vec: či nezostali niektoré tagy otvorené (`<a><b></b>`).

Zoznam otvorených tagov:

Ako si má náš program pamätať zoznam otvorených tagov tak, aby sme nové tagy vedeli spracúvať čo najefektívnejšie?
Všimnime si, že potrebujeme robiť nasledujúce operácie:

– pridaj nový tag na koniec zoznamu
– zisti, aký tag je na konci zoznamu
– odstráň posledný tag zo zoznamu

Na toto je ako stvorená dátová štruktúra *zásobník*. V našom programe na uloženie zásobníka používame statické pole a jednu premennú, v ktorej si pamätáme aktuálny počet uložených tagov.

Pár slov k implementácii:

Náš program spracúva vstup po znakoch. Vždy, keď narazí na znak `<`, zavolá procedúru, ktorá sa pokúsi sa načítať názov tagu. Ak sa jej to nepodarí, program okamžite ukončíme. Ak áno, tag spracujeme podľa vyššie uvedeného postupu.

Všimnite si, že vďaka použitiu tejto procedúry sa hlavný program zjednodušil v podstate len na samotné spracúvanie tagov.

Listing programu:

```
function je_znak(ch : char) : boolean;
begin je_znak := (ch>='a') and (ch<='z'); end;

procedure urcite_nacitaj(var ch : char);
begin
   if eof then begin writeln('nie'); halt; end;
   read(ch);
end;

procedure nacitaj_tag(var zaciatok : boolean; var nazov : string);
var ch : char;
begin
   nazov:=''; zaciatok:=true;
   urcite_nacitaj(ch);
   if (ch='/') then begin zaciatok:=false; urcite_nacitaj(ch); end;
   while true do begin
      if (ch='>') then break;
      if (not je_znak(ch)) then begin writeln('nie'); halt; end;
      nazov := nazov + ch;
      if (length(nazov) > 8) then begin writeln('nie'); halt; end;
      urcite_nacitaj(ch);
   end;
   if (nazov='') then begin writeln('nie'); halt; end;
end;

var zasobnik : array[1..10000] of string;
    pocet : longint;
    ch : char;
    zaciatok : boolean;
    nazov : string;

begin
   pocet := 0;
   while not eof do begin
      read(ch);
      if (ch='<') then begin
         nacitaj_tag( zaciatok, nazov );
         if (zaciatok) then begin
            { vlozime novy tag na zasobnik }
            inc(pocet);
            zasobnik[pocet] := nazov;
         end else begin
            { porovname tag s vrchom zasobnika }
            if (pocet=0) then begin writeln('nie'); halt; end;
            if (zasobnik[pocet]<>nazov) then begin writeln('nie'); halt; end;
            dec(pocet);
         end;
      end;
   end;
```

```
   if (pocet>0) then begin writeln('nie'); halt; end;
   writeln('ano');
end.
```

B-I-4 Rákosie

Úlohou je nájsť odrodu rákosia, ktorej steblá majú 2^n (mŕtvych) buniek. Asi prvá myšlienka, ktorá nám môže napadnúť, je použiť pravidlo $A \to AA$, aby sa nám bunky množili. Rákosie by mohlo rásť vo fázach, pričom v každej ďalšej by sa zdvojnásobil počet áčok.

Jediným problémom je teda dohliadnuť na to, aby sa v každej fáze počet áčok naozaj zdvojnásobil. Všimnite si, že samotné pravidlo $A \to AA$ je problematické, pretože nemôžeme ovplyvniť, na ktoré áčka sa použije. Môže sa použiť na každé áčko raz (tak by sme to chceli), ale môže sa napríklad použiť aj opakovane vždy na prvé áčko (tak o chvíľu stratíme pojem o ich počte).

Aby sme do rastu zaviedli trochu disciplíny, budeme mať jednu špeciálnu bunku, ktorá sa bude po steble „pohybovať“. Návod, ako na to, nám dáva už druhý príklad zo študijného materiálu. Vezmime si také pravidlo $AB \to BA$. Presvedčte sa, že keby sme mali steblo $ABBBBB$, použitím tohto pravidla by A postupne preliezlo cez všetky béčka.

Naša špeciálna bunka sa bude volať M, pretože pri pohybe „doprava“ (od koreňa) bude navyše *množiť* áčka. Dosiahneme to pravidlom $Ma \to aaM$. Napríklad, ak sú na začiatku áčka štyri, po prechode bude áčok osem:

$$Maaaa \Rightarrow aaMaaa \Rightarrow aaaaMaa \Rightarrow aaaaaaMa \Rightarrow aaaaaaaaM$$

Čo sa stane, keď M príde na koniec? Máme dve možnosti: buď nám už vyrástlo dosť dlhé steblo a chceli by sme skončiť, alebo by sme sa mali vrátiť a znovu a znovu dĺžku zdvojnásobovať.

Ako ale vôbec zistíme, či už sme na konci? Odpoveď je jednoduchá: na začiatku použijeme pravidlo $Z \to BaME$, ktorým vytvoríme prvé áčko, špeciálnu „množiacu“ bunku a navyše si aj „označíme“ začiatok a koniec stebla (B ako begin a E ako end). Teraz už budeme vedieť, či sme na konci stebla.

Ako sa vrátime? Nemôžeme pridať pravidlo $aM \to Ma$, pretože potom by M mohlo chodiť striedavo aj doprava aj doľava (a nevedeli by sme ho „donútiť“ počet buniek práve zdvojnásobiť; M-ko by si „mohlo robiť, čo chce“; ak ideme doprava, musíme ísť až na koniec; ak sa vraciame, musíme sa vrátiť až na začiatok). Preto M-ko, ktoré je na konci (označenom E-čkom) zmeníme na R,

ktoré bude chodiť *iba* doľava. Inými slovami, písmenom R budeme označovať našu špeciálnu bunku M, keď sa vracia. Keď R príde na začiatok (označený B-čkom), zmení sa späť na M. Máme teda pravidlá

$$Z \to BaME \quad Ma \to aaM \quad ME \to RE \quad aR \to Ra \quad BR \to BM$$

Čo urobíme, keď budeme chcieť skončiť? Po predchádzajúcej úvahe by to už nemal byť problém: Zmeníme M napríklad na H (ako halt). Bunka H sa postará o to, aby v steble ostali iba áčka, t.j. „zmaže“ B, E, a nakoniec aj samu seba. Dosiahneme to pomocou pravidiel

$$ME \to H \quad aH \to Ha \quad BH \to \varepsilon.$$

Úplne všetky genetické pravidlá našej odrody teda vyzerajú takto:

Z	$\to$	$BaME$	(označíme začiatok a koniec)
Ma	$\to$	aaM	(množíme áčka)
ME	$\to$	$RE \mid H$	(na konci sa otočíme, alebo končíme)
aR	$\to$	Ra	(vraciame sa)
BR	$\to$	BM	(otočíme sa, ideme množiť)
aH	$\to$	Ha	(ideme zmazať B)
BH	$\to$	ε	(ostanú len áčka)

Ukážme si príklad, ako steblo narastie na 4 áčka:

$$\begin{aligned}
Z &\Rightarrow BaME \Rightarrow BaRE \Rightarrow BRaE \Rightarrow BMaE \\
&\Rightarrow BaaME \Rightarrow BaaRE \Rightarrow BaRaE \Rightarrow BRaaE \\
&\Rightarrow BMaaE \Rightarrow BaaMaE \Rightarrow BaaaaME \Rightarrow BaaaaH \\
&\Rightarrow BaaaHa \Rightarrow BaaHaa \Rightarrow BaHaaa \Rightarrow BHaaaa \Rightarrow aaaa
\end{aligned}$$

Na záver len malú poznámku: na trojicu M, R, H sa dá pozerať ako na *jednu* špeciálnu bunku, ktorá behá po steble. Pri tom si „pamätá“, čo práve robí (či množí bunky, vracia sa, alebo ukončuje steblo). V našich genetických pravidlách sme určili, ako túto činnosť vykonáva, a ako a kedy svoju činnosť mení.

Riešenia krajského kola kategórie A

A-II-1 Parkovanie kočov

Sluhovia majú koče zaparkovať tak, aby pri odchode vždy najskôr odišiel jeden celý rad, až potom začal odchádzať ďalší. To znamená, že v každom rade musia byť zaparkované koče, ktoré idú podľa dôležitosti bezprostredne po sebe. Inými slovami, ak sú v rade za sebou zaparkované koče dôležitosti d_1 a d_2, nemôže existovať koč s dôležitosťou d_3 taký, že $d_1 < d_3 < d_2$.

Všimnite si tiež, že akonáhle zistíme, že koč, ktorý práve parkujeme, môžeme postaviť do nejakého radu, nič nepokazíme, ak ho tam naozaj postavíme.

Na základe týchto pozorovaní sa dá napísať program s časovou zložitosťou $O(N^3)$. Budeme si pamätať všetky dosiaľ vytvorené rady. Keď príde ďalší hosť, nájdeme pre neho vhodný rad, alebo vytvoríme nový. Koč s dôležitosťou d_i môžeme zaparkovať na koniec už existujúceho radu (nech sa tento rad končí kočom s dôležitosťou d), ak $d < d_i$ a žiadny ďalší hosť už nie je medzi nimi, teda neplatí $d < d_j < d_i$ pre žiadne j. Podobne vieme zistiť, či koč môžeme pridať na začiatok nejakého radu. Keďže kočov je N, radov môže byť v najhoršom prípade, ako sme videli v poslednom príklade v zadaní, až $\Theta(N)$ a keďže pre každý koč a rad nám test trvá $O(N)$, máme naozaj kubický algoritmus.

Testovanie, či koč môžeme zaparkovať do daného radu, vieme zrýchliť: Stačí si vstup predspracovať. Koče prečíslujeme (stále podľa ich dôležitosti) na čísla $1, 2, \ldots, N$. Potom budú rad tvoriť po sebe idúce čísla. Rad kočov s číslami $k, k+1, \ldots, l$ si stačí pamätať ako dvojicu $[k, l]$. Koč číslo i môžeme pridať na koniec radu, ktorý končil kočom číslo $l = i - 1$, alebo na začiatok radu, ktorý začínal kočom číslo $k = i + 1$. Pre každého hosťa prezrieme maximálne $O(N)$ radov, takže máme kvadratický algoritmus.

Ako koče prečíslujeme? Pre každý koč potrebujeme zistiť jeho poradie podľa dôležitosti. Preto koče jednoducho utriedime. Presnejšie, budeme triediť dvojice (d_i, i) podľa dôležitosti d_i. Po zotriedení prvé zložky prepíšeme číslami $1, \ldots, N$. Druhé zložky nám pomôžu vrátiť koče do pôvodného poradia (stačí dvojice usporiadať podľa druhej zložky). Triediť môžeme napríklad quick-sortom alebo heap-sortom.

Vzorové riešenie je (až na prečíslovanie) lineárne a veľmi jednoduché. Keďže koče majú teraz čísla $1, \ldots, N$, stačí si spraviť jedno veľké pole od 0 po $N + 1$, kde budeme zaškrtávať čísla kočov, ktoré sme už zaparkovali. Keď príde koč s číslom i, pozrieme sa, či už sme zaparkovali koč číslo $i-1$ (či je $i-1$ odškrtnuté;

ak áno, môžeme ho dať na koniec existujúceho radu), alebo či je odškrtnuté $i+1$ (ak áno, môžeme koč dať na začiatok existujúceho radu). V opačnom prípade musíme vytvoriť nový rad (zvýšime počet radov o 1). Koč číslo i odškrtneme. Takýto algoritmus má časovú zložitosť $O(N)$ plus zložitosť triedenia a pamäťovú zložitosť $O(N)$.

Iná možnosť ako prečíslovať koče je použiť asociatívne pole (`map` v STL). Stačí koče utriediť, do asociatívneho poľa si ku každému zapamätať poradie po utriedení, a potom pri parkovaní kočov si „prekladať“ ich čísla keď potrebujeme.

Iné riešenie:

Pre zaujímavosť uvedieme ešte jedno trochu iné riešenie. Začneme tým, že si zadané pole skopírujeme a kópiu utriedime, aby sme vedeli, aké koče sa na vstupe nachádzajú, a aby sme vedeli pre každé k, ktorý koč je k-ty najmenší v poradí.

V niektorom rade musí skončiť najmenší koč. Určite nič nepokazíme, ak do tohto radu umiestnime čo najviac kočov – ale koľko to bude?

Vieme napísať funkciu, ktorá pre dané k overí, či vieme k najmenších kočov umiestniť do jedného radu – simulujeme umiestňovanie, pričom koče väčšie ako k-ty najmenší ignorujeme.

Takto vieme spraviť riešenie: postupným skúšaním zistíme, koľko najviac kočov vieme dať do prvého radu. Potom všetky tieto koče vyhodíme z poradia a pre zvyšné koče (ak ešte nejaké zostali) začneme úplne odznova ten istý proces.

Nie je to na prvý pohľad evidentné, ale časová zložitosť takéhoto riešenia je $O(N^2)$. (Myšlienka dôkazu: za skoro každé zavolanie overovacej funkcie nám pribudne jeden koč, ktorý už vieme zaparkovať.)

Listing programu:

```
#include <cstdio>
#include <vector>
#include <algorithm>
using namespace std;

int main() {
  int N, K=0;
  vector<pair <int,int> > A;

  scanf ("%d", &N);
  for (int i=0; i<N; ++i) {
    int x; scanf ("%d", &x);
    A.push_back (make_pair (x, i));
  }
  sort (A.begin(), A.end());
```

```
  for (int i=0; i<N; ++i) { A[i].first = A[i].second; A[i].second = i+1; }
  sort (A.begin(), A.end());

  vector<bool> B(N+2, false);
  for (int i=0; i<N; ++i) {
    int x = A[i].second;
    if (!(B[x-1] || B[x+1])) ++K;
    B[x] = true;
  }

  printf ("%d\n", K);
}
```

A-II-2 Dlhopisy

Prvá vec, ktorú sme si mohli všimnúť je to, že na konci roka (po tom, ako Kleofáš dostane výnosy) vždy môže všetky dlhopisy predať a nakúpiť nové. Vždy by ich mal nakúpiť tak, aby na konci roka dostal čo najvačšie výnosy. Algoritmus bude vyzerať nasledovne:

1. Na začiatku roka nakúp čo najlepšie.
2. Na konci roka vyber výnosy a predaj všetky dlhopisy.
3. Ak chcem ešte jeden rok pokračovať, tak choď na krok 1.

Ako výhodne nakupovať?:

Pre jednoduchosť vyjadrovania si najkôr zaveďme nejaké označenia. $V[i]$ odteraz označuje najväčšie výnosy, ktoré vieme dosiahnuť ak máme i peňazí. c_i bude cena i-teho dlhopisu a v_i je ročný výnos i-teho dlhopisu. Keď dostaneme obnos peňazí K, tak chceme zistiť $V[K]$.

Zjavne je jedno v akom poradí kupujeme dlhopisy, zaujíma nás len výsledná množina. Niektorý dlhopis ale musíme kúpiť ako prvý. Aké vieme mať najvyššie výnosy, ak by to bol dlhopis číslo i? Predsa $v_i + V[K - c_i]$. Ľudovou rečou povedané, kúpime najskôr i-ty dlhopis (výnos z neho bude v_i), a ostane nám $K - c_i$ peňazí. Za tie nakúpime čo najvýhodnejšie.

Ako teda zistiť, ktorý dlhopis kúpiť ako prvý? Vyskúšame všetky možné i a vyberieme ten, pre ktorý bude celkový výnos najväčší. Teda dostávame vzťah:

$$V[K] = \max\Big\{V[K - c_1] + v_1,\ V[K - c_2] + v_2,\ \ldots,\ V[K - c_D] + v_D\Big\}$$

(Samozrejme, maximum berieme len cez tie hodnoty, kde $K \geq c_i$, teda berieme do úvahy len tie dlhopisy, ktoré za K peňazí môžeme kúpiť.)

Všimnite si, že na vypočítanie $V[K]$ potrebujeme poznať $V[K-c_1], V[K-c_2], \ldots, V[K-c_D]$. Ako vypočítame tie? Na to si pomôžeme postupom, ktorý sa volá dynamické programovanie – začneme od najmenších hodnôt. Určite vieme, že $V[0] = 0$. Teraz si vypočítame $V[1]$. Potom $V[2]$. Takto budeme postupne pokračovať, až nakoniec dostaneme k výpočtu $V[K]$. Každú hodnotu sme vypočítali z predchádzajúch hodnôt, ktoré sme už tou dobou poznali. A teda aj v čase, keď sme sa dostali k výpočtu $V[K]$, sme už mali vypočítané všetky potrebné hodnoty.

Tiež si môžeme všimnúť, že hodnota K predstavuje len hornú hranicu, po ktorú pole vypĺňame. Samotný obsah poľa od nej nezávisí. Inými slovami, keď sa nám zmení finančná situácia, nepotrebujeme preratúvať celé pole V odznova.

Aký najväčší index v poli V nás bude zaujímať? Máme dve rovnocenné možnosti: buď si ho na začiatku odhadnúť a rovno spočítať dosť veľa hodnôt, alebo pole V dopočítavať podľa potreby po každom roku.

Prvý prístup: V zadaní sa dalo dočítať, že výnos z dlhopisu je najviac 10% z ceny dlhopisu. Teda za rok vie Kleofáš znásobiť svoj majetok maximálne o desať percent. Za R rokov teda Kleofášov majetok narastie nanajvýš na $K \cdot 1.1^R$.

V zadaní sa tiež môžeme dočítať, že ceny dlhopisov sú násobky $T = 1000$. Teda ak napr. má Kleofáš 14 947 korún, nemôže si nakúpiť nič iné ako to, čo zvládne nakúpiť za 14 000 korún. Preto nám budú stačiť hodnoty $V[i]$ pre násobky T. Budeme teda potrebovať vypočítať $K \cdot 1.1^R/T$ hodnôt $V[i]$.

Po vypočítaní týchto hodnôt už len R krát zopakujeme postup: „čo najlepšie nakúpiť a na konci roka vybrať výnosy“.

Aká je časová zložitosť nášho algoritmu? Potrebujeme si predrátať hodnoty $V[i]$, čo vieme spraviť v čase $O(D \cdot K \cdot 1.1^R/T)$. Zvyšok algoritmu beží v zanedbateľnom čase $O(R)$.

Listing programu:

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

int main(){
    int K, D, R, T = 1000;
    cin >> K >> D;
    vector<int> ceny(D), vynosy(D);
```

```
    for (int i=0; i<D; i++) {
      cin >> ceny[i] >> vynosy[i];
      ceny[i] /= T;
    }
    cin >> R;
    int maxV = 2 + int( K*pow(1.1,R) / T );
    vector<int> V(maxV);
    for (int i=0; i<maxV; i++)
        for (int j=0; j<D; j++)
            if (i-ceny[j] >= 0)
                V[i] = max( V[i], V[i-ceny[j]]+vynosy[j] );
    for (int i=0; i<R; i++) K += V[K/T];
    cout << K << endl;
}
```

A-II-3 O víťazovi turnaja

Základné pozorovanie:
Začneme základným pozorovaním, ktoré povedie k efektívnemu riešeniu úlohy:

♡: Ak program x môže vyhrať turnaj, a program y môže vyhrať v zápase s programom x, tak aj program y môže vyhrať celý turnaj.

Toto pozorovanie si teraz dokážeme. Zoberme si nejaké poradie zápasov a výsledkov, ktoré vedie k tomu, že x vyhrá turnaj. V práve jednom z týchto zápasov y prehrá a z turnaja vypadne. Predstavme si teraz, že odohráme všetky zápasy presne rovnako, až na to, že vynecháme zápas, v ktorom by y vypadol. Dostaneme poradie zápasov, ktoré keď sa zahrajú, ostanú v turnaji len dvaja hráči – x a y. No a keďže y môže vyhrať nad x, stačí teraz pridať na koniec zápas, v ktorom y porazí x.

Naše pozorovanie ♡ vieme teda použiť na postupné zisťovanie ďalších a ďalších programov, ktoré turnaj vyhrajú. Potrebujeme ale niekde začať – vedieť aspoň jeden program, ktorý môže turnaj vyhrať. Ako ho zistiť? To je ľahké – stačí jednoducho jeden možný turnaj odsimulovať a zobrať jeho víťaza.

Máme teda už prvú predstavu, ako môže naše vzorové riešenie vyzerať:

1. Odsimuluj jeden turnaj a nájdi prvého možného víťaza.
2. Používaním ♡ pridávaj nových možných víťazov.

Nájdeme všetkých víťazov?:

Z toho, čo sme zatiaľ povedali, ešte nevyplýva odpoveď na jednu dôležitú otázku: Nájdeme naším postupom naozaj **všetky** programy, ktoré môžu turnaj vyhrať?

Zjavne ♡ môžeme použiť len konečne veľa krát. Skôr či neskôr sa dostaneme do situácie, že už použitím ♡ nevieme o žiadnom ďalšom programe povedať, že môže vyhrať turnaj.

V tomto okamihu máme programy rozdelené do dvoch množín. V prvej (nazvime ju V ako víťazi) sú programy, o ktorých už vieme, že môžu turnaj vyhrať. V druhej (tú nazveme P ako porazení) sú ostatné programy.

Keďže použitím ♡ nevieme žiaden program presunúť z P do V, znamená to, že každý program z V nutne vyhrá nad každým programom z P.

Teraz tvrdíme, že žiaden program, ktorý je v tomto okamihu v P, turnaj nemôže vyhrať.

Prečo je to tak? Všimnime si priebeh ľubovoľného turnaja. Skôr či neskôr ostane v turnaji z programov z množiny V už len jeden. No a ten už nemá ako vypadnúť – všetky ostatné programy, ktoré sú ešte v turnaji, s ním totiž prehrajú. Preto tento program nutne turnaj vyhrá. Víťaz je teda vždy z nami zostrojenej množiny V.

Dokázali sme teda, že algoritmus „kým sa dá použiť ♡, pridávaj do V nové programy“ naozaj nájde všetky programy, ktoré môžu vyhrať turnaj. Zostáva zodpovedať otázku, ako efektívne vieme tento algoritmus implementovať.

Podrobnejší popis algoritmu:

Prvým krokom nášho programu bude simulácia jedného turnaja. Tým dostaneme jedného víťaza v.

Počas zvyšku algoritmu budeme postupne zostojovať množinu víťazov V. Zároveň s tým si budeme udržiavať aj množinu P tých programov, ktoré s každým programom z aktuálnej množiny V nutne prehrajú.

Všimnime si, že zatiaľ čo množinu V budeme zväčšovať, množina P sa nikdy zväčšiť nemôže. Keď pridáme nový program do V, jediné, čo sa stane, je, že z P ubudnú tie programy, ktoré nad ním môžu vyhrať.

Tiež si všimnime, že až kým náš algoritmus neskončí, budú existovať programy, ktoré nie sú ani vo V, ani v P. O týchto už vieme, že môžu vyhrať nad niektorým programom z V, ale ešte sme ich do V nepridali. Aby sme ich nemuseli zakaždým hľadať, budeme si tieto programy čakajúce na spracovanie pamätať vo fronte S.

Keď teda zistíme prvého možného víťaza, nastavíme si $V = \{v\}$. Ďalej P

bude množina programov, s ktorými v určite vyhrá. No a všetky ostatné programy vložíme do fronty S. Kým fronta S nie je prázdna, dokola opakujeme:

1. Vyberieme program x z fronty S.
2. Pridáme program x do množiny V.
3. Prejdeme prvky v P. Tie, nad ktorými x nevyhrá, presunieme z P do S.

Akonáhle sa fronta S vyprázdni, máme všetky programy rozdelené do V a P a môžeme skončiť.

Prvé kvadratické riešenie:

V tomto okamihu je už ľahké napísať program s časovou zložitosťou $O(N^2)$.

Načítame vstup a do dvojrozmerného poľa si zaznačíme, kto nad kým môže vyhrať. (Na začiatku môže vyhrať každý nad každým, ako postupne čítame vstup, značíme si, kto nad kým vyhrať nemôže.)

Odsimulujeme jeden turnaj tak, že najskôr hrá 1 s 2, potom víťaz s 3, a tak ďalej až kým nezostane len jeden program.

Teraz opakujeme dokola cyklus z predchádzajúcej časti. Pridať nový prvok do x vieme v konštantnom čase a upraviť množinu P v čase $O(N)$. (Na to si stačí množinu P reprezentovať ako pole, kde máme o každom prvku zaznačené, či do P patrí alebo nie.)

Keďže v každom opakovaní cyklu pridáme do V jeden program, bude opakovaní najviac $N - 1$. A teda celková časová zložitosť bude $O(N^2)$.

Aby sme dosiahli lepšiu časovú zložitosť, potrebujeme zlepšiť dve miesta v našom programe: Prvou je vedieť aj bez pamäte $O(N^2)$ efektívne odsimulovať turnaj, druhou je šikovnejšie upraviť množinu P.

Prienik dvoch utriedených postupností:

Majme dve postupnosti A a B celých čísel, pričom obe sú utriedené podľa veľkosti a žiadna neobsahuje to isté číslo dvakrát. Chceme nájsť ich prienik, teda tie čísla, ktoré sa vyskytujú v oboch postupnostiach.

Riešenie je ľahké: Začnime tým, že sa pozrieme na prvé členy postupností. Ak sú oba rovnaké, dáme príslušnú hodnotu na výstup a oba zahodíme. Ak je jeden z nich menší, môžeme ho rovno zahodiť – v druhej postupnosti taká hodnota určite nebude. Toto celé teraz opakujeme dokola až kým sa nám jedna z postupností neminie.

Každý krok vieme spraviť v konštantnom čase. (Postupnosti A a B v skutočnosti nebudeme meniť, len si budeme pamätať, kde v ktorej z nich je práve

prvý nevyhodený prvok.) A keďže v každom kroku niečo vyhodíme, bude počet krokov nanajvýš rovný súčtu dĺžok postupností.

Reprezentácia množiny porazených:

Ako sme už naznačili, množinu P si budeme pamätať ako utriedené pole čísel programov, ktoré do nej patria.

Na začiatku toto máme „zadarmo“, programy, nad ktorými v vyhral, máme presne v tejto podobe dané na vstupe.

Keď teraz pridávame do V nový program, spravíme vyššie uvedeným postupom prienik doterajšej množiny P a množiny programov, nad ktorými pridávaný program vyhrá.

Čo sme týmto získali? Ukážeme, ako dlho budú trvať dokopy všetky zmeny množiny P. Nech q_i je počet programov, o ktorých máme na vstupe povedané, že nad programom i vyhrajú. Zjavne $q_1 + \cdots + q_N = M$. Všimnime si teraz ľubovoľný program p, ktorý sme na začiatku umiestnili do množiny P. Tento program v nej vydrží najviac q_p krokov, a teda ho budeme spracúvať najviac $q_p + 1$ krát.

Dokopy všetky prvky z P budeme teda spracúvať nanajvýš $(q_1 + 1) + \cdots + (q_N + 1) = M + N$ krát. Preto celková časová zložitosť udržiavania množiny P bude $O(M + N)$.

Simulácia turnaja:

Posledné, čo potrebujeme, je efektívne odsimulovať jeden turnaj, pričom si informácie o tom, kto s kým musí vyhrať, budeme pamätať len v podobe, v akej sú dané na vstupe.

Budeme mať jedno pole, v ktorom si pre každý program pamätáme, či ešte je v turnaji alebo už vypadol. Na začiatku nastavíme, že kandidátom k na výhru v turnaji je program 1. Z turnaja vyhodíme všetky programy, nad ktorými program 1 určite vyhrá.

Teraz budeme postupne prechádzať všetky programy. Zakaždým, keď narazíme na nejaký, ktorý ešte nevypadol, spravíme nasledovné veci:

V prvom rade, práve nájdený program p môže vyhrať nad doterajším kandidátom k. (Tie, ktoré nad k vyhrať nemôžu, sme už vyhodili.) Doterajšieho kandidáta z turnaja vyhodíme, odteraz bude kandidátom na výhru náš program p. A samozrejme, teraz z turnaja vyhodíme všetky programy, nad ktorými náš nový kandidát určite vyhrá. (Teda presnejšie, vyhodíme tie z nich, ktoré sme ešte nevyhodili.)

Keď takto prejdeme cez všetky programy, ostane nám už v turnaji len jeden

program (aktuálny kandidát), a ten teda turnaj vyhral.

Zjavne sa na každý program pozrieme práve raz a každú informáciu zo vstupu použijeme najviac raz, preto je časová zložitosť simulácie $O(M + N)$.

Alternatívne vzorové riešenie:

V čase $O(M + N)$ sa dá programy zoradiť do poradia, v ktorom platí, že každý program môže vyhrať nad programom, ktorý nasleduje bezprostredne po ňom.

Jedna možnosť ako to spraviť: Postupne pridávame programy do poradia. Nech je aktuálne poradie $p_1, \ldots, p_k$ a práve pridávame nový program p. Nájdeme najväčšie také j, že p_j môže vyhrať nad p (teda nepatrí medzi programy, nad ktorými p určite vyhrá). Umiestnime p do poradia tesne za p_j. Jediný špeciálny prípad: ak zistíme, že p porazí všetky p_i, dáme ho na začiatok poradia.

Zjavne máme opäť korektné poradie – p_j sme vybrali tak, aby vedelo vyhrať nad p, a navyše p určite vyhrá nad každým napravo od seba, teda aj s programom bezprostredne za ním (ak taký existuje).

Toto poradie vieme zostrojiť v čase $O(M + N)$, dokonca si na to vystačíme s dvoma obyčajnými poliami. V poli P na pozíciách 1 až k si budeme pamätať aktuálne poradie programov. Okrem toho budeme mať jedno pole B, v ktorom si o každom programe budeme pamätať, či nad ním aktuálny program p určite vyhrá alebo nie. Aby sme nemuseli pole B pri každom novom p celé nulovať, použijeme jeden jednoduchý trik.

Algoritmus bude fungovať nasledovne:

1. Vynuluj pole B, nastav k na 1 a $P[1]$ na 1.
2. Postupne pre $x = 2$ až N opakuj nasledujúce kroky:
3. Prejdi zoznam d_i programov, nad ktorými program x vyhrá.
   Do poľa B na príslušné pozície zapíš číslo x (toto je spomínaný trik).
4. Postupne od konca prechádzaj pole P, kým neprídeš na pozíciu j takú, že $B[P[j]] \neq x$ alebo na začiatok (kedy bude $j = 0$).
5. Prvky v poli P na pozíciách $j + 1$ až k posuň o 1 doprava.
   Do $P[j]$ ulož x a zväčši k o 1.

(Všimnite si, že všetky kroky 3 dokopy trvajú $O(d_1 + \cdots + d_n) = O(M)$, a že každý krok 4 trvá rádovo toľko isto ako jemu predchádzajúci krok 3.)

Pozrime sa teraz na naše poradie programov. Ak nejaký program môže vyhrať turnaj, môžu vyhrať turnaj aj všetky programy, ktoré sú v poradí pred ním. Takže stačí nájsť hranicu takú, že naľavo od nej sú programy, ktoré turnaj vyhrať môžu, a napravo tie, ktoré vyhrať nemôžu.

Na hľadanie hranice opäť použijeme pozorovanie ♡, len sa nám tentokrát bude ľahšie implementovať. Keď teraz spracúvame nejaký program p, o ktorom vieme, že môže vyhrať turnaj, stačí nám nájsť v našom poradí prvý program q od konca, ktorý vie nad p vyhrať. Program q aj všetky naľavo od neho vedia turnaj vyhrať. Ak q leží napravo od doteraz nájdenej hranice, príslušne ju posunieme. No a hľadanie programu q je jednoduché – sú to opäť presne kroky 3 a 4 z vyššie uvedeného algoritmu.

Dokopy bude teda táto druhá fáza vyzerať nasledovne:

1. Nastav index k posledného nám známeho vyhrávajúceho programu na 1. Nastav index x vyhrávajúceho programu, ktorý treba spracovať, na 1.
2. Ak $x > k$, už sme spracovali všetky vyhrávajúce programy a nič nové sa nedozvieme, takže vieme, že turnaj môže vyhrať práve prvých k programov z nášho poradia.
3. Nájdi index j prvého programu od konca poradia, ktorý môže vyhrať nad programom $P[x]$. (Ak $P[x]$ nutne vyhrá nad každým napravo od seba, bude $j = x$.)
4. Ak $j > k$, nastav k na j. Zväčši x o 1 a pokračuj krokom 2.

Pomalšie riešenia:

Na dvojrozmerné pole, v ktorom máme zaznačené, kto nad kým môže vyhrať, sa môžeme dívať ako na maticu susednosti orientovaného grafu.

Vo vzorovom riešení sme zdôvodnili, že ak v vie vyhrať turnaj, tak množinu všetkých víťazov dostaneme tak, že zoberieme:

1. v
2. každého kto vie vyhrať nad v
3. každého, kto vie vyhrať nad niekým z kroku 2
4. ...

V grafovej terminológii to vyjadríme ľahšie: možní víťazi turnaja sú práve tie programy, z ktorých je v našom grafe dosiahnuteľný vrchol v.

Druhé kvadratické riešenie teda dostávame nasledovne: Podobne ako v prvom riešení simuláciou turnaja nájdeme prvé v. Teraz otočíme všetky hrany a prehľadávaním (do hĺbky alebo do šírky) z v nájdeme všetky programy, ktoré môžu turnaj vyhrať.

V podobnom duchu môžeme dostať riešenie s časovou zložitosťou $O(N^3)$. Na takéto riešenie nepotrebujeme ani len pozorovanie ♡. Iba si stačí všimnúť (a

dokázať), že vo vyššie zostrojenom grafe platí: Program p môže vyhrať turnaj práve vtedy, ak je z p dosiahnuteľný každý vrchol v grafe. Postupne pre každý vrchol teda spustíme prehľadávanie, ktorým nájdeme všetky z neho dosiahnuteľné vrcholy.

Listing programu:

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

int N; // pocet programov
vector< vector<int> > vyhra; // pre kazdy program zoznam tych, nad ktorymi vyhra

void nacitaj() { // nacita vstup
  cin >> N;
  vyhra.resize(N+1);
  for (int i=1; i<=N; i++) {
    int d;
    cin >> d;
    vyhra[i].resize(d);
    for (int j=0; j<d; j++) cin >> vyhra[i][j];
  }
}

int simuluj() {
  vector<bool> hra(N+1,true); // o kazdom programe, ci este hra v turnaji
  int kandidat = 1;
  for (unsigned i=0; i<vyhra[kandidat].size(); i++)
    hra[ vyhra[kandidat][i] ] = false;
  for (int dalsi=2; dalsi<=N; dalsi++)
    if (hra[dalsi]) {
      hra[ kandidat ] =0;
      kandidat = dalsi;
      for (unsigned i=0; i<vyhra[kandidat].size(); i++)
        hra[ vyhra[kandidat][i] ] = false;
    }
  return kandidat;
}

int main() {
  nacitaj();
  // najdeme prveho mozneho vitaza
  int v = simuluj();

  // inicializujeme V a P
  vector<int> V, P;
```

```
  V.push_back(v);
  P = vyhra[v];

  // do fronty S vlozime vsetky prvky, co nie su vo V ani P
  queue<int> S;
  vector<int> tmp(N+1);
  tmp[v] = 1;
  for (unsigned i=0; i<P.size(); i++) tmp[P[i]] = 1;
  for (int i=1; i<=N; i++) if (!tmp[i]) S.push(i);

  // kym mame nieco vo fronte S, pridame to do V a preratame P
  while (!S.empty()) {
    int x = S.front(); S.pop();
    V.push_back(x);
    // ideme preratat P + co vyhodime z P, ide do S
    vector<int> newP;
    unsigned a=0, b=0;
    while (a<P.size() && b<vyhra[x].size()) {
      if (P[a] == vyhra[x][b]) { newP.push_back(P[a]); a++; b++; continue; }
      if (P[a] < vyhra[x][b]) { S.push(P[a]); a++; continue; }
      if (P[a] > vyhra[x][b]) { b++; continue; }
    }
    P = newP;
  }
  for (unsigned i=0; i<V.size(); i++) cout << V[i] << endl;
}
```

A-II-4 Prekladacie stroje

Podúloha a):

Použijeme postup podobný tomu z domáceho kola.

V prvom kroku zostrojíme prekladací stroj, ktorý kopíruje vstup na výstup a navyše na ľubovoľné miesto v reťazci vie dopísať ľubovoľne veľa znakov c. Tento stroj môže vyzerať nasledovne:

$$\begin{aligned} Z_1 &= (K_1, \Sigma, P_1, \spadesuit, F_1) \\ \Sigma &= \{a, b, c\} \\ K_1 &= \{\spadesuit\} \\ F_1 &= \{\spadesuit\} \\ P_1 &= \{(\spadesuit, a, a, \spadesuit),\ (\spadesuit, b, b, \spadesuit), (\spadesuit, \varepsilon, c, \spadesuit)\} \end{aligned}$$

Množina $M_2 = Z_1(M_1)$ teda obsahuje práve všetky reťazce z písmen a, b, c, kde je rovnako veľa a a b.

Analogicky teraz zostrojíme druhý stroj, ktorý vyrobí tie reťazce, kde bude rovnako veľa b a c, a ľubovoľne veľa a. Pre zmenu môžeme túto množinu reťazcov zostrojiť z práve zostrojenej M_2 tak, že cyklicky zameníme všetky a za b, b za c a c za a. Toto robí napríklad tento prekladací stroj:

$$\begin{aligned} Z_2 &= (K_2, \Sigma, P_2, \clubsuit, F_2) \\ \Sigma &= \{a, b, c\} \\ K_2 &= \{\clubsuit\} \\ F_2 &= \{\clubsuit\} \\ P_2 &= \{(\clubsuit, a, b, \clubsuit),\ (\clubsuit, b, c, \clubsuit), (\clubsuit, c, a, \clubsuit)\} \end{aligned}$$

Teraz sme teda zostrojili množinu $M_3 = Z_2(M_2)$.

No a je evidentné, že prienikom množín M_2 a M_3 dostaneme práve hľadanú množinu G.

Podúloha b):

Základným trikom pri riešení tejto úlohy je uvedomiť si, že ak má byť výstupná množina reťazcov dostatočne jednoduchá, vstup vlastne na nič nepotrebujeme – vieme ju vygenerovať aj „bez dopomoci“.

V našom prípade bude teda výsledný prekladací stroj pracovať nasledovne:

1. Prečíta celý vstup a zahodí ho, teda zatiaľ nič nevypíše.
   (Podľa definície stroja toto musíme urobiť, preklad je platný len ak spracujeme celý vstupný reťazec!)

2. Začneme generovať číslo, pričom si v stave pamätáme, aký zvyšok dáva doteraz zapísané číslo po delení siedmimi. Ukončovací stav bude zodpovedať zvyšku 0 – teda prestať generovať môžeme (a nemusíme) práve vtedy, keď sme vygenerovali číslo deliteľné siedmimi.

Treba si dať ešte pozor na to, aby naše číslo nezačínalo nulou, toto vieme ale elegantne ošetriť napríklad v okamihu, keď zistíme, že sme dočítali vstupný reťazec.

Výsledný prekladací stroj bude vyzerať takto:

$$\begin{aligned} Z &= (K, \Sigma, P, read, F) \\ \Sigma &= \{0, 1, \ldots, 9\} \\ K &= \{read\} \cup \{0, 1, \ldots, 6\} \end{aligned}$$

$$
\begin{aligned}
F &= \{0\} \\
P &= \big\{(read, x, \varepsilon, read) \quad \big| \quad \forall x \in \{0, \ldots, 9\}\big\} \ \cup \\
&\cup \ \big\{(read, \$, y, y \bmod 7) \quad \big| \quad \forall y \in \{1, \ldots, 9\}\big\} \ \cup \\
&\cup \ \big\{(x, \varepsilon, y, (10x + y) \bmod 7) \quad \big| \quad \forall x \in \{0, \ldots, 6\},\ \forall y \in \{0, \ldots, 9\}\big\}
\end{aligned}
$$

Slovný popis: Pravidlami v prvom riadku vieme prečítať celé vstupné slovo a nič nezapísať.

Pravidlo v druhom riadku použijeme práve raz, a to po dočítaní vstupného slova. Na výstup zapíšeme prvú cifru čísla (všimnite si, že musí byť kladná) a nastavíme si stav na jej zvyšok po delení siedmimi.

Teraz už môžeme používať len pravidlá v treťom riadku. Každé z nich dopíše na koniec výsledného reťazca nejakú novú cifru y. Matematicky vieme operáciu „pridaj na koniec čísla cifru y“ povedať aj „vynásob číslo desiatimi a pripočítaj k nemu y“. Ak teda doteraz zapísané číslo malo zvyšok po delení siedmimi rovný x, nové číslo bude mať rovnaký zvyšok ako $10x+y$. A presne do zodpovedajúceho stavu prejde náš prekladací stroj.

Z argumentu v predchádzajúcom odseku vyplýva, že každé číslo, ktoré bude vo výslednej množine $Z(X)$, je naozaj deliteľné siedmimi.

No a keďže na výstup vieme vypísať ľubovoľné kladné celé číslo, určite vie náš stroj vyrobiť každé číslo deliteľné siedmimi, a po jeho vypísaní bude určite v ukončovacom stave 0. Preto naozaj $Y = Z(X)$.

Riešenia krajského kola kategórie B

B-II-1 Zberateľky

Ako zistiť, či sú v úseku $B[1..K]$ nejaké z hľadaných čísel? Ľahko. Stačí porovnať hodnoty $B[K]$ a $A[K]$. Ak sú rovnaké, obe hľadané čísla sú niekde ďalej v poli B. Ak je hodnota v poli B menšia, aspoň jedna z hľadaných hodnôt leží v skúmanom úseku.

Vieme toho dokonca povedať aj viac. Tým, že sme pridali dva nové prvky, sa niektoré z pôvodných prvkov posunuli „doprava“ (na pozíciu s väčším indexom), a to najviac o 2. Teda hodnota $A[i]$ je v poli B na pozícii i, $i+1$ alebo $i+2$. A podľa toho, o koľko pozícii je vpravo, vieme, koľko z hľadaných nových prvkov je v poli B naľavo od nej.

Pomocou tejto úvahy vieme ľahko zistiť, koľko hľadaných čísel je v úseku $B[1..K]$: Pozrime sa na hodnotu $B[K]$. Sú dve možnosti: Ak je aj v poli A, tak musí nutne byť na jednej z pozícii $A[K-2]$, $A[K-1]$ alebo $A[K]$. A podľa toho, na ktorej z nich je, vieme, koľko hľadaných čísel je naľavo. Ak táto hodnota v poli A nie je, je to jedno z hľadaných čísel. Zopakovaním úvahy pre $B[1..(K-1)]$ zistíme, či je v tejto časti poľa aj druhé hľadané číslo.

Podobne vieme zistiť počet hľadaných čísel v ľubovoľnom úseku $B[k..l]$ – spočítame ho jednoducho ako (počet v $B[1..l]$) mínus (počet v $B[1..(k-1)]$).

Všimnite si, že na zodpovedanie takejto otázky nám stačí konštantný počet operácií.

Vzorové riešenie bude používať myšlienku binárneho vyhľadávania.

Majme najskôr jednoduchší problém: Máme úsek poľa B, o ktorom vieme, že v ňom je práve jedno hľadané číslo. Ako ho nájsť?

Ak má úsek dĺžku 1, už sme ho našli. Ak je dlhší, rozdeľme ho približne na polovice. Teraz si vieme v konštantnom čase spočítať, v ktorej polovici hľadané číslo je. Dostali sme tú istú úlohu, len na úseku polovičnej dĺžky. Tento postup teda opakujeme dovtedy, kým nám nezostane už len úsek dĺžky 1.

Ak sme teda začali s úsekom dĺžky K, po približne $\log_2 K$ krokoch nájdeme hľadané číslo.

A čo teraz s pôvodným problémom? Majme teda úsek poľa B, o ktorom vieme, že sú v ňom obe hľadané čísla. Opäť ho rozdelíme na polovice. Môžu

nastať dva prípady:

Ak sú obe čísla v tej istej polovici, máme pôvodný problém a úsek polovičnej dĺžky, pokračujeme ďalej v delení na polovice.

Ak je každé číslo v inej polovici, dostali sme dva jednoduchšie problémy, ktoré už vieme riešiť.

Časová zložitosť tohoto riešenia je $O(\log N)$ – v najhoršom prípade rozdelíme kus poľa na dve časti $2 \log_2 N$ krát.

Listing programu:

```
var A,B : array[1..10000] of longint;
    N : longint;


{ spocita kolko hladanych cisel je v B[1..(kon-1)] }
function spocitaj(kon: longint) : longint;
begin
   if (kon=1) then begin spocitaj:=0; exit; end;
   if (kon>2) and (A[kon-3]=B[kon-1]) then begin spocitaj:=2; exit; end;
   if (kon>1) and (kon<N+3) and (A[kon-2]=B[kon-1]) then
      begin spocitaj:=1; exit; end;
   if (kon<N+2) and (A[kon-1]=B[kon-1]) then begin spocitaj:=0; exit; end;
   { ak sa dostal sem, B[kon-1] je jedno z hladanych cisel }
   spocitaj := 1 + spocitaj(kon-1);
end;

{ spocita kolko hladanych cisel je v B[zac..(kon-1)] }
function spocitaj(zac, kon: longint) : longint;
begin spocitaj := spocitaj(kon) - spocitaj(zac); end;

procedure najdi_jeden(zac, kon: longint; var x : longint);
var stred,prvy : longint;
begin
   if (kon-zac=1) then begin x:=B[zac]; exit; end;
   stred:=(zac+kon) div 2;
   prvy:=spocitaj(zac,stred);
   if (prvy=0) then najdi_jeden(stred,kon,x) else najdi_jeden(zac,stred,x);
end;

procedure najdi_dva(zac, kon: longint; var x,y : longint);
var stred,prvy : longint;
begin
   stred:=(zac+kon) div 2;
   prvy:=spocitaj(zac,stred);
   if (prvy=0) then begin najdi_dva(stred,kon,x,y); exit; end;
   if (prvy=2) then begin najdi_dva(zac,stred,x,y); exit; end;
   najdi_jeden(zac,stred,x);
```

```
   najdi_jeden(stred,kon,y);
end;

{ pre nazornost este priklad programu, ktory nasu proceduru vola }
var i,x,y : longint;
begin
   read(N);
   for i:=1 to N do read(A[i]);
   for i:=1 to N+2 do read(B[i]);
   najdi_dva(1,N+3,x,y);
   writeln(x,' ',y);
end.
```

B-II-2 Krtkovou norou tam a späť

Použijeme postup z domáceho kola. Do každej chodbičky pridáme nové brlôžky, ktoré ju rozdelia na úseky dĺžky 1.

Prehľadávaním do šírky z brlôžku 1 teraz vieme v čase $O(M+N)$ pre každý brlôžok x spočítať dĺžku $d(1,x)$ najkratšej cesty doň. Okrem iného sa teda dozvieme dĺžku $d(1,N)$ cesty, ktorou má Vítek ísť.

Spustíme ešte jedno prehľadávanie do šírky, tentokrát z brlôžku N. Tým sa dozvieme dĺžky najkratších ciest medzi brlôžkom N a ľubovoľným brlôžkom x.

Ako teraz zistiť, či vie Vítek ísť cez konkrétny brlôžok x, ak ide najkratšou cestou do N? Jednoducho. Pozrime sa, aká najkratšia môže byť jeho cesta, ak pôjde cez x. V prvom rade musí prísť najkratšou cestou z 1 do x. No a následne musí prejsť, opäť najkratšou cestou, z x do N.

Ak teda Vítek pôjde najkratšou cestou, ktorá vedie cez brlôžok x, prejde trasu dĺžky $d(1,x)+d(x,N)$. Ak je táto hodnota rovná $d(1,N)$, Vítek cez x ísť môže. A naopak, ak je dĺžka cesty cez x väčšia od $d(1,N)$, Vítek cez x ísť nemôže.

Takto vieme teda o každom brlôžku v konštantnom čase zistiť, či cez neho môže Vítek ísť. Táto časť algoritmu má teda časovú zložitosť $O(N)$, a keďže prehľadávania trvajú dlhšie, celková časová zložitosť je $O(M+N)$.

Listing programu:

```
var G : array[1..1000,1..1000] of longint; { pre kazdy vrchol zoznam susedov }
    stupne : array[1..1000] of longint; { stupne vrcholov }
    N,M : longint; { pocet vrcholov a hran }
    dist1, distN : array[0..1000] of longint; { vzdialenosti od 1 a od N }

procedure nacitaj;
```

```
var i,a,b,h,kde,dalsi : longint;
begin
   readln(N,M);
   dalsi := N+1; { cislo, ktore dostane dalsi brlozok }
   for i:=1 to M do begin
      readln(a,b,h);
      kde := a;
      while (h>1) do begin { kopeme novy brlozok }
         inc(stupne[kde]); inc(stupne[dalsi]);
         G[kde][ stupne[kde] ] := dalsi;
         G[dalsi][ stupne[dalsi] ] := kde;
         kde := dalsi; inc(dalsi);
         dec(h);
      end;
      inc(stupne[kde]); inc(stupne[b]);
      G[kde][ stupne[kde] ] := b;
      G[b][ stupne[b] ] := kde;
   end;
end;

{ spusti prehladavanie z brlozka "odkial", v poli "vzdialenost" je vystup }
procedure prehladaj(odkial : longint; var vzdialenost : array of longint);
var fronta : array[1..1000] of longint;
    bol : array[1..1000] of boolean;
    zac,kon,kde,i,kam : longint;
begin
   zac:=1; kon:=2; fronta[1]:=odkial;
   fillchar(bol,sizeof(bol),0);
   bol[odkial]:=true; vzdialenost[odkial]:=0;
   while (zac < kon) do begin
      kde := fronta[zac]; inc(zac);
      for i:=1 to stupne[kde] do begin
         kam := G[kde][i];
         if (not bol[kam]) then begin
            bol[kam]:=true; vzdialenost[kam]:=vzdialenost[kde]+1;
            fronta[kon]:=kam; inc(kon);
         end;
      end;
   end;
end;

var i : longint;
begin
   nacitaj;
   prehladaj(1,dist1);
   prehladaj(N,distN);
   for i:=1 to N do
      if dist1[i]+distN[i] = dist1[N] then writeln(i);
end.
```

B-II-3 Opravovanie XML

Na pamätanie si otvorených tagov môžeme použiť, podobne ako v riešeniach domáceho kola, zásobník. Hlavným problémom bude, ako pri spracúvaní zatváracieho tagu zistiť, či nastal prípad 2 (máme niekde v zásobníku zodpovedajúci otvárací tag) alebo prípad 3 (takýto otvárací tag nemáme).

Základné riešenie:

Každý otvárací tag vložíme na zásobník. Keď nám príde zatvárací tag, prezrieme obsah zásobníka, a podľa toho, či ho tam nájdeme, buď upravíme obsah zásobníka, alebo zahodíme spracúvaný zatvárací tag.

Ak je na vstupe N tagov, takéto riešenie môže spraviť až rádovo N^2 porovnaní tagov, teda jeho časová zložitosť je $O(N^2)$. (Príklad zlého vstupu: najskôr $N/2$ otváracích tagov, potom $N/2$ zatváracích tagov, ktoré nemajú zodpovedajúci otvárací tag.)

Lepšie riešenie:

Aby sme vedeli toto riešenie zefektívniť, potrebujeme lepší spôsob, ako zistiť, či je daný tag práve otvorený. Existuje viacero spôsobov, ako na to.

Jednou možnosťou napríklad je pamätať si otvorené tagy okrem zásobníka aj vo vyváženom binárnom vyhľadávacom strome. Takto vieme každý tag spracovať v čase $O(\log N)$. Takéto riešenie je však dosť náročné na implementáciu. Ukážeme si preto aj jednoduchší spôsob, ako túto časovú zložitosť dosiahnuť.

Všetky názvy tagov z dokumentu (otváracích aj zatváracích) si uložíme do poľa. Pole utriedime a vynecháme z neho opakujúce sa záznamy. Teraz si môžeme tagy očíslovať číslami od 1 do nejakého M (kde určite $M \leq N$) – každému tagu priradíme index, na ktorom je jeho názov v tomto utriedenom poli. Kedykoľvek, keď budeme chcieť z názvu tagu určiť jeho číslo, vieme ho zistiť v čase $O(\log N)$ binárnym vyhľadávaním.

Keď sme sa takto pripravili, môžeme začať samotné spracúvanie tagov. Budeme si pamätať otvorené tagy v zásobníku, a navyše si budeme v pomocnom poli ku každému názvu tagu pamätať, koľkokrát je práve otvorený.

Otvárací tag spracujeme ľahko: Zistíme si jeho číslo, zvýšime príslušnú hodnotu v poli a vložíme ho na vrch zásobníka.

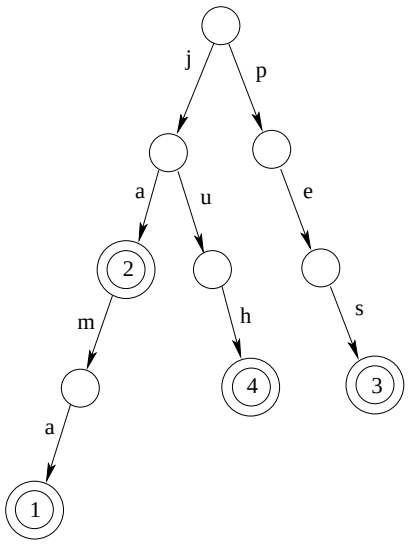
A ako je to so zatváracími tagmi? Zistíme si jeho číslo a pozrieme na príslušné miesto do pomocného poľa. Ak je tam nula, takýto tag nie je otvorený, práve spracúvaný tag teda zahodíme. Ak tam nie je nula, ideme zatvárať otvorené tagy, kým nezavrieme aj tento.

Ako spočítať časovú zložitosť tohoto postupu? Každý otvárací tag raz vložíme na zásobník a raz ho odtiaľ vyberieme. Toto je nanajvýš $2N$ operácií. každú z nich vieme spraviť v čase $O(\log N)$, lebo musíme vždy upraviť aj hodnotu v pomocnom poli, a na to potrebujeme „preložiť“ tag na číslo.

Okrem toho už len každý zatvárací tag potrebujeme preložiť na číslo, keď chceme zistiť, či máme v zásobníku k nemu pár. Takto budeme prekladať najviac N tagov. Preto celková časová zložitosť tohoto riešenia je $O(N \log N)$.

Vzorové riešenie:

Zatiaľ sme nijako nevyužili, že názvy tagov sú reťazce, ktoré sa skladajú z písmeniek. Vďaka tomu existujú aj šikovnejšie spôsoby, ako si pamätať množinu reťazcov. Popíšeme jeden z nich, tzv. písmenkový strom (po anglicky *trie*). Písmenkový strom je zakorenený strom, v ktorom každý vrchol má najviac 26 synov, a hrany do synov sú označené rôznymi písmenkami (od `a` po `z`). Každá cesta z koreňa nadol zodpovedá slovu, ktoré si „prečítame“ na hranách, po ktorých ideme.



Písmenkový strom pre `jama`, `ja`, `pes` *a* `juh`.

Písmenkový strom sa dá použiť na uloženie množiny slov. Začneme so stromom, ktorý obsahuje len koreň a nič viac. Teraz budeme postupne po jednom pridávať slová. Pri pridávaní slova vytvoríme všetky vrcholy, ktoré treba na to, aby sme si mohli na ceste z koreňa dole „prečítať“ toto slovo. (Teda pri vkladaní „`jama`“ pridáme 4 nové vrcholy, pri následnom vkladaní „`ja`“ žiadny.) Vo vrchole, kde slovo končí, si zapíšeme jeho poradové číslo.

Vzorové riešenie teda bude vyzerať nasledovne:

- Načítame zo vstupu postupnosť tagov.
- Názvy tagov postupne vkladáme do písmenkového stromu. Vždy, keď vložíme nový názov (ktorý tam dovtedy nebol), priradíme mu nové číslo.
- Postupne spracúvame postupnosť tagov ako v predchádzajúcom riešení, pričom na preklad tagov na čísla používame náš písmenkový strom.

Čas potrebný na vloženie nového reťazca do písmenkového stromu je priamo úmerný dĺžke dotyčného reťazca. Podobne aj čas potrebný na nájdenie čísla

priradeného reťazcu. V našom prípade (názvy tagov majú dĺžku nanajvýš 8) vieme teda každý tag spracovať v konštantnom čase.

Preto je celková časová zložitosť vzorového riešenia $O(N)$.

Listing programu:

```
program oprava_xml_pismenkovy_strom;

type pvrchol = ^tvrchol;
     tvrchol = record
        cislo : longint;
        deti : array['a'..'z'] of pvrchol;
     end;

procedure inicializuj(co: pvrchol);
var ch : char;
begin co^.cislo:=-1; for ch:='a' to 'z' do co^.deti[ch]:=nil; end;

{ zacni vo vrchole "koren", zlez dole cestou zodpovedajucou "slovo" }
function zlez(slovo: string; koren: pvrchol) : pvrchol;
var i : integer;
begin
   for i:=1 to length(slovo) do begin
      if (koren^.deti[slovo[i]]=nil) then begin
         new( koren^.deti[slovo[i]] );
         inicializuj( koren^.deti[slovo[i]] );
      end;
      koren := koren^.deti[slovo[i]];
   end;
   zlez := koren;
end;

procedure vloz(slovo: string; koren : pvrchol; var N : longint);
begin
   koren := zlez(slovo,koren);
   if koren^.cislo >= 0 then exit;
   koren^.cislo := N;
   inc(N);
end;

function preloz(slovo: string; koren : pvrchol) : longint;
begin
   koren := zlez(slovo,koren);
   preloz := koren^.cislo;
end;

{ skusi nacitat dalsi tag, ak sa podarilo, vyplni premenne a vrati true }
function nacitaj_tag(var nazov : string; var zaciatok : boolean) : boolean;
var ch : char;
begin
```

```
   ch:='?';
   while ch<>'<' do begin
      if eof then begin nacitaj_tag:=false; exit; end;
      read(ch);
   end;
   nazov:=''; zaciatok:=true;
   read(ch);
   if (ch='/') then begin zaciatok:=false; read(ch); end;
   while true do begin
      if (ch='>') then break;
      nazov := nazov + ch;
      read(ch);
   end;
   nacitaj_tag:=true;
end;

{ nacita vstup do danych premennych }
procedure load(var N : longint; var tagy : array of string;
                                var typy : array of boolean);
var tag : string;
    typ : boolean;
begin
   N := 0;
   while (nacitaj_tag(tag,typ)) do
      begin tagy[N]:=tag; typy[N]:=typ; inc(N); end;
end;

var N,M,Z,i,cislo,cislo2,vysledok : longint;
    tagy, zasobnik : array[0..1000] of string;
    typy : array[0..1000] of boolean;
    pocty : array[0..1000] of longint;
    strom : pvrchol;

begin
   load(N,tagy,typy);
   new(strom); inicializuj(strom); M:=0;
   for i:=0 to N-1 do vloz(tagy[i],strom,M);
   Z := 0;
   vysledok := 0;
   for i:=0 to N-1 do begin
      cislo:=preloz(tagy[i],strom);
      if (typy[i]) then begin { otvaraci tag, vlozime na zasobnik }
         zasobnik[Z]:=tagy[i];
         inc(Z);
         inc(pocty[cislo]);
      end else begin { zatvaraci tag }
         if (pocty[cislo]>0) then begin { mame ho, ideme vyberat }
            while true do begin
               cislo2:=preloz(zasobnik[Z-1],strom);
               dec(pocty[cislo2]);
```

```
            dec(Z);
            if (cislo=cislo2) then break;
            inc(vysledok);
          end;
        end else inc(vysledok); { nemame ho, zahodime }
      end;
    end;
    inc(vysledok,Z); { vyhadzeme co ostalo v zasobniku }
    writeln(vysledok);
end.
```

B-II-4 Rákosie sa vracia

Podúloha a):

Použijeme podobný postup ako v jednom z príkladov v študijnom texte. Genetické pravidlá nášho rákosia budú najskôr generovať živé bunky A, B a C, pričom počet C bude stále rovnaký ako počet A a B dokopy. Tieto živé bunky si budú môcť ľubovoľne vymieňať miesta. A keď sa už bunka rozhodne, že na nejakom mieste chce skončiť, zmení sa na zodpovedajúcu mŕtvu bunku.

Pravidlá teda budú vyzerať napríklad takto:

$$Z \to ACZ \mid BCZ \mid \varepsilon \tag{1}$$

$$AB \to BA, \qquad BA \to AB \tag{2}$$

$$AC \to CA, \qquad CA \to AC \tag{3}$$

$$BC \to CB, \qquad CB \to BC \tag{4}$$

$$A \to a, \qquad B \to b, \qquad C \to c \tag{5}$$

Zjavne pre každé steblo, ktoré môže narásť, platí, že počet a plus počet b (veľkých aj malých dokopy) je rovný počtu c (opäť veľkých aj malých dokopy). Toto je preto, že každé pravidlo túto rovnosť zachováva. Pravidlami (2) až (4) nové bunky nepribúdajú, pravidlami (5) sa len živá bunka zmení na zodpovedajúcu mŕtvu, čo rovnosť zjavne nepokazí. No a pri použití pravidla $Z \to ACZ$ alebo $Z \to BCZ$ pribudne jedno c a jedno iné písmenko, čo rovnosť tiež nepokazí. Preto určite každé mŕtve steblo tejto odrody obsahuje rovnako veľa c ako a a b dokopy.

Naopak, majme konkrétny reťazec písmen a, b, c, v ktorom je rovnako veľa c ako a a b dokopy. Nech α je počet písmen a a β počet písmen b v ňom. Keď použijeme α krát pravidlo $Z \to ACZ$, potom β krát pravidlo $Z \to BCZ$ a potom raz $Z \to \varepsilon$, dostaneme reťazec, v ktorom sú už počty A, B a C také, aké majú byť počty a, b a c na konci. Pomocou pravidiel 2-7 vieme všetky A, B a C

preusporiadať do správneho poradia. Na záver ich pravidlami 8-10 prepíšeme na malé písmená a sme hotoví. Preto určite vieme mať mŕtve steblo s ľubovoľným požadovaným reťazcom.

Podúloha b):

Myšlienka bude nasledovná: Vygenerujeme si reťazec písmen X a reťazec písmen Y, každý z nich dĺžky aspoň 2. Teraz by sme chceli ich dĺžky „vynásobiť“. To spravíme takto: Škrtneme jedno X a za každé Y pridáme do reťazca jedno a. Toto opakujeme, kým sa nám všetky X neminú. Na záver už len zmažeme všetky Y.

Aby sme sa v tom celom nezamotali, použijeme ešte niekoľko pomocných symbolov ako zarážky – B (ako begin) na začiatku, E (ako end) na konci.

$$Z \rightarrow BXXP \tag{1}$$

$$P \rightarrow XP \mid YYQ \tag{2}$$

$$Q \rightarrow YQ \mid SE \tag{3}$$

Pomocou postupného aplikovania pravidiel 1, 2 a 3 zo symbolu Z postupne dostaneme reťazec tvaru $BX \ldots XY \ldots YSE$. Vieme vyrobiť ľubovoľný taký reťazec, v ktorom sú aj X, aj Y aspoň dve.

$$aS \rightarrow Sa \tag{4}$$

$$YS \rightarrow SY \tag{5}$$

$$XS \rightarrow T \tag{6}$$

$$Ta \rightarrow aT \tag{7}$$

$$TY \rightarrow YaT \tag{8}$$

$$TE \rightarrow SE \tag{9}$$

S (ako späť) je symbol, ktorý sa vracia späť (pravidlá 4 a 5) na začiatok nájsť ďalšie X. Ak sa mu to podarí, použitím pravidla 6 ho vymaže a zmení sa na T (ako tam). Symbol T použitím pravidiel 7 a 8 presúvame doprava, pričom vždy, keď stretneme Y, pridáme do reťazca nové a.

$$BS \rightarrow F \tag{10}$$

$$Fa \rightarrow aF \tag{11}$$

$$FY \rightarrow F \tag{12}$$

$$FE \rightarrow \varepsilon \tag{13}$$

Ak S prejde celým reťazcom až ku B a žiadne X už nenájde, násobenie sa skončilo a začína upratovanie. Zmeníme použitím pravidla 10 S na F (a

jedným krokom zahodíme už nepotrebnú zarážku B). Použitím pravidiel 11 a 12 prechádza F doprava reťazcom, písmená a necháva na pokoji a zahadzuje všetky Y. Keď F príde na koniec slova, pravidlom 13 zahodíme posledné dva pomocné symboly F a E a skončili sme.

Zjavne keď si zvolíme ľubovoľné zložené číslo $z = mn$, vieme reťazec a^z vyrobiť tak, že pravidlami 1-3 vyrobíme reťazec BX^mY^nSE a odsimulujeme vyššie popísaný postup.

Na druhej strane, v okamihu, kedy použijeme pravidlo $Q \to SE$, je celé pokračovanie jednoznačne určené. V každom okamihu môžeme spraviť práve jednu jedinú zmenu reťazca. Tento postup, ako sme už vyššie ukázali, skončí s tým, že máme reťazec a^z pre nejaké zložené číslo z. Preto v našej odrode rákosia určite nič zlé nenarastie.

Riešenia celoštátneho kola kategórie A

A-III-1 Najväčšia horúčava

Označme si roky zo vstupu $r_1, \ldots, r_n$ a zodpovedajúce teploty $t_1, \ldots, t_n$. Platí $r_1 < \cdots < r_n$.

Začneme tým, že si uvedomíme nasledovnú vec: Ak pre nejaké i a j platí $j < i$ a $t_j \geq t_i$, tak určite rok r_i mohol byť nanajvýš najteplejší od roku r_j, určite nie dlhšie.

Zišlo by sa nám teda pre každý zo zadaných rokov zistiť, ktorý je najbližší skorší rok, v ktorom bolo aspoň tak teplo. Presnejšie, chceme hodnoty $prev_1, \ldots, prev_n$, kde $prev_i$ je najväčšie j, pre ktoré platí $j < i$ a $t_j \geq t_i$ (alebo 0, ak také j neexistuje).

Podobne sa nám budú hodiť hodnoty $next_1, \ldots, next_n$, ktoré hovoria, kde je najbližší neskorší rok, v ktorom bolo aspoň tak teplo.

Tieto hodnoty vieme jednoducho spočítať v čase $\Theta(n^2)$, jednoducho pre každé i v cykle vyskúšame všetky j. Neskôr ukážeme, ako hodnoty $prev_i$ šikovnejšie spočítať v čase $\Theta(n)$.

Majme teraz výrok „v roku q_1 bolo najväčšie teplo od roku q_2“. Ako zistiť jeho pravdivostnú hodnotu?

V prvom rade si zistíme, či sú roky q_1 a q_2 medzi zadanými n rokmi, a ak áno, nájdeme indexy a a b také, že $q_1 = r_a$ a $q_2 = r_b$. Keďže roky na vstupe boli utriedené, vieme toto spraviť v čase $\Theta(\log n)$, napríklad pomocou binárneho vyhľadávania. (V našej implementácii namiesto toho používame `map` z STL.)

Na základe toho, či sa nám podarilo roky q_1 a q_2 nájsť medzi údajmi zo vstupu, vieme pravdivosť daného výroku rozhodnúť nasledovne:

1. **Ani** a **ani** b **neexistuje.**
   Výrok je vždy otázny.

2. **Aj** a **aj** b **existuje.**
   Ak $prev_a = b$, výrok je pravdivý alebo otázny (podľa toho, či poznáme všetky teploty medzi danými rokmi). Inak je výrok nepravdivý.

3. a **existuje,** b **nie.**
   Nech $prev_a = x$. Ak $r_x > q_2$, tak je výrok nepravdivý, inak je otázny.

4. b **existuje,** a **nie.**
   Nech $next_b = x$. Ak $r_x < q_1$, tak je výrok nepravdivý, inak je otázny.

Zdôvodníme teraz jednotlivé prípady.

- V prípade 1 výrok zjavne nie je pravdivý (lebo nepoznáme niektoré teploty). Je však vždy otázny – aby bol pravdivý, stačí doplniť v rokoch q_1 a q_2 tú istú teplotu, vyššiu od všetkých teplôt medzi nimi.

- V prípade 2 vieme teplotu aj v roku q_1, aj v roku q_2. Aby výrok mohol byť pravdivý, musí nutne byť teplota v q_2 aspoň taká ako v q_1, a navyše q_2 musí byť najbližší takýto rok. Inými slovami, musí platiť $prev_a = b$. Ak teda neplatí $prev_a = b$, výrok je nepravdivý.

  Ak platí $prev_a = b$, výrok je buď pravdivý (ak poznáme všetky teploty medzi q_1 a q_2), alebo otázny (inak).

  To, či poznáme všetky teploty v danom úseku, ľahko zistíme v konštantnom čase. Stačí porovnať hodnoty $q_1 - q_2$ a $a - b$. Ak si nie sú rovné (a teda $q_1 - q_2 > a - b$), niektoré roky nám chýbajú.

- Prípad 3 vyzerá nasledovne: Nech $prev_a = x$. Inými slovami, rok r_x je najbližší rok pred q_1, o ktorom vieme, že v ňom bolo teplejšie. Ak $r_x > q_2$, leží tento rok medzi rokmi q_1 a q_2, a teda daný výrok je nutne nepravdivý. Naopak, ak $r_x < q_2$, znamená to, že o žiadnom takom roku medzi q_1 a q_2 nevieme, a ľahko doplníme chýbajúce teploty tak, aby výrok pravdivý bol.

- Prípad 4 funguje analogicky ako prípad 3.

Predpočítanie:

Zostáva vysvetliť, ako v lineárnom čase spočítať hodnoty $prev$ a $next$. Vysvetlíme si spočítanie hodnôt $prev$, pre $next$ to vyzerá analogicky.

Budeme postupne spracúvať hodnoty zo vstupu. V každom okamihu si budeme pamätať množinu tých rokov, ktoré ešte môžu byť „najbližším predchodcom“ nejakého roku.

Kľúčové pozorovanie je nasledovné: Akonáhle spracujeme rok r, v ktorom bola teplota t, môžeme zabudnúť na všetky predchádzajúce roky, ktoré mali teplotu nižšiu ako t.

(Napríklad keby boli na vstupe postupne roky s teplotami: 100, 14, 74, 39, 40, 27 a 12, stačilo by si pamätať roky s teplotami 100, 74, 40, 27 a 12.

Keby teraz prišiel ďalší rok s teplotou 47, mohli by sme po jeho spracovaní zabudnúť na ďalšie tri roky, a pamätať si len roky s teplotami 100, 74 a 47.)

Spracovanie jedného roku r teda bude vyzerať nasledovne: Zabudneme na všetky roky, ktoré majú nižšiu teplotu ako práve spracúvaný. Následne zoberieme posledný rok p s vyššou alebo rovnou teplotou ako r, a zapíšeme si $prev_r = p$. Na záver pridáme rok r medzi pamätané roky.

Keďže roky, ktoré si v ľubovoľnom okamihu pamätáme, sú utriedené podľa teploty, môžeme na implementáciu vyššie popísaného postupu jednoducho použiť zásobník.

Odhad časovej zložitosti potom spravíme nasledovne: Každý rok raz spracujeme a vložíme na zásobník, a každý rok najviac raz zo zásobníka vyhodíme. Preto dokopy spravíme pri predpočítaní $O(N)$ operácií.

Záver:

Ukázali sme riešenie, ktoré si v čase $O(N)$ predpočíta hodnoty, ktoré následne používa na to, aby ľubovoľnú otázku zodpovedalo v čase $O(\log N)$. Celková časová zložitosť nášho riešenia je teda $O(N + M \log N)$. Pamäťová zložitosť je $O(N)$.

Iné riešenia:

Existuje viacero iných prístupov, ktorými sa dá dosiahnuť rovnaká časová zložitosť ako vo vzorovom riešení. Môžeme si napríklad nad vstupnými dátami postaviť dva intervalové stromy. Pomocou prvého budeme vedieť v ľubovoľnom intervale rokov v čase $O(\log N)$ nájsť maximálnu teplotu, pomocou druhého budeme o ľubovoľnom intervale vedieť povedať, či máme zadané teploty pre všetky roky v ňom. Detaily implementácie maximového intervalového stromu nájdete napríklad vo vzorových riešeniach domáceho kola.

Listing programu:

```
#include <iostream>
#include <vector>
#include <map>
#include <stack>
using namespace std;

#define MAXN 101000
#define NEKONECNO 1987654321

int N, M;
int R[MAXN], T[MAXN];
map<int,int> kde;
```

```
int prev[MAXN], next[MAXN];

int main() {
  // nacitame vstup
  cin >> N;
  for (int i=1; i<=N; i++) cin >> R[i] >> T[i];
  vector< pair<int,int> > napln_kde(N);
  for (int i=1; i<=N; i++) napln_kde[i-1] = make_pair(R[i],i);
  kde.insert( napln_kde.begin(), napln_kde.end() );
  R[0] = -NEKONECNO;
  R[N+1] = NEKONECNO;

  // spocitame polia prev[] a next[]
  stack<int> teplo;
  stack<int> cislo;
  teplo.push(NEKONECNO); cislo.push(0);
  for (int i=1; i<=N; i++) {
    while (teplo.top() < T[i]) { cislo.pop(); teplo.pop(); }
    prev[i] = cislo.top();
    teplo.push(T[i]); cislo.push(i);
  }
  while (!teplo.empty()) { teplo.pop(); cislo.pop(); }
  teplo.push(NEKONECNO); cislo.push(N+1);
  for (int i=N; i>=1; i--) {
    while (teplo.top() < T[i]) { cislo.pop(); teplo.pop(); }
    next[i] = cislo.top();
    teplo.push(T[i]); cislo.push(i);
  }

  // odpovedame na otazky
  cin >> M;
  while (M--) {
    int q1, q2;
    cin >> q1 >> q2;
    int a = kde[q1], b = kde[q2];
    if (a==0 && b==0) { cout << "otazny" << endl; continue; }
    if (a!=0 && b!=0 && b!=prev[a]) { cout << "nepravdivy" << endl; continue; }
    if (a!=0 && b!=0 && a-b==q1-q2) { cout << "pravdivy" << endl; continue; }
    if (a!=0 && b!=0) { cout << "otazny" << endl; continue; }
    if (a!=0 && R[prev[a]]>q2) { cout << "nepravdivy" << endl; continue; }
    if (a!=0) { cout << "otazny" << endl; continue; }
    if (R[next[b]]<q1) { cout << "nepravdivy" << endl; continue; }
    { cout << "otazny" << endl; continue; }
  }
  return 0;
}
```

A-III-2 Tobogany

V riešení budeme používať grafovú terminológiu, bazény budeme volať vrcholmi a tobogany hranami.

Definujme *výšku* vrcholu nasledovne: Cieľový vrchol (spodný bazén) má výšku 0. Pre každý iný vrchol výšku spočítame ako jedna plus maximum výšok vrcholov, do ktorých sa z neho vieme dostať toboganom.

Množinu vrcholov, ktoré majú rovnakú výšku, budeme volať *vrstva*.

Jazdou z vrcholu v nazveme postupnosť, v ktorej sa striedajú typy bazénov, ktoré stretáme a čísla toboganov, ktoré si vyberáme. Napríklad pre druhý príklad v zadaní jedna možná jazda z vrcholu 2 vyzerá nasledovne: 6,1,4,3,6,3. (Začneme v šesťuholníku, toboganom 1 prejdeme do štvorca, odtiaľ toboganom 3 do šesťuholníka, a odtiaľ toboganom 3 do cieľa.)

Pozorovanie 1. Výška vrcholu zodpovedá počtu toboganov v najdlhšej jazde z neho.

Pozorovanie 2. Ak majú dva vrcholy rovnakú množinu jázd, tak ležia v tej istej vrstve.

Pravdivosť pozorovania 2 vyplýva jednoducho z toho, že množinou jázd je jednoznačne určená výška vrcholu.

Dva vrcholy, ktoré majú rovnakú množinu jázd, budeme volať *ekvivalentné*.

Pozorovanie 3. V optimálnom grafe žiadne dva vrcholy nebudú ekvivalentné.

Pravdivosť tohoto tvrdenia by mala byť intuitívne jasná. Formálne ho môžeme dokázať napríklad nasledovne. Sporom, nech tvrdenie neplatí. Nájdime teda v optimálnom grafe najnižšiu vrstvu, v ktorej sú dva ekvivalentné vrcholy. Potom ale môžeme zostrojiť nový graf, v ktorom jeden z týchto dvoch vrcholov zahodíme, a všetky tobogany, ktoré šli doň, presmerujeme do toho druhého. To je ale spor s tým, že pôvodný graf už bol optimálny.

Pozorovanie 4. Optimálny graf má toľko vrcholov, koľko rôznych množín jázd majú vrcholy pôvodného grafu – čiže koľko rôznych navzájom neekvivalentných vrcholov pôvodný graf obsahuje.

Všimnime si ľubovoľný vrchol v pôvodného grafu. V pôvodnom grafe existuje cesta, ktorou sa doň z horného vrcholu dostaneme. Táto istá cesta musí v optimálnom grafe tiež viesť do nejakého vrcholu v', a keďže má optimálny graf byť nerozlíšiteľný od pôvodného, musí mať v' rovnakú množinu jázd ako v. Preto má optimálny graf aspoň toľko vrcholov, koľko rôznych množín jázd majú vrcholy pôvodného grafu. Opačná nerovnosť vyplýva z pozorovania 3.

Pozorovanie 5. Optimálny graf vieme vyrobiť z pôvodného tak, že postupne oddola nahor pospájame v každej vrstve všetky skupiny navzájom ekvivalentných vrcholov.

Ako vieme efektívne zistiť, či sú nejaké dva vrcholy ekvivalentné? Najjednoduchšia (a najmenej efektívna) metóda je samozrejme vyskúšať všetky možné jazdy z jedného a skontrolovať, že idú spraviť aj z druhého a naopak. Našťastie pre nás, ak budeme postupovať pekne systematicky po vrstvách, vieme si skoro všetku túto prácu ušetriť.

Pozorovanie 6. Nech u a v sú dva vrcholy v tej istej vrstve, a nech už v nižších vrstvách žiadne dva vrcholy nie sú ekvivalentné. Potom u a v sú ekvivalentné vtedy a len vtedy, ak majú rovnaký tvar a pre každé i platí, že i-tou hranou sa z u dostanem do toho istého vrcholu ako z v.

Pozorovanie 6 nám dáva veľmi efektívny návod, ako zistiť, či je práve spracúvaná dvojica vrcholov ekvivalentná.

Naše riešenie sa bude skladať z dvoch krokov. V prvom prehľadávaním do hĺbky zistíme výšky jednotlivých vrcholov, a roztriedime ich do vrstiev. Následne v druhom kroku budeme vrcholy postupne po vrstvách spracúvať.

Popíšeme si teraz podrobnejšie, ako bude spracovanie jednej vrstvy prebiehať. Najskôr utriedime vrcholy podľa štvoríc (tvar,ciele všetkých troch vychádzajúcich hrán). Takto sa dostanú ekvivalentné vrcholy k sebe. Teraz z každej skupiny ekvivalentných vrcholov necháme len jeden, a presmerujeme do neho hrany, ktoré doteraz viedli do ostatných vrcholov.

Odstránenie vrcholu vieme ľahko spraviť v čase lineárnom od počtu vchádzajúcich hrán. (Potrebujeme na to pamätať si pre každý vrchol zoznam hrán, ktoré doň vchádzajú.) Keďže každý vrchol odstránime najviac raz, vieme dokopy všetko odstraňovanie spraviť v čase $O(N)$.

Ak by sme na triedenie vrcholov použili všeobecný triediaci algoritmus, dostali by sme tak riešenie s časovou zložitosťou $O(N \log N)$.

V našom prípade však vieme vrcholy utriediť v lineárnom čase, pomocou triedenia CountSort. (Tvary majú len 3 rôzne hodnoty. Vrcholov, do ktorých smerujú hrany, je len lineárne veľa, a v čase lineárnom od ich počtu si ich vieme prečíslovať číslami od 1 po nanajvýš ich počet.)

Takto dostávame riešenie s časovou zložitosťou $O(N)$.

Listing programu:

```
#include <algorithm>
#include <iostream>
```

```
#include <vector>
using namespace std;

#define SIZE(t) ((int)((t).size()))
#define MAXN 100047

int N;
int tvar[MAXN], tobogany[MAXN][3];
int hlbka[MAXN];

int pocitaj_hlbku(int v) {
  int &res = hlbka[v];
  if (res >= 0) return res;
  if (v==N-1) return res=0;
  for (int i=0; i<3; i++) res = max(res,1+pocitaj_hlbku(tobogany[v][i]));
  return res;
}

vector<int> count_sort(vector<int> prvky, vector<int> hodnoty) {
  int najvacsia = *max_element( hodnoty.begin(), hodnoty.end() );

  vector<int> pocty(najvacsia+1), offset(najvacsia+1);
  for (int i=0; i<SIZE(hodnoty); i++) pocty[ hodnoty[i] ]++;
  for (int i=1; i<=najvacsia; i++) offset[i] = offset[i-1] + pocty[i-1];

  vector<int> vystup( SIZE(prvky) );
  for (int i=0; i<SIZE(prvky); i++) vystup[ offset[ hodnoty[i] ]++ ] = prvky[i];
  return vystup;
}

bool rovnake(int x, int y) {
  if (tvar[x] != tvar[y]) return false;
  for (int q=0; q<3; q++) if (tobogany[x][q] != tobogany[y][q]) return false;
  return true;
}

int main() {
  // nacitame vstup
  cin >> N;
  for (int i=0; i<N-1; i++)
    scanf("%d %d %d %d ",
          &tvar[i], &tobogany[i][0], &tobogany[i][1], &tobogany[i][2]);
  for (int i=0; i<N-1; i++) for (int q=0; q<3; q++) tobogany[i][q]--;

  // spocitame hlbky vrcholov a roztriedime vrcholy do vrstiev
  memset(hlbka,-1,sizeof(hlbka));
  for (int i=0; i<N; i++) hlbka[i] = pocitaj_hlbku(i);
  int H = hlbka[0]+1;
  vector< vector<int> > vrstvy(H);
  for (int i=0; i<N; i++) vrstvy[hlbka[i]].push_back(i);
```

```
  // pre kazdy vrchol si najdeme vsetky, ktore nan ukazuju
  vector< vector<int> > opacne(N);
  for (int i=0; i<N-1; i++)
    for (int j=0; j<3; j++)
      opacne[ tobogany[i][j] ].push_back(i);

  vector<int> cislo(N);

  // postupne po vrstvach hladame a odstranujeme ekvivalentne vrcholy
  for (int i=0; i<H; i++) {
    // utriedime vrcholy vo vrstve
    vector<int> hodnoty( SIZE(vrstvy[i]) );
    for (int j=0; j<SIZE(vrstvy[i]); j++) hodnoty[j] = tvar[ vrstvy[i][j] ];
    vrstvy[i] = count_sort( vrstvy[i], hodnoty );
    for (int q=0; q<3; q++) {
      // precislujeme vrcholy
      int last = 0;
      for (int j=0; j<SIZE(vrstvy[i]); j++)
        if (!cislo[ tobogany[ vrstvy[i][j] ][q] ])
          cislo[ tobogany[ vrstvy[i][j] ][q] ] = ++last;
      // vyrobime pomocne pole
      for (int j=0; j<SIZE(vrstvy[i]); j++)
        hodnoty[j] = cislo[ tobogany[ vrstvy[i][j] ][q] ];
      // utriedime podla neho
      vrstvy[i] = count_sort( vrstvy[i], hodnoty );
      // upraceme po sebe cisla
      for (int j=0; j<SIZE(vrstvy[i]); j++)
        cislo[ tobogany[ vrstvy[i][j] ][q] ] = 0;
    }
    // najdeme a odstranime duplikaty
    vector<int> ostava(1,vrstvy[i][0]);
    for (int j=1; j<SIZE(vrstvy[i]); j++) {
      if (rovnake(vrstvy[i][j],vrstvy[i][j-1])) {
        int stare = vrstvy[i][j], nove = ostava.back();
        for (int k=0; k<SIZE( opacne[stare] ); k++)
          for (int q=0; q<3; q++)
            if (tobogany[ opacne[stare][k] ][q] == stare)
              tobogany[ opacne[stare][k] ][q] = nove;
      } else ostava.push_back( vrstvy[i][j] );
    }
    vrstvy[i] = ostava;
  }

  int res = 0;
  for (int i=0; i<H; i++) res += SIZE( vrstvy[i] );
  cout << res << endl;
}
```

A-III-3 Prekladacie stroje

Podúloha a):

Všetky stroje, ktoré v tejto úlohe zostrojíme, budú samozrejme fungovať tak, že kopírujú vstupný reťazec na výstup a popri tom kontrolujú, či má požadovaný tvar.

Najjednoduchšie riešenie je stroj s 105 stavmi, ktorý si bude v stave pamätať zvyšok, ktorý dáva doteraz prečítaný reťazec po delení 105.

$$\begin{aligned} A_1 &= (K_1, \Sigma, P_1, 0, F_1) \\ \Sigma &= \{0, \ldots, 9\} \\ K_1 &= \{0, \ldots, 104\} \\ F_1 &= K_1 - \{0\} \\ P_1 &= \{(x, y, y, (10x + y) \bmod 105) \mid \forall x \in K,\ \forall y \in \Sigma\} \end{aligned}$$

Existuje však ešte veľa priestoru na zlepšenie. V prvom rade si môžeme 105 zapísať ako 5×21. Aby sme zistili deliteľnosť 105, stačí zistiť deliteľnosť 21 a deliteľnosť 5. Lenže na kontrolu deliteľnosti 5 si nepotrebujeme priebežne počítať zvyšok. Stačí sa pozrieť na poslednú cifru, či je to 5 alebo 0.

Ukončovacie stavy takéhoto stroja budú teda zodpovedať situácii „doteraz prečítané číslo nie je deliteľné 21, alebo posledná cifra nebola 0 alebo 5“.

Takýto stroj vieme ľahko zostrojiť tak, že bude mať 42 stavov – budeme si v stave pamätať zvyšok po delení 21 a jeden bit hovoriaci, či bola posledná cifra 0 alebo 5.

Riešenie vieme ešte zlepšiť, ak si uvedomíme, že posledná cifra nás zaujíma len ak doteraz prečítané číslo je deliteľné 21. Takto upravený stroj už má len 22 stavov:

$$\begin{aligned} A_2 &= (K_2, \Sigma, P_2, 0_{zle}, F_2) \\ \Sigma &= \{0, \ldots, 9\} \\ K_2 &= \{0_{zle}, 0_{dobre}, 1, \ldots, 20\} \\ F_2 &= K_2 - \{0_{zle}\} \\ P_2 &= \{(x, y, y, z(x, y) \mid \forall x \in \{1, \ldots, 20\},\ \forall y \in \Sigma\} \end{aligned}$$

Nový stav $z(x, y)$ je jednoznačne určený pôvodným stavom x a prečítanou číslicou y nasledovne: Nech x' je číselná hodnota x. (Teda pre $x \in \{0_{zle}, 0_{dobre}\}$ je $x' = 0$, inak $x' = x$.) Hodnota x' bol zvyšok, ktorý dávala doteraz prečítaná

časť čísla po delení 21. Nový zvyšok po delení 21 je $z' = (10x' + y) \bmod 21$. Ak $z' \neq 0$, je nový stav $z(x, y) = z'$. V opačnom prípade nový stav závisí od y. Ak $y \in \{0, 5\}$, tak $z(x, y) = 0_{zle}$, inak $z(x, y) = 0_{dobre}$.

Aj toto riešenie však ešte môžeme výrazne zlepšiť.

Doteraz boli všetky prekladacie stroje, ktoré sme zostrojili, *deterministické*. (T. j. na každom vstupe bol preklad jediný, jednoznačne určený.)

Môžeme však ešte využiť to, že prekladací stroj môže mať na jednom slove prekladov viac, a výstupná množina je tvorená výstupmi všetkých platných prekladov.

Namiesto toho, aby sme na každom slove mali jeden výpočet „over, že nie je deliteľné 21“, budeme mať výpočty dva: „over, že nie je deliteľné 7“ a „over, že nie je deliteľné 3“.

Uvedieme konštrukciu prekladacieho stroja, ktorý si vystačí s 12 stavmi: začiatočný stav Z, sedem stavov na kontrolu deliteľnosti siedmimi $s_0, \ldots, s_6$, tri stavy na kontrolu deliteľnosti troma t_0, t_1, t_2 a ukončovací stav U, do ktorého sa budeme vedieť dostať len vtedy, ak prekladané slovo nie je deliteľné 105.

$$
\begin{aligned}
A_3 &= (K_3, \Sigma, P_3, Z, F_3) \\
\Sigma &= \{0, \ldots, 9\} \\
K_3 &= \{Z, U, s_0, \ldots, s_6, t_0, t_1, t_2\} \\
F_3 &= \{U\} \\
P_3 &= \{(Z, \varepsilon, \varepsilon, s_0), (Z, \varepsilon, \varepsilon, t_0)\} \\
&\cup \{(s_x, y, y, s_{(10x+y) \bmod 7}) \mid \forall x \in \{0, \ldots, 6\},\ \forall y \in \Sigma\} \\
&\cup \{(t_x, y, y, t_{(10x+y) \bmod 3}) \mid \forall x \in \{0, \ldots, 2\},\ \forall y \in \Sigma\} \\
&\cup \{(s_x, \varepsilon, \varepsilon, U) \mid \forall x \in \{1, 2, 3, 4, 5, 6\}\} \\
&\cup \{(t_x, \varepsilon, \varepsilon, U) \mid \forall x \in \{1, 2\}\} \\
&\cup \{(s_x, y, y, U) \mid \forall x \in \{0, \ldots, 6\},\ \forall y \in \{1, 2, 3, 4, 6, 7, 8, 9\}\} \\
&\cup \{(t_x, y, y, U) \mid \forall x \in \{0, \ldots, 2\},\ \forall y \in \{1, 2, 3, 4, 6, 7, 8, 9\}\}
\end{aligned}
$$

Do ukončovacieho stavu U môže takýto stroj prejsť len v nasledujúcich prípadoch:

- Ak sa rozhodol overovať, že číslo nie je deliteľné 7, a doteraz prečítané číslo deliteľné 7 nebolo.

- Ak sa rozhodol overovať, že číslo nie je deliteľné 3, a doteraz prečítané číslo deliteľné 3 nebolo.
- Ak sa rozhodol, že práve číta poslednú cifru, a tá nie je deliteľná 5.

Zjavne pre každé číslo, ktoré nie je deliteľné 105, vieme nájsť nejaký platný preklad. A naopak, ak máme platný preklad, musel do U prísť až po prečítaní celého vstupu, a teda vstup musel byť jedného z vyššie popísaných tvarov.

Na záver riešenie podľa Samuela Hapáka: Na ušetrenie ešte jedného stavu môžeme použiť to, že pomocou špeciálneho znaku \$ vieme zistiť, kedy nám skončil vstup.

Oproti predchádzajúcej konštrukcii zrušíme ukončovací stav U a namiesto neho prehlásime za ukončovacie stavy s_x a t_y pre všetky $x, y \neq 0$. Takto už máme zabezpečené, že vypíšeme čísla, ktoré nie sú deliteľné 3 a 7. Kvôli číslam nedeliteľným 5 pridáme pravidlá tvaru „ak si v nejakom stave t_x a čítaš reťazec $y\$$ (kde y je cifra iná od 0 a 5), vypíš y a prejdi do niektorého ukončovacieho stavu.

Dostávame teda prekladací stroj, ktorý rieši zadanú úlohu a má len 11 stavov.

Pomocné tvrdenie:

Ku každému prekladaciemu stroju A existuje prekladací stroj B, ktorý vyrába presne rovnaké preklady, ale v každom kroku prekladu vypíše najviac jedno písmeno.

Dôkaz tohto tvrdenia je triviálny. Budeme B postupne vyrábať z A, pričom len upravíme prekladové pravidlá, pri ktorých A zapisoval viac ako jedno písmeno.

Nech napríklad A obsahuje pravidlo (q, def, aba, p) ("ak si v stave q, a neprečítaná časť vstupu začína na def, môžeš prečítať def, zapísať aba a zmeniť stav na p„).

Toto pravidlo je zlé, lebo ak by sme ho v niektorom kroku použili, zapíšeme až tri písmená. Nahradíme ho teda v B tromi pravidlami, ktoré tieto tri písmená zapíšu postupne.

Presnejšie, do B pridáme dva nové stavy (nazvime ich tu trebárs ♡ a ♠), a namiesto pôvodného pravidla z A budeme v B mať pravidlá: $(q, def, a, \heartsuit)$, $(\heartsuit, \varepsilon, b, \spadesuit)$, $(\spadesuit, \varepsilon, a, p)$.

Ľahko nahliadneme, že akonáhle pri preklade v B použijeme prvé z trojice pravidiel, musíme v nasledujúcich dvoch krokoch použiť zvyšné dve pravidlá.

Uvedeným spôsobom upravíme postupne všetky pravidlá z A. Dostaneme tým B, ktorý má síce viac stavov ako A, ale vyrába presne tie isté preklady, a navyše v každom kroku prekladu vypíše najviac jedno písmeno. A to je presne to, čo sme chceli.

Podúloha b):

Tento prekladací stroj neexistuje. Intuitívne môžeme povedať, že je to preto, že máme len konečne veľa stavov, a teda si nevieme pomocou stavu pamätať, koľko a sme už zapísali.

Dokážeme to sporom, a pre jednoduchší dôkaz použijeme pomocné tvrdenie, ktoré sme si práve dokázali.

Predpokladajme teda, že hľadaný prekladací stroj existuje. Potom podľa pomocného tvrdenia existuje prekladací stroj, ktorý robí to isté, a navyše v každom kroku prekladu vypíše najviac jedno písmeno.

Zoberme si jeden takýto prekladací stroj A. Označme jeho počet stavov n.

Všimnime si teraz reťazce ε, ab, $aabb$, $\ldots$, až $\underbrace{a\ldots a}_{n}\underbrace{b\ldots b}_{n}$.

Toto je $n+1$ rôznych reťazcov, ktoré náš prekladací stroj musí vypísať na výstup. Ku každému z nich musí existovať v A aspoň jeden platný preklad. Vyberme si teda ku každému reťazcu jeden platný preklad.

Teraz využijeme, že náš stroj vypisuje písmená na výstup po jednom. Všimnime si v každom z našich $n+1$ prekladov okamih, kedy sme práve vypísali na výstup posledné a. Vypíšme si, v akom stave sa práve náš stroj nachádzal.

Takto dostaneme zoznam, v ktorom bude $n+1$ stavov. Náš stroj A však má len n rôznych stavov, a teda je v našom zozname nejaký stav aspoň dvakrát.

Čo toto znamená? Znamená to, že existujú dva rôzne platné preklady, ktoré vyzerajú nasledovne:

Prvý: Začneme v začiatočnom stave. Vypíšeme x písmen a. V tomto okamihu sme v nejakom stave p. Vypíšeme x písmen b, a tým sa dostaneme do ukončovacieho stavu.

Druhý: Začneme v začiatočnom stave. Vypíšeme y písmen a (kde $y \neq x$). V tomto okamihu sme v tom istom stave p ako prvý preklad. Vypíšeme y písmen b, a tým sa dostaneme do ukončovacieho stavu.

Aby sme dostali spor, stačí si už len uvedomiť, že môžeme oba preklady "rozstrihnúť„ a následne "zlepiť„ prvú polovicu prvého a druhú polovicu druhého prekladu.

Inými slovami, aj toto musí byť platný preklad:

Tretí: Začneme v začiatočnom stave. Vypíšeme x písmen a. V tomto okamihu sme v stave p. Vypíšeme y písmen b, a tým sa dostaneme do ukončovacieho stavu.

Vo výslednej množine $A(X)$ sa teda musí nachádzať aj slovo $\underbrace{a\ldots a}_{x}\underbrace{b\ldots b}_{y}$. Takéto slovo tam ale byť nesmie, a to je práve hľadaný spor.

A-III-4 O lenivých prasiatkach

Začneme tým, že si uvedomíme, že stromy ani senníky sa nehýbu. V každom okamihu teda vieme situáciu jednoznačne popísať tak, že udáme súradnice všetkých prasiatok.

Všetkých možných situácií nie je až tak veľa, ako by sa na prvý pohľad mohlo zdať. Máme $2P$ súradníc, každá je z rozsahu od 1 po N, teda možných popisov je rádovo N^{2P}. (V skutočnosti menej, lebo prípustné sú len tie popisy, kde je všetkých P políčok navzájom rôznych.) Pre intuíciu, pre $P = 4$ a $N = 8$ je N^{2P} len 16 miliónov.

Prvé, pomerne priamočiare riešenie bude používať prehľadávanie do šírky. Budeme postupne zostrojovať situácie, ktoré sú dosiahnuteľné na jeden krok, na dva kroky, a tak ďalej.

Samozrejme, aby sme neprezerali tú istú situáciu viackrát, potrebujeme si napríklad v poli pamätať, ktoré situácie sme už videli. Na to môžeme buď použiť $2P$-rozmerné pole, alebo si napísať pomocné funkcie na kódovanie celého popisu situácie do jedného čísla a použiť pole 1-rozmerné. (Druhý prístup je o čosi lepší, lebo sa nepotrebujeme trápiť s rozmermi poľa, spravíme ho proste také veľké, ako nám pamäťový limit dovolí.)

Prvé zlepšenie:

Takýto prístup však ešte na plný počet bodov nestačí. Môžeme však našu situáciu ľahko o niečo vylepšiť. Stačí si napríklad uvedomiť, že nám nezáleží na tom, ktoré prasiatko je ktoré. A teda si ich súradnice môžeme udržiavať utriedené. Takto sa nám vždy $P!$ situácii zmení na jednu. Pre intuíciu, pre $P = 4$ a $N = 8$ je $N^{2P}/P!$ už len 700 tisíc.

Prehľadávanie z oboch strán naraz:

Napriek tomu nás problém s priveľa stavmi začne pre veľké hodnoty N trápiť znova.

Problém je v tom, že pri každom ďalšom ťahu môže počet dosiahnuteľných situácií narásť až exponenciálne: v prvom ťahu máme rádovo $4P$ možností, po druhom ich už môže byť až rádovo $(4P)^2$, a tak ďalej. A väčšina týchto situácií je nám úplne nanič, ale to naše riešenie (zatiaľ) nevie rozpoznať.

Zaujímavým a jednoduchým trikom, ako zmenšiť množstvo spracúvaných "zbytočných„ situácií je začať riešenie hľadať naraz z oboch strán – aj zo začiatočnej situácie, aj z cieľovej.

Ak teda má napríklad optimálne riešenie 31 krokov, nevygenerujeme všetky situácie, ktoré idú dosiahnuť zo začiatočnej na 31 krokov, ale len: (tie, ktoré idú dosiahnuť na 16 krokov) + (tie, z ktorých ide na 15 krokov dosiahnuť cieľ).

Algoritmus A*:

Ešte lepšie by bolo, keby náš program vedel robiť to, čo my "od oka„ spravíme ľahko: pozrieť sa na nejakú situáciu a zhodnotiť, že ju nemá zmysel skúšať, lebo vyzerá príliš nanič.

Ako môže takéto niečo vedieť program robiť? Použijeme na to *heuristickú funkciu* h, ktorá bude hovoriť nejaký dolný odhad riešenia. Inými slovami, pre h musí platiť: Ak S je ľubovoľná situácia, tak najlepšie riešenie pre S má aspoň $h(S)$ krokov.

Načo nám je takáto funkcia h dobrá? Pomocou nej môžeme vedieť niektoré situácie rovno zahodiť. Predstavme si napríklad, že už poznáme nejaké riešenie našej úlohy s hodnotou X, a práve sme zistili, že sa zo začiatočnej situácie vieme na $d(S)$ krokov dostať do situácie S. Spočítame si $h(S)$, a ak zistíme, že $d(S) + h(S) \geq X$, tak je pre nás S nezaujímavá – určite nám nepomôže nájsť lepšie riešenie.

Rôznych funkcií, ktoré spĺňajú našu podmienku, je samozrejme veľa. Spĺňa ju napríklad aj funkcia $h(S) = 0$. Tá nám ale príliš nepomôže. Trik je samozrejme v tom, že treba zvoliť funkciu h, ktorá na jednej strane hodnotu riešenia odhadne čo najlepšie, ale na druhej strane ju musíme vedieť efektívne počítať.

Samotný algoritmus A* je v podstate len upravené prehľadávanie do šírky. Zmena bude jednoduchá: Nebudeme používať obyčajnú frontu (z ktorej ako prvú vyberieme situáciu, ktorá tam bola prvá vložená). Namiesto toho budeme používať prioritnú frontu. Vyberať na spracovanie budeme vždy tú situáciu S, ktorá má najmenší súčet $d(S) + h(S)$. Prestaneme, akonáhle sa prvýkrát

dostaneme do cieľa.

Pre ľubovoľnú funkciu h spĺňajúcu našu podmienku, bude algoritmus A* fungovať a nájde najkratšiu cestu do cieľa. Totiž keď sa dostaneme do cieľovej situácie C, pre ľubovoľnú ešte nespracovanú situáciu S platí, že $d(S) + h(S) \geq d(C) + h(C) = d(C) + 0 = d(C)$, a teda cez S už lepšiu cestu od práve objavenej určite nevyrobíme.

Pritom platí, že čím lepšiu funkciu h budeme mať, tým menej situácií budeme musieť zobrať do úvahy. Vhodnou voľbou h však vieme dosiahnuť to, že ako prvé pôjde náš algoritmus prezerať tie situácie, ktoré vyzerajú najviac nádejne. Presný dôkaz je komplikovaný, ukážeme si ale aspoň oba okrajové prípady:

Keby sme zobrali $h(S) = 0$, dostali by sme pôvodné prehľadávanie do šírky, kde by sme spracúvali situácie usporiadané podľa vzdialenosti od začiatku.

Naopak, čo by sa stalo, ak by funkcia h bola presná, teda vracala počet krokov potrebný na dosiahnutie cieľa? V takomto prípade by algoritmus A* jednoducho prešiel po najkratšej ceste a skončil. Žiadne odchýlky od nej by neskúšal, keďže pre také situácie by súčet $d(S) + h(S)$ bol väčší.

Heuristická funkcia:

V našom prípade môžeme použiť napríklad takúto funkciu h:

Vyskúšame všetkých $P!$ spôsobov, ako priradiť prasiatka senníkom. Pre každú možnosť zoberieme pre každé prasiatko p najkratšiu vzdialenosť d_p od jeho aktuálnej polohy k jeho senníku. Keďže prasiatko vie robiť na jeden krok posun najviac o dve políčka, určite bude potrebovať aspoň $\lceil d_p/2 \rceil$ krokov. Toto sčítame pre všetky prasiatka. Spomedzi všetkých $P!$ spôsobov vyberieme ten, kde nám výsledok vyjde najmenší, a ten vrátime ako hodnotu h.

Najkratšie vzdialenosti medzi každou dvojicou políčok si vieme na začiatku predpočítať (napr. prehľadávaniami do šírky v celkovom čase $O(N^4)$), a teda jedno volanie funkcie h má časovú zložitosť $O(P \cdot P!)$.

Ešte o niečo lepšie je pri počítaní vzdialeností brať do úvahy to, či sa tam vôbec dá skákať. Teda predpočítame si priamo najmenšie počty krokov, ktoré by prasiatko na presun potrebovalo, keby vždy, keď chce skákať, malo na správnom mieste kamaráta.

Listing programu:

```
#include <algorithm>
#include <cstdio>
#include <queue>
#include <map>
```

```
using namespace std;

#define SIZE(t) ((int)((t).size()))

#define MAX_P 4
#define MAX_N 22

int N,P; // rozmer mapy a pocet prasiatok
char M[MAX_N+4][MAX_N+4]; // mapa
int dr[] = {-1,1,0,0}, dc[] = {0,0,-1,1}; // smery pohybu
int G[600][600]; // prasacie vzdialenosti medzi policka mapy

inline int encode(int r, int c) { return (N+3)*r + c; }
inline void decode(int co, int &r, int &c) { c=co%(N+3); r=co/(N+3); }

// spravime si class na pamatanie si situacie
class State {
  public:
  int cells[MAX_P];
  // situacie vieme normalizovat (zoradit prasiatka):
  void normalize() {
    for (int i=0; i<P; i++) for (int j=i+1; j<P; j++)
      if (cells[i]>cells[j]) swap(cells[i],cells[j]);
  }
  // konstruktor z pola suradnic
  State(int *coords) {
    for (int i=0; i<P; i++) cells[i] = encode(coords[2*i],coords[2*i+1]);
    normalize();
  }
  // situacie vieme lexikograficky zoradit
  bool operator<(const State &rhs) const {
    for (int i=0; i<P; i++) {
      if (cells[i] < rhs.cells[i]) return true;
      if (cells[i] > rhs.cells[i]) return false;
    }
    return false;
  }
};

typedef pair<int,State> Record;

// funkcia ktora vygeneruje mozne tahy zo situacie "kde"
vector<State> generuj_tahy(State kde) {
  vector<State> res;
  int x[2*MAX_P];
  for (int i=0; i<P; i++) decode(kde.cells[i],x[2*i],x[2*i+1]);
  for (int i=0; i<P; i++) M[x[2*i]][x[2*i+1]]='O';

  for (int i=0; i<P; i++) for (int d=0; d<4; d++) {
    x[2*i]+=dr[d]; x[2*i+1]+=dc[d];
```

```
    bool two = false;
    if (M[x[2*i]][x[2*i+1]]=='O') { two=true; x[2*i]+=dr[d]; x[2*i+1]+=dc[d]; }

    if (M[x[2*i]][x[2*i+1]]=='.') res.push_back(State(x));

    if (two) { x[2*i]-=dr[d]; x[2*i+1]-=dc[d]; }
    x[2*i]-=dr[d]; x[2*i+1]-=dc[d];
  }

  for (int i=0; i<P; i++) M[x[2*i]][x[2*i+1]]='.';
  return res;
}

// eval() vracia hodnotu -(d(kde)+h(kde))
int eval(int dist, const State &kde, const State &koniec) {
  int x[2*MAX_P], y[2*MAX_P], perm[MAX_P];
  for (int i=0; i<P; i++) decode(koniec.cells[i],x[2*i],x[2*i+1]);
  for (int i=0; i<P; i++) decode(kde.cells[i],y[2*i],y[2*i+1]);
  for (int i=0; i<P; i++) perm[i]=i;
  int bestdist = 987654321;
  do {
    int thisdist = 0;
    for (int i=0; i<P; i++) {
      int a = encode(x[2*i],x[2*i+1]);
      int b = encode(y[2*perm[i]],y[2*perm[i]+1]);
      thisdist += G[a][b];
    }
    bestdist = min(bestdist,thisdist);
  } while (next_permutation(perm,perm+P));
  dist += bestdist;
  return -dist;
}

int main() {
  // nacitame vstup
  scanf("%d ",&N);
  for (int i=0; i<=N+1; i++) for (int j=0; j<=N+1; j++) M[i][j]='#';
  for (int i=1; i<=N; i++) scanf("%s",M[i]+1);
  scanf("%d ",&P);
  int x[2*MAX_P];
  for (int i=0; i<2*P; i++) scanf("%d ",&x[i]);
  State zaciatok(x);
  for (int i=0; i<2*P; i++) scanf("%d ",&x[i]);
  State koniec(x);

  // spocitame prasiatkove vzdialenosti policok na mape
  memset(G,47,sizeof(G));
  for (int i=1; i<=N; i++) for (int j=1; j<=N; j++) {
    if (M[i][j]!='.') continue;
    int x = encode(i,j);
```

```
    G[x][x]=0;
    for (int d=0; d<4; d++) {
      int k,l;
      k=i+dr[d], l=j+dc[d];
      if (M[k][l]!='.') continue;
      G[x][encode(k,l)]=1;

      k=i+2*dr[d], l=j+2*dc[d];
      if (M[k][l]!='.') continue;
      G[x][encode(k,l)]=1;
    }
  }
  int TOP = (N+3)*(N+3);
  for (int k=0; k<TOP; k++)
  for (int i=0; i<TOP; i++)
  for (int j=0; j<TOP; j++) G[i][j] = min(G[i][j], G[i][k]+G[k][j]);

  // spustime algoritmus A*
  map<State,int> dist;
  dist[zaciatok] = 0;

  priority_queue<Record> Q;
  Q.push( Record( eval(0,zaciatok,koniec), zaciatok ) );

  while (!dist.count(koniec)) {
    State kde = Q.top().second; Q.pop();
    vector<State> kam = generuj_tahy(kde);
    for (int i=0; i<SIZE(kam); i++) {
      bool ok = false;
      if (!dist.count(kam[i])) ok = true;
      if (dist.count(kam[i])) if (dist[kam[i]] > dist[kde]+1) ok = true;
      if (!ok) continue;
      dist[kam[i]] = dist[kde] + 1;
      Q.push( Record( eval(dist[kam[i]],kam[i],koniec), kam[i] ) );
    }
  }

  printf("%d\n",dist[koniec]);
  return 0;
}
```

A-III-5 Posledná dobrota

Začneme terminológiou. Stav hry, teda aktuálne počty koláčov a ovocia, budeme volať *pozícia*. *Vyhrávajúca stratégia* je postup, ktorý nám zaručí, že hru vyhráme, bez ohľadu na to, ako bude ťahať protihráč. Pozícia je *vyhrávajúca*, ak hráč, ktorý je práve na ťahu, má vyhrávajúcu stratégiu. Ostatné pozície voláme *prehrávajúce*.

Prezeranie stromu hry:

Najjednoduchšie riešenie, ktoré sa dá použiť pre ľubovoľnú konečnú matematickú hru, je založené na dvoch jednoduchých myšlienkach:

- Ak z danej pozície všetky ťahy vedú do vyhrávajúcich pozícií, tak je táto pozícia prehrávajúca.
- Ak z danej pozície existuje ťah do prehrávajúcej, tak je táto pozícia vyhrávajúca.

(Ak všetky ťahy vedú do vyhrávajúcich pozícií, nech si vyberieme ktorýkoľvek, vždy tým dostaneme súpera do vyhrávajúcej pozície. A ak sa potom bude súper držať nejakej vyhrávajúcej stratégie, hru prehráme. Preto takáto pozícia je prehrávajúca. Naopak, ak existuje ťah do prehrávajúcej pozície, spravíme ho, a tým dostaneme súpera do tejto, pre neho prehrávajúcej pozície.)

Túto myšlienku ľahko prepíšeme do rekurzívnej funkcie, ktorá nám o pozícii povie, či je vyhrávajúca alebo prehrávajúca.

Dynamické programovanie / memoizácia:

Problém predchádzajúceho prístupu spočíva v tom, že je príliš pomalý. Hlavný dôvod je ten, že pri rekurzívnych volaniach vlastne skúša všetky možné priebehy hry, a pri tom mnohé pozície vyhodnotí veľa krát.

Tu je samozrejme ľahká pomoc – akonáhle o nejakej pozícii zistíme, či je vyhrávajúca, zapíšeme si to do pomocného poľa. Takto dosiahneme to, že každú pozíciu budeme spracúvať práve raz.

Rôznych pozícií je $O(AB)$. Preto je taká aj pamäťová zložitosť tohoto riešenia. Na vyhodnotenie pozície potrebujeme prezrieť všetky možné ťahy, ktorých je $O(K)$. Preto je časová zložitosť riešenia $O(ABK)$.

Na toto riešenie sa môžeme dívať aj z opačnej strany: Keby sme napríklad pozície spracúvali zoradené podľa celkového množstva dobrôt, tak by platilo, že v okamihu, keď vyhodnocujeme nejakú pozíciu, už vieme o všetkých pozíciách, do ktorých môžeme ťahať, či sú vyhrávajúce alebo prehrávajúce.

Šikovnejšie dynamické programovanie:

Predchádzajúce riešenie ešte stále robí zbytočne veľa práce, keď skúša všetky možné ťahy. Namiesto toho môžeme využiť skutočnosť, že prehrávajúcich pozícií je málo.

Budeme postupne prechádzať pozície. Vždy, keď narazíme na prehrávajúcu pozíciu (x, y), zaznačíme si o všetkých pozíciách $(x+i, y)$, $(x, y+i)$ aj $(x+i, y+i)$ pre $1 \leq i \leq K$, že sú vyhrávajúce:

```
for (int x=0; x<=A; x++)
  for (int y=0; y<=B; y++)
    if (!vyhravajuca[x][y])
      for (int i=1; i<=K; i++)
        vyhravajuca[x+i][y] = vyhravajuca[x][y+i] = vyhravajuca[x+i][y+i] = true;
```

Zjavne každú pozíciu zmeníme z prehrávajúcej na vyhrávajúcu najviac raz. Preto má tento algoritmus časovú zložitosť $O(AB)$.

Iná formulácia hry:

V ďalšom texte riešenia bude lepšie predstaviť si našu hru ináč. Predstavme si, že máme obrovskú šachovnicu, ktorá pokrýva celý prvý kvadrant súradnicovej sústavy. Políčko v ľavom dolnom rohu nech má súradnice $(0, 0)$.

Na políčku (A, B) stojí figúrka. Hráč na ťahu ju môže posunúť nanajvýš o K políčok, a to buď doľava, alebo dodola, alebo šikmo doľava dodola. Vyhráva ten, kto figúrku privedie do ľavého dolného rohu.

Táto hra je identická (presnejšie povedané, *izomorfná*) s hrou, ktorú hrajú Julka a Monika s koláčmi a ovocím. Prvá súradnica figúrky zodpovedá počtu koláčov na stole, druhá počtu kusov ovocia.

Redukcia priestoru stavov:

Predchádzajúce riešenie ešte stále nestačí na riešenie úlohy pre limity dané v zadaní. Ďalším krokom môže byť napríklad odpozorovanie nejakej závislosti v tabuľke vyhrávajúcich a prehrávajúcich pozícií. Jednu takúto závislosť si teraz dokážeme:

Tvrdíme, že pozícia (A, B) je vyhrávajúca práve vtedy, keď je vyhrávajúca pozícia $(A \bmod (K+1), B \bmod (K+1))$.

Intuícia za týmto tvrdením: Ak je ešte hra ďaleko od konca, vieme doplniť ťah súpera tak, aby sme dokopy odobrali aj koláčov, aj ovocia buď 0 alebo $K+1$ kusov, a tak zabezpečiť, že sa zvyšok nezmení.

Dôkaz budeme vysvetľovať pre hru s figúrkou na šachovnici. Predstavme si,

že šachovnicu rozdelíme na veľké štvorce so stranou $K+1$, začínajúc samozrejme v ľavom dolnom rohu.

Ukážeme najskôr, že ak je vyhrávajúca pozícia $(A \bmod (K+1), B \bmod (K+1))$, tak je vyhrávajúca aj pozícia (A, B).

Uvažujme nasledujúcu stratégiu:

- V prvom ťahu potiahneme rovnako ako by sme ťahali v pozícii $(A \bmod (K+1), B \bmod (K+1))$.
- Ak protihráč svojím predchádzajúcim ťahom prekročil hranicu veľkého štvorca, potiahneme tým istým smerom tak, aby sme dokopy posunuli figúrku presne o $K+1$.
  (Toto určite vieme urobiť, a dosiahneme tým, že figúrka bude na tej istej pozícii, len v inom veľkom štvorci ako bola.)
- V opačnom prípade potiahneme aj my len v rámci veľkého štvorca, a to rovnako ako by sme ťahali v tejto pozícii v ľavom dolnom veľkom štvorci.

Ak sa jej budeme držať, zjavne hru vyhráme.

Podobne sa dá ukázať, že ak je pozícia $(A \bmod (K+1), B \bmod (K+1))$ prehrávajúca, tak aj pozícia (A, B) je prehrávajúca – nech začneme ľubovoľným ťahom, súperovi na to, aby vyhral, stačá použiť práve popísanú stratégiu.

Ak použijeme algoritmus z časti "Šikovnejšie dynamické programovanie„, dostaneme riešenie s časovou aj pamäťovou zložitosťou $O(K^2)$.

Vzorové riešenie:

Všimnime si, že keďže nás už zaujímajú len pozície od $(0,0)$ po (K,K), môžeme úplne zabudnúť na to, že máme nejaké obmedzenie na počet vecí, ktoré môžeme v jednom ťahu zobrať. Budeme teda odteraz uvažovať hru, kde toto obmedzenie nie je.

Pre každé x určite existuje najviac jedno y také, že (x,y) je prehrávajúca pozícia. Zoberme totiž najmenšie y také, že $(x.y)$ je prehrávajúca. Potom pre ľubovoľné $z > y$ vieme z (x,z) ťahať do (x,y), a teda (x,z) je vyhrávajúca pozícia. (Dá sa dokonca dokázať, že existuje práve jedno také y.)

Nepotrebujeme teda dvojrozmerné pole pozícii, stačí nám pamätať si v poli pre každé x jemu zodpovedajúce y, ak sme ho už zistili.

Prehrávajúce pozície budeme zostrojovať usporiadané podľa $x+y$. Budeme mať dve polia: v jednom si pamätáme, kde je prehrávajúca pozícia v ktorom

riadku, v druhom, kde je na ktorej "uhlopriečke„. (Uhlopriečkami voláme množiny políčok "rovnobežné„ s hlavnou uhlopriečou tabuľky.)

Budeme v cykle prechádzať riadky od 0 po A. Vždy, keď natrafíme na riadok x, v ktorom ešte nepoznáme prehrávajúcu pozíciu, nájdeme ju jednoducho tak, že nájdeme k nemu prvý taký stĺpec y, že ešte nepoznáme prehrávajúcu pozíciu ani v stĺpci y, ani na uhlopriečke idúcej cez (x, y). Stĺpec y stačí hľadať napravo od stĺpca, ktorý sme našli k predchádzajúcemu riadku. Zaznačíme si, že v riadku x je prehrávajúca pozícia y. A zo symetrie vyplýva, že v riadku y je prehrávajúca pozícia x, to si zaznačíme tiež. V oboch prípadoch si príslušný údaj zapíšeme aj pre uhlopriečku, na ktorej práve nájdená pozícia leží.

Takto dostávame riešenie, ktoré všetky potrebné prehrávajúce pozície spočíta v čase $O(K)$. Pomocou nich vieme následne každý ťah spraviť v konštantnom čase.

Listing programu:

```
#include "dobrota.h"
#include <cstdio>
using namespace std;

int prehraR[MAX_K+1];
int prehraU[2*MAX_K+1];
int K;

void zaciatok (int A, int B, int _K) {
  K = _K;
  for (int i=0; i<=2*K; i++) prehraU[i] = -1;
  for (int i=0; i<=K; i++) prehraR[i] = -1;

  int stlpec = 0;
  for (int riadok = 0; riadok <= K; riadok++) {
    if (prehraR[riadok] >= 0) continue;
    while (prehraU[riadok-stlpec+K]>=0 || prehraR[stlpec]>=0) {
      stlpec++;
      if (stlpec > K) break;
    }
    if (stlpec > K) break;
    prehraR[riadok] = stlpec;
    prehraR[stlpec] = riadok;
    prehraU[riadok-stlpec+K] = riadok;
    prehraU[stlpec-riadok+K] = stlpec;
  }
}

void tahaj (int A, int B, int *ber_A, int *ber_B) {
  A %= K+1;
```

```
  B %= K+1;
  if (prehraR[A]!=-1 && B > prehraR[A]) {
    *ber_A = 0;
    *ber_B = B-prehraR[A];
    return;
  }
  if (prehraR[B]!=-1 && A > prehraR[B]) {
    *ber_B = 0;
    *ber_A = A-prehraR[B];
    return;
  }
  if (prehraU[A-B+K]!=-1 && A > prehraU[A-B+K]) {
    *ber_A = *ber_B = A-prehraU[A-B+K];
    return;
  }
  printf("zle je!\n"); *ber_A=*ber_B=0; // nemalo by nastat
}
```

Výsledky krajských kôl kategórie B

V kategórii B súťaž končí krajským kolom, nižšie uvedené sú výsledky tohto kola v siedmich z ôsmich krajov. (V Banskobystrickom kraji sa do kategórie B nezapojil žiaden riešiteľ.)

Výsledky neboli koordinované, je teda možné, že v rôznych krajoch boli použité mierne odlišné bodovacie škály.

Bratislavský kraj:

Meno	Ročník, škola	1.	2.	3.	4.	Σ
1. Boris Vavrík	1 Gym. Jura Hronca BA	9	1	7	8	25
2. Ján Hozza	1 Gym. Jura Hronca BA	5	4	4	10	23
3. Lukáš Polák	2 Gym. Jura Hronca BA	10	0	7	4	21
4. Juraj Masár	1 Gym. Bilíkova BA	6	1	7	4	18
5. Juraj Macháč	1 Gym. Jura Hronca BA	5	0	6	5	16
Mariana Phuong	1 Gym. Jura Hronca BA	5		7	4	16
Stanislav Párnický	2 Gym. Grösslingová BA	9	0	6	1	16
8. Bruno Cuc	2 Gym. Grösslingová BA	4	1	7	1	13
9. Marek Kukan	2 Gym. Grösslingová BA	7	0	0	4	11
10. Martin Strapko	1 Gym. Jura Hronca BA	5	0	0	0	5

Košický kraj:

Meno	Ročník, škola	1.	2.	3.	4.	Σ
1. Bačo Ladislav	2.A G Poštová KE	6	8	4	8	26
2. Klučár Marek	2.B G Javorová SNV	4	2	3	3	12
Novella Tomáš	sexta G Alejová KE	5	2	5	0	12
Rohár Pavol	sexta G Alejová KE	5	2	5		12
5. Peťura Oto	2.B SZS Partizánska MI	5	1	3	2	11
6. Gajdoš Tomáš	sexta G Alejová KE	5	3	2		10
7. Jančár Milan	2.A G P. Horova MI	5			4	9
8. Babej Tomáš	1.A G Poštová KE	8				8
Spišiak Matej	2.B G Javorová SNV	4		2	2	8
10. Rovňák Tomáš	2.B SZS Partizánska MI	4		3		7

Nitriansky kraj:

Meno	Ročník, škola	1.	2.	3.	4.	Σ
1. Lami Vince	Gym. H. Selyeho Komárno	3	3	1	1	8
2. Szabó Roland	Gym. H. Selyeho Komárno	4		1	2	7
3. Šabík Matúš	Gym. Golianova Nitra	4	0	1	0	5
Miklovič Tomáš	Gym. Ul. M. R. Štefánika Nové Zámky	4		1		5
Laiszner Tomáš	Gym. H. Selyeho Komárno	2	0	3	0	5

Prešovský kraj:

Meno	Ročník, škola	1.	2.	3.	4.	Σ
1. Anderko Maroš	2.D G Konštantínova PO	5		5	0	10
2. Kmec Peter	1.D G Konštantínova PO	5	0	4	0	9
3. Wittner Rudolf	1.D G Konštantínova PO	4		3	1	8
4. Phong Nguyen Tien	2.D G Konštantínova PO	4			3	7
5. Aštary Matej	2.D G Konštantínova PO	3	0	3	0	6
6. Fotta Michal	2.D G Konštantínova PO	3			0	3

Trenčiansky kraj:

Meno	Ročník, škola	1.	2.	3.	4.	Σ
1. Krejčír Andrej	2 Gym. Nedožerského Prievidza	6	0	4	5	15

Trnavský kraj:

Meno	Ročník, škola	1.	2.	3.	4.	Σ
1. Miroslav Rýzek	Gym. L. Novomeského Senica	4			3	7

Žilinský kraj:

Meno	Ročník, škola	1.	2.	3.	4.	Σ
1. Martin Habovštiak	2 Gym. Tvrdošín	5		4		9
2. Tomáš Dopita	2 Súkromné gym. Oravská Žilina	1	1	1	2	5

Výsledky celoštátneho kola kategórie A

Celoštátne kolo kategórie A sa v 23. ročníku Olympiády v informatike uskutočnilo v dňoch 20. až 23. apríla 2008 v hoteli Drotár v obci Hronec, okres Brezno. Spomedzi 28 pozvaných riešiteľov bolo najlepších 16 vyhlásených za úspešných riešiteľov a najlepší štyria za víťazov tohto ročníka OI.

	Meno	Ročník, škola	1.	2.	3.	4.	5.	Σ
1.	Peter Ondrúška	4 SPŠ Dubnica nad Váhom	9	10	6	4	15	44
2.	Peter Fulla	3 SPŠ Spišská Nová Ves	9	6	9	3	15	42
3.	Samuel Hapák	4 G. Grösslingová BA	9	7	10	0	15	41
	Vladimír Boža	4 G. Tatarku Poprad	10	7	6	3	15	41
5.	Tomáš Kočiský	4 G. Grösslingová BA	8	6	4	2	1	21
6.	Michal Spišiak	3 G. Grösslingová BA	8	7	5	0	0	20
7.	Albert Herencsár	3 G. maď. Galanta	5	5	6	3	–	19
8.	Matúš Kukan	4 G. Grösslingová BA	10	–	7	1	0	18
9.	Katarína Tureková	4 G. Tajovského B. Bystrica	7	5	4	1	–	17
	Lucia Simanová	4 G. Grösslingová BA	5	6	6	–	0	17
11.	Michal Petrucha	3 G. Metodova BA	5	5	2	–	4	16
12.	Tobiáš Hudec	2 G. Partizánske	6	2	5	0	2	15
13.	Martin Šrámek	3 G. Alejová Košice	7	–	3	3	1	14
	Lenka Matejovičová	3 G. Jura Hronca BA	3	4	7	0	0	14
	Filip Kubina	4 G. Dolný Kubín	5	5	3	1	0	14
	Peter Hlavatý	4 G. Jura Hronca BA	8	6	0	–	0	14
17.	Dušan Klinec	4 G. Nedožerského Prievidza	7	2	2	1	1	13
	Michal Hozza	3 G. Jura Hronca BA	7	0	6	0	0	13
19.	Dávid Štrbka	4 G. Grösslingová BA	7	2	2	1	0	12
20.	András Varga	4 G. H. Selyeho Komárno	4	0	6	1	0	11
	Kristián Nagy	4 G. Jura Hronca BA	7	–	2	2	0	11
22.	Alena Košinárová	4 G. Grösslingová BA	5	0	5	0	0	10
	Pavol Blaho	4 G. Jura Hronca BA	5	4	1	–	–	10
24.	Adam Saleh	4 G. Jura Hronca BA	4	–	4	–	1	9
	Michal Bubnár	3 G. sv. Františka Vranov n/T	4	–	–	4	1	9
26.	Ján Súkeník	4 G. Lettricha Martin	3	5	0	–	0	8
27.	Dominika Fedáková	4 G. Stará Ľubovňa	5	1	0	0	0	6
28.	Daniel Bene	4 G. Haličská Lučenec	3	–	1	–	0	4

Výsledky výberového sústredenia

V dňoch 10. až 16. mája sa v Bratislave konalo výberové sústredenie. Na toto sústredenie boli pozvaní najlepší riešitelia celoštátneho kola OI, kategórie A. Štyria najlepší riešitelia výberového sústredenia majú možnosť reprezentovať Slovensko na Medzinárodnej olympiáde v informatike. Na základe výberového sústredenia taktiež vyberá SK OI reprezentačný tím pre Stredoeurópsku olympiádu v informatike a pre prípravné sústredenia.

V nasledujúcej tabuľke sú postupne uvedené body za celoštátne kolo OI a za sedem súťažných dní výberového sústredenia.

Meno	$\sum$	CK	so	ne	po	ut	st	št	pi
1. Vladimír Boža	599.5	41.0	47.5	127.5	114.5	53.5	120.0	86.5	9.0
2. Peter Ondrúška	518.5	44.0	47.5	75.5	136.5	47.5	81.5	86.0	0.0
3. Peter Fulla	432.0	42.0	17.0	73.5	84.5	68.5	47.5	60.0	39.0
4. Samuel Hapák	429.0	41.0	16.5	77.0	98.5	56.0	62.5	45.5	32.0
5. Matúš Kukan	259.5	18.0	16.5	47.5	67.0	54.5	15.0	35.0	4.0
6. Martin Šrámek	242.5	14.0	22.0	47.5	68.5	19.0	17.5	37.0	17.0
7. Tomáš Kočiský	227.5	21.0	15.5	45.0	58.5	38.0	0.0	39.0	10.5
8. Michal Petrucha	215.0	16.0	32.5	35.5	72.5	0	17.0	34.0	7.5
9. Albert Herencsár	206.5	19.0	13.0	41.5	54.5	40.0	14.0	19.5	5.0
10. Katarína Tureková	171.0	17.0	5.5	26.5	20.5	39.5	15.5	40.5	6.0
11. Michal Spišiak	150.5	20.0	1.0	27.0	24.0	0.0	20.0	43.0	15.5
12. Tobiáš Hudec	86.0	15.0	0.0	0.0	13.0	21.0	12.5	19.5	5.0
13. Lucia Simanová	85.5	17.0	3.0	23.0	21.5	8.0	0.0	13.0	0.0

Slovensko-švajčiarske prípravné sústredenia

Vďaka blízkej spolupráci medzi národnými olympiádami v informatike na Slovensku a vo Švajčiarsku mali vybraní riešitelia OI možnosť zúčastniť sa dvoch prípravných sústredení vo Švajčiarsku. Prvé z nich sa konalo v Davose v dňoch 10. až 16. februára 2008, druhé v Zürichu v dňoch 13. až 18. júla.

Výsledky prípravného sústredenia v Davose:

Meno	q	d1	d2	d3	d4	$\sum$
1. Peter Ondrúška	39.5	94	82	72	110	397.5
Vladimir Serbinenko	35.5	99	64	110	89	397.5
3. Vladimír Boža	29.5	87	60	96	92	364.5
4. Samuel Hapák	19.5	57	35	64	83	258.5
5. Peter Fulla	34	39	57	61	48	239.0
6. Isaac Deutsch	25.5	35	40	45	59	204.5
7. Johannes Josi	19.5		35	54	82	190.5
8. Adrian Roos	24	38	40	43	44	189.0
9. Daniel Graf	32	40	26	15	60	173.0
10. Joel Bohnes	20	44	35	46	27	172.0
11. Beat Küng	27	43	20	34	33	157.0
12. Samuel Hitz	21	35	34	36	20	146.0
13. Florian Scheidegger	30.5	36	35	30	10	141.5
Yannick Stucki	23.5	43	16	13	46	141.5
15. Jonas Wagner	15.5	42	16	17	30	120.5
16. Franzi ?	5	35	35	10	35	120.0
17. Sebastien Vasey	12			42	22	76.0

Výsledky prípravného sústredenia v Zürichu:

Meno	d1	d2	d3	d4	$\sum$
1. Vladimír Boža	398	292	329	380	1399
2. Peter Ondrúška	398	249	196	395	1238
3. Vladimir Serbinenko	328	202	238	260	1028
4. Martin Šrámek	207	94	150	200	651
5. Michal Petrucha	271	64	133	95	563
6. Adrian Roos	215	8	135	195	553
7. Isaac Deutsch	148	68	130	195	541

Česko-poľsko-slovenské prípravné sústredenie

U súťažného stretnutia českých, poľských a slovenských stredoškolákov v informatike už smelo môžeme hovoriť o tradícii – v roku 2008 sa konal už desiaty ročník tejto akcie. Sústredenie sa konalo v dňoch 22. až 28. júna 2008, miestom konania boli Danišovce – malá dedinka neďaleko Spišskej Novej Vsi. (Po štvrtý krát bola táto akcia organizovaná na Slovensku.)

Organizačnú, technickú a odbornú stránku akcie zabezpečovali v súčinnosti ľudia z PF UPJŠ v Košiciach (doc. RNDr. Gabriela Andrejková, CSc., RNDr. Rastislav Krivoš-Belluš, Mgr. Ján Katrenič, Ján Jerguš) a z FMFI UK v Bratislave (RNDr. Michal Forišek, Peter Perešíni, Lukáš Poláček, Martin Rejda). Ako sprievod súťažiacich a autori časti súťažných úloh sa akcie zúčastnili z ČR Pavel Nejedlý a z Poľska Piotr Niedzwiedz, Wojciech Smietanka a Jakub Radoszewski.

Účastníci sústredenia absolvovali štyri súťažné dni v rovnakom štýle ako na Medzinárodnej olympiáde v informatike. Súčasťou programu bol taktiež celodenný výlet do neďalekého Slovenského raja. Výsledky sústredenia sú uvedené v nasledujúcej tabuľke.

Meno, krajina		$\sum$	day 1	day 2	day 3	day 4
1. Marcin Andrychowicz	POL	960.0	230.0	140.0	290.0	300.0
2. Marcin Kościelnicki	POL	690.0	200.0	150.0	100.0	240.0
3. Tomasz Kleczek	POL	521.0	106.0	70.0	190.0	155.0
4. Peter Ondrúška	SVK	510.0	150.0	10.0	110.0	240.0
5. Jarosław Błasiok	POL	500.0	180.0	10.0	110.0	200.0
6. Maciej Klimek	POL	478.0	88.0	30.0	60.0	300.0
7. Maciej Andrejczuk	POL	392.0	62.0	0	50.0	280.0
8. Matúš Kukan	SVK	390.0	160.0	30.0	60.0	140.0
9. Peter Fulla	SVK	365.0	100.0	20.0	120.0	125.0
10. Roman Smrž	CZE	355.0	100.0	30.0	50.0	175.0
11. David Klaška	CZE	276.0	136.0	20.0	50.0	70.0
12. Libor Plucnar	CZE	245.0	160.0	0	0	85.0
13. Vojtěch Tůma	CZE	224.0	144.0	30.0	20.0	30.0
14. Michal Petrucha	SVK	175.0	60.0	20.0	50.0	45.0
15. Martin Šrámek	SVK	140.0	0	10.0	60.0	70.0
16. Hynek Jemelík	CZE	68.0	8.0	10.0	10.0	40.0

Stredoeurópska olympiáda v informatike

V roku 2008 sa Stredoeurópska olympiáda v informatike (CEOI) konala v dňoch 6. až 12. júla v nemeckých Drážďanoch. Súťaže sa zúčastnili tímy z tradičných siedmich organizujúcich krajín: domáceho Nemecka, Chorvátska, Českej republiky, Maďarska, Poľska, Rumunska a Slovenska. Súťaže sa tiež zúčastnil tím z Izraela, ktorý bol pozvaný organizátormi, a taktiež tím reprezentujúci spolkovú republiku Sasko.

Ako lídri zastupujúci našu krajinu sa tejto CEOI zúčastnili doc. RNDr. Gabriela Andrejková, CSc. z ÚI PF UPJŠ v Košiciach a Bc. Marek Zeman z FMFI UK v Bratislave.

Ako už býva tradíciou, Slovensko na rozdiel od ostatných krajín na CEOI posiela delegáciu zloženú prevažne zo súťažiacich, ktorí ešte v danom roku nekončia strednú školu. Tak tomu bolo aj v roku 2008, kedy Slovensko reprezentovali nasledujúci stredoškoláci: Peter Fulla z SPŠ Spišská Nová Ves, Matúš Kukan z Gymnázia Grösslingová v Bratislave, Martin Šrámek z Gymnázia Alejová v Košiciach a Michal Petrucha z Gymnázia Metodova v Bratislave. Výsledky našich súťažiacich uvádzame zhrnuté v tabuľke.

	Meno	1. deň			2. deň			$\sum$	medaila
8.	Peter Fulla	100	0	98	40	0	60	298	strieborná
26.	Matúš Kukan	0	8	41	0	0	100	149	–
31.	Martin Šrámek	35	46	0	40	0	2	123	–
35.	Michal Petrucha	0	0	41	0	0	9	50	–

Medzinárodná olympiáda v informatike

V roku 2008 sa Medzinárodná olympiáda v informatike (IOI) konala v dňoch 16. až 23. augusta v Egypte neďaleko Káhiry. Súťaže sa zúčastnilo 283 súťažiacich z takmer 80 krajín celého sveta.

Ako lídri zastupujúci našu krajinu pri hlasovaniach sa tejto IOI zúčastnili doc. RNDr. Gabriela Andrejková, CSc. z ÚI PF UPJŠ v Košiciach a Mgr. Juliana Lipková z FMFI UK v Bratislave. Ako člen medzinárodnej odbornej komisie (ISC) sa tejto IOI zúčastnil RNDr. Michal Forišek z FMFI UK v Bratislave.

Našu krajinu tento rok reprezentovala táto štvorica stredoškolákov: Vladimír Boža z Gymnázia D. Tatarku v Poprade, Peter Fulla z SPŠ Spišská Nová Ves, Samuel Hapák z Gymnázia Grösslingová v Bratislave a Peter Ondrúška z SPŠ Dubnica nad Váhom. Výsledky našich súťažiacich uvádzame zhrnuté v tabuľke.

	Meno	1. deň			2. deň			$\sum$	medaila
33.	Vladimír Boža	100	4	26	30	100	25	285	strieborná
63.	Peter Ondrúška	100	10	13	0	80	40	243	strieborná
92.	Peter Fulla	100	0	–	–	100	0	200	bronzová
138.	Samuel Hapák	100	0	29	–	0	–	129	bronzová

Zadania prvého súťažného dňa

Ostrovy

Práve ste navštívili park, ktorý má N ostrovov. Z každého ostrova i bol postavený práve jeden most. Dĺžka tohto mosta je označená L_i. Celkový počet mostov v parku je teda N. Hoci bol každý most postavený z jedného ostrova na iný, teraz je možné prechádzať cez každý most oboma smermi. Zároveň pre každú dvojicu ostrovov existuje práve jedna kompa, ktorá premáva tam a späť medzi nimi.

Pretože radšej chodíte peši, než sa prevážate na kompe, chcete maximalizovať súčet dĺžok mostov, cez ktoré prejdete. Pritom musíte dodržať nasledujúce podmienky:

- Môžete začať na ostrove, ktorý si vyberiete.
- Žiadny ostrov nesmiete navštíviť viac než jedenkrát.
- V ľubovoľnom okamihu sa môžete presunúť z daného ostrova S na iný ostrov D, ktorý ste predtým nenavštívili. Môžete sa presunúť:
  - Peši: Je to možné len vtedy, keď medzi ostrovami S a D existuje most. Pri tejto voľbe je dĺžka mosta pripočítaná k celkovej vzdialenosti, ktorú ste prešli peši.
  - Kompou: Je to možné len vtedy, keď D nie je dosiahnuteľný z S použitím kombinácie mostov a/alebo predtým použitých kômp. (Pri kontrole dosiahnuteľnosti musíte uvažovať všetky cesty, aj cesty prechádzajúce cez ostrovy, ktoré ste už navštívili.)

Poznamenajme, že nemusíte navštíviť všetky ostrovy a tiež sa môže stať, že neprejdete cez všetky mosty.

Súťažná úloha:

Napíšte program, ktorý pre daných N mostov s danými ich dĺžkami vypočíta maximálnu vzdialenosť, ktorú prejdete cez mosty dodržaním vyššie uvedených pravidiel.

Ohraničenia:

$2 \leq N \leq 1,000,000$	Počet ostrovov v parku.
$1 \leq L_i \leq 100,000,000$	Dĺžka mosta i.

Vstup:

Váš program musí čítať zo štandardného vstupu nasledujúce údaje:

- Riadok 1 obsahuje celé číslo N, počet ostrovov v parku. Ostrovy sú očíslované od 1 do N, vrátane.
- Každý z nasledujúcich N riadkov popisuje most. i-ty riadok popisuje most skonštruovaný z ostrova i pomocou dvoch celých čísel oddelených jedinou medzerou. Prvé celé číslo reprezentuje ostrov, na ktorom most končí, druhé celé číslo predstavuje dĺžku mosta – L_i.

  Môžete predpokladať, že každý most má svoje konce vždy na dvoch rôznych ostrovoch.

Výstup:

Program musí vypísať na štandardnom výstupe jediný riadok obsahujúci jediné celé číslo, maximálnu prejdenú vzdialenosť.

Poznámka 1. Pre niektoré prípady odpoveďou nemusí byť 32-bitové celé číslo, budete potrebovať `int64` v Pascale alebo `long long` v C/C++, aby ste mohli získať plný počet bodov za túto úlohu.

Poznámka 2. Keď pracujete v Pascale, načítavanie čísel zo štandardného vstupu do 64-bitových typov je výrazne pomalšie než do 32-bitových typov, a to aj v prípade, že ide o čísla, ktoré sa zmestia do 32 bitov. Odporúčame vám načítavať do 32-bitových typov.

Bodovanie:

Pre niektoré prípady v celkovej hodnote 40 bodov, N neprekročí 4 000.

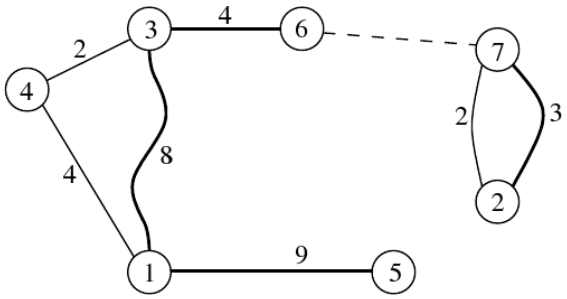
Príklad:

Vstup

```
7
3 8
7 2
4 2
1 4
1 9
3 4
2 3
```

Výstup

```
24
```



$N = 7$ mostov v príklade je $(1-3), (2-7), (3-4), (4-1), (5-1), (6-3)$ a $(7-2)$. Poznamenajme, že existujú dva rôzne mosty spájajúce ostrovy 2 a 7. Jedna z možností, ako dosiahnuť maximálnu prejdenú vzdialenosť, je nasledujúca:

- Štart na ostrove 5.
- Prejsť most dĺžky 9 na ostrov 1.
- Prejsť most dĺžky 8 na ostrov 3.
- Prejsť most dĺžky 4 na ostrov 6.
- Použiť kompu z ostrova 6 na ostrov 7.
- Prejsť most dĺžky 3 na ostrov 2.

Na konci ste na ostrove 2 a vami prejdená vzdialenosť je $9+8+4+3 = 24$. Len jeden ostrov 4 ste nenavštívili. Poznamenajme, že na konci vyššie popísaného výletu tento ostrov už nemôžete navštíviť. Presnejšie:

- Nemôžete ho navštíviť peši, pretože neexistuje žiadny most spájajúci ostrov 2 (kde sa práve nachádzate) a ostrov 4.

- Nemôžete ho navštíviť použitím kompy, pretože ostrov 4 je dosiahnuteľný z ostrova 2, kde sa práve nachádzate. Spôsob dosiahnutia je nasledujúci: použiť most (2-7), potom použiť kompu, ktorú ste už použili, z ostrova 7 na ostrov 6, potom most (6-3), a nakoniec most (3-4).

Tlačiareň

Práve ste dostali za úlohu vytlačiť N slov na staručkej písmenkovej tlačiarni. Na to, aby ste na nej nejaké slovo mohli vytlačiť, musíte do nej zasunúť kovové odliatky písmen slova (každý odliatok obsahuje práve jedno písmenko). Keď ich všetky zasuniete, môžete slovo vytlačiť na papier.

Na vašej tlačiarni teda môžete robiť nasledovné operácie:

- Pridať jedno písmenko na koniec slova práve pripraveného v tlačiarni.
- Odobrať jedno písmenko z konca slova práve pripraveného v tlačiarni. (Táto operácia je možná iba ak máte v tlačiarni aspoň jedno písmenko.)
- Vytlačiť slovo práve pripravené v tlačiarni.

Na začiatku je tlačiareň prázdna, t.j. nie sú v nej zasunuté žiadne písmenká. Na konci tlače môže zostať v tlačiarni akékoľvek slovo. Dané slová môžete tlačiť v ľubovoľnom poradí. Vaším cieľom je minimalizovať celkový počet operácií.

Súťažná úloha:

Napíšte program, ktorý pre daných N slov vypočíta minimálny počet operácií potrebných na ich vytlačenie. Slová môžu byť vytlačené v ľubovoľnom poradí, každé musí byť vytlačené práve raz. Váš program má taktiež vypísať príslušnú postupnosť operácií (ak existuje viac riešení, môžete vypísať ľubovoľné z nich).

Ohraničenia:

$1 \leq N \leq 25\,000$ počet slov, ktoré treba vytlačiť

Vstup:

Váš program má načítať nasledovné údaje zo štandardného vstupu:

- Prvý riadok obsahuje celé číslo N – počet slov, ktoré treba vytlačiť.
- Každý z nasledovných N riadkov obsahuje jedno slovo. Každé slovo pozostáva iba z malých písmen (`a` – `z`) a má dĺžku medzi 1 a 20 vrátane. Žiadne dve slová na vstupe nie sú rovnaké.

Výstup:

Váš program má vypísať nasledovné údaje na štandardný výstup:

- Prvý riadok má obsahovať celé číslo M – minimálny počet operácií potrebných na vytlačenie daných N slov.

- Nasledujúcich M riadkov popisuje postupnosť operácií potrebných na vytlačenie daných slov. Každý z týchto riadkov má obsahovať jeden znak, ktorý nasledovne popisuje príslušnú operáciu:
  – malé písmenko: pridanie príslušného písmenka
  – znak '`-`' (mínus, ASCII kód 45): vymazanie posledného písmenka
  – znak '`P`' (veľké P): vytlačenie práve pripraveného slova

Bodovanie:

V testovacích vstupoch v celkovej hodnote 40 bodov nepresiahne počet slov N hodnotu 18.

Podrobná spätná väzba:

Počas súťaže budú vaše odovzdané riešenia vyhodnotené na časti oficiálnych testovacích dát. O výsledkoch tohoto testovania budete informovaní.

Príklad:

Vstup

```
3
print
the
poem
```

Výstup

```
20
t
h
e
P
-
-
-
p
o
e
m
P
-
-
-
r
i
n
t
P
```

Rybiška

Určite ste už počuli povesť o Jazere, ktoré nikto nikdy nenašiel. Žijú v ňom rybišky a tie rybišky jedia rybišky, ktoré jedia rybišky. A nie, tie rybišky chudášik Šmígol neje. Pretože ani on toto Jazero nikdy nenašiel.

Kedysi dávno žilo v Jazere F rybišiek. Domorodci doniesli K rôznych typov kamienkov a hodili ich rybiškám. Rybišky sa rozpŕchli za kamienkami a každá zjedla práve jeden kamienok. Kedže K môže byť menšie ako F, niektoré rôzne rybišky môžu zjesť rovnaký typ kamienka.

Čas bežal a rybišky sa papali. Avšak nie každá rybiška vie zjesť inú rybišku. Rybiška vie zjesť inú, iba ak je aspoň dvakrát taká dlhá. (Rybiška A vie zjesť rybišku B práve vtedy, keď $L_A \geq 2 \cdot L_B$.) Nikde nie je povedané, kedy zje rybiška inú rybišku. Niektoré rybišky sú pažravé a rozhodnú sa jesť jednu za druhou, niektoré sú menej pažravé a dávajú si pauzu medzi jednotlivými chodmi, a niektoré možno nie sú nikdy hladné. Ak rybiška zje menšiu rybišku, nezmení sa jej dĺžka. Avšak kamienok zo žalúdka menšej rybišky skončí nepoškodený v žalúdku väčšej.

V povesti sa vraví, že ak by ste raz našli Jazero, bude vám povolené vyloviť jedinú rybišku a ponechať si kamienky z jej žalúdka. Radi by ste išli hľadať Jazero, ale skôr ako sa vydáte na túto dlhočiznú cestu, oplatí sa zistiť si, koľko rôznych kombinácii kamienkov viete takto získať.

Súťažná úloha:

Napíšte program, ktorý pre zadané dĺžky rybišiek a typy kamienkov, ktoré na začiatku rybišky prehltli, nájde **počet rôznych kombinácií kamienkov, ktoré môžu skončiť v žalúdku ľubovoľnej rybišky *modulo zadané celé číslo*** M. Kombinácia je jednoznačne určená počtami kamienkov jednotlivých K typov. Je jedno, ako sú kamienky zoradené a kamienky rovnakého typu sú nerozlíšiteľné.

Ohraničenia:

$1 \geq F \geq 500\,000$	Pôvodný počet rybišiek
$1 \geq K \geq F$	Počet rôznych typov kamienkov
$2 \geq M \geq 30\,000$	
$1 \geq L_X \geq 1\,000\,000\,000$	Dĺžka rybišky X

Vstup:

Váš program musí zo štandardného vstupu načítať nasledujúce údaje:

- Riadok 1 obsahuje jediné celé číslo F, pôvodný počet rybišiek v jazere.

- Riadok 2 obsahuje jediné celé číslo K, počet rôznych typov kamienkov. Jednotlivé typy kamienkov sú očíslované číslami 1 až K, vrátane.
- Riadok 3 obsahuje jediné celé číslo M.
- Nasledujúcich F riadkov popisuje jednotlivé rybišky a obsahuje dvojicu celých čísel oddelených jedinou medzerou. Prvé číslo je dĺžka rybišky a druhé typ kamienku, ktorý rybiška na začiatku prehltla.

Poznámka: Pre každý testovací vstup vám garantujeme, že z každého druhu kamienkov bude aspoň jeden prítomný.

Výstup:

Váš program musí na štandardný výstup vypísať jediný riadok obsahujúci celé číslo z intervalu 0 až $M-1$ (vrátane): počet rôznych kombinácií kamienkov *modulo* M. Na vyriešenie úlohy nie je hodnota čísla M nijak dôležitá, v zadaní je iba na zjednodušenie výpočtov.

Bodovanie:

V testovacích vstupoch s celkovou hodnotou 70 bodov číslo K nepresiahne hodnotu 7 000. V podmnožine týchto vstupov s celkovou hodnotou 25 bodov číslo K dokonca nepresiahne hodnotu 20.

Podrobná spätná väzba:

Počas súťaže budú vaše odovzdané riešenia vyhodnotené na časti oficiálnych testovacích dát. O výsledkoch tohoto testovania budete informovaní.

Príklad:

Vstup	Výstup
5 3 7 2 2 5 1 8 3 4 1 2 3	4

Existuje 11 možných kombinácií, takže by ste mali dať na výstup 11 mod 7, čo je 4. Možné kombinácie sú: [1], [1, 2], [1, 2, 3], [1, 2, 3, 3], [1, 3], [1, 3, 3], [2], [2, 3], [2, 3, 3], [3] a [3, 3]. (Pre každú kombináciu sme uviedli zoznam kamienkov, ktoré obsahuje. Napr. [2, 3, 3] reprezentuje kombináciu, ktorá obsahuje jeden kamienok typu 2 a dva kamienky typu 3.) Uvedené kombinácie môžeme dosiahnuť nasledovne:

- [1]: Ihneď chytíte druhú (alebo štvrtú) rybišku, skôr než stihne zjesť nejakú ďalšiu.
- [1, 2]: Ak druhá rybiška zje prvú, bude mať v žalúdku kamienok typu 1 (ktorý pôvodne prehltla) a typu 2 (obsah žalúdka zjedenej rybišky).
- [1, 2, 3]: Jedna z možností dosiahnutia tejto kombinácie: štvrtá rybiška zje prvú, a potom tretia zje štvrtú. Ak teraz chytíte tretiu rybišku, získate po jednom kamienku z každého druhu.
- [1, 2, 3, 3]: Štvrtá zje prvú, tretia štvrtú, tretia piatu, chytíte tretiu.
- [1, 3]: Tretia zje štvrtú a vy ju chytíte.
- [1, 3, 3]: Tretia zje piatu, tretia štvrtú a vy ju chytíte.
- [2]: Chytíte prvú rybišku.
- [2, 3]: Tretia zje prvú a chytíte tretiu.
- [2, 3, 3]: Tretia zje prvú, potom piatu a chytíte tretiu.
- [3]: Chytíte tretiu rybišku.
- [3, 3]: Tretia zje piatu a vy chytíte tretiu.

Zadania druhého súťažného dňa

Pyramídy

Požiadali vás vybrať miesto na postavenie čo najväčšej pyramídy. K dispozícii máte popis pozemku, na ktorom má pyramída stáť. Pozemok má tvar obdĺžnika rozdeleného na $M \times N$ štvorcových políčok. Pôdorys pyramídy musí byť štvorec so stranami rovnobežnými so stranami pozemku.

Popis pozemku pozostáva z P prekážok. Každá prekážka má tvar obdĺžnika, ktorý sa celý nachádza na pozemku a jeho strany sú rovnobežné so stranami pozemku. Jednotlivé prekážky sa môžu prekrývať. Odstránenie i-tej prekážky má cenu C_i. Prekážku musíte buď celú odstrániť, alebo celú ponechať; nie je možné odstrániť iba časť prekážky. Pôdorys pyramídy musí byť umiestnený na časti pozemku, z ktorej boli všetky prekážky odstránené. Odstránenie prekážky nemá vplyv na prekážky, ktoré sa s ňou prekrývajú.

Súťažná úloha:

Napíšte program, ktorý načíta rozmery pozemku $M \times N$, pozície P prekážok, ceny na odstránenie každej z nich a rozpočet na stavbu pyramídy B. Program má vypočítať, akú najväčšiu dĺžku strany môže mať pôdorys pyramídy za predpokladu, že celková cena odstránených prekážok je najviac B.

Ohraničenia a bodovanie:

Váš program bude bodovaný na troch disjunktných sadách testov. Pre každý test platia nasledovné ohraničenia:

– Rozmery pozemku: $1 \leq M, N \leq 1\,000\,000$
– Cena za odstránenie i-tej prekážky: $1 \leq C_i \leq 7\,000$
– Súradnice rohových políčok prekážok: $1 \leq X_{i1} \leq X_{i2} \leq M$
$1 \leq Y_{i1} \leq Y_{i2} \leq N$

Prvá sada testov, v celkovej hodnote 35 bodov:
– Rozpočet na stavbu: $B = 0$ (Neviete odstrániť žiadne prekážky.)
– Počet prekážok na pozemku: $1 \leq P \leq 1\,000$

Druhá sada testov, v celkovej hodnote 35 bodov:
– Rozpočet na stavbu: $0 < B \leq 2\,000\,000\,000$
– Počet prekážok na pozemku: $1 \leq P \leq 30\,000$

Tretia sada testov, v celkovej hodnote 30 bodov:
– Rozpočet na stavbu: $B = 0$ (Neviete odstrániť žiadne prekážky.)
– Počet prekážok na pozemku: $1 \leq P \leq 400\,000$

Vstup:

Váš program má načítať nasledovné údaje zo štandardného vstupu:

- Prvý riadok obsahuje dve celé čísla M, N oddelené jednou medzerou.
- Druhý riadok obsahuje celé číslo B – maximálnu celkovú cenu odstránených prekážok.
- Tretí riadok obsahuje celé číslo P – počet prekážok na pozemku.
- Každý z nasledujúcich P riadkov popisuje jednu prekážku. Presnejšie, i-ty z týchto riadkov popisuje i-tu prekážku a obsahuje 5 celých čísel oddelených medzerou: X_{i1}, Y_{i1}, X_{i2}, Y_{i2} a C_i. Čísla popisujú (v danom poradí) súradnice najľavejšieho najdolnejšieho políčka i-tej prekážky, súradnice jej najpravejšieho najhornejšieho políčka a cenu za jej odstránenie. Ľavé dolné políčko pozemku má súradnice $(1, 1)$ a pravé hodné políčko má súradnice (M, N).

Výstup:

Váš program má vypísať na štandardný výstup jediný riadok obsahujúci jedno celé číslo – maximálnu možnú dĺžku strany pôdorysu pyramídy. Ak nie je možné postaviť žiadnu pyramídu, váš program má vypísať číslo 0.

Podrobná spätná väzba:

Počas súťaže budú vaše odovzdané riešenia vyhodnotené na časti oficiálnych testovacích dát. O výsledkoch tohoto testovania budete informovaní.

Príklad:

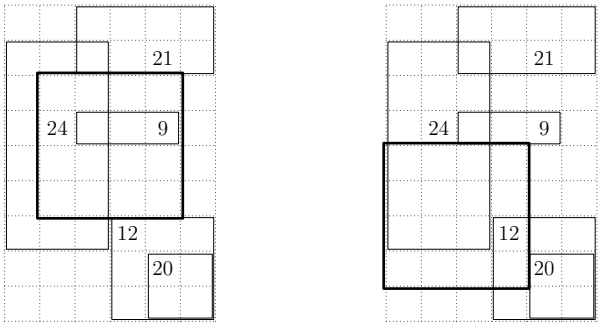
Vstup

```
6 9
42
5
4 1 6 3 12
3 6 5 6 9
1 3 3 8 24
3 8 6 9 21
5 1 6 2 20
```

Výstup

```
4
```

Dve možné polohy pôdorysu 4×4*:*



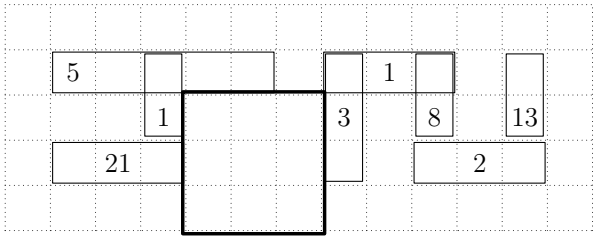
Vstup

```
13 5
0
8
8 4 10 4 1
4 3 4 4 1
10 2 12 2 2
8 2 8 4 3
2 4 6 4 5
10 3 10 4 8
12 3 12 4 13
2 2 4 2 21
```

Výstup

```
3
```

Jediná možná poloha pôdorysu 3×3



Teleporty

Prihlásili ste sa do súťaže, ktorej cieľom je prejsť pozdĺž danej úsečky Egypt od západu na východ. Na začiatku sa súťažiaci nachádzajú v najzápadnejšom bode úsečky. V súťaži platí pravidlo, že súťažiaci sa musia pohybovať len pozdĺž úsečky, a to iba smerom na východ.

Na úsečke je N teleportov. Každý teleport má dva koncové body. Vždy, keď súťažiaci dosiahne jeden z koncových bodov teleportu, je teleportovaný na jeho druhý koniec. (Poznamenajme, že v závislosti od dosiahnutého koncového bodu teleportu teleportácia môže byť smerom na východ alebo na západ.) Po teleportácii súťažiaci musí pokračovať smerom na východ pozdĺž úsečky. Na svojej ceste sa nikdy nemôže vyhnúť koncovým bodom teleportu. Žiadne dva koncové body teleportov nie sú na tej istej pozícii. Koncové body sú umiestnené vo vnútri medzi štartom a cieľom úsečky.

Pri každej teleportácii súťažiaci získava 1 bod. Cieľom súťaže je získať čo najviac bodov. Súťažiaci má povolené pred svojím štartom pridať nanajvýš M nových teleportov na úsečku. Ak použije nový teleport, tiež získava bod.

Koncové body nových teleportov môžu byť umiestnené hocikde, aj do neceločíselných pozícií. Pritom ale naďalej všetky koncové body všetkých teleportov musia byť rôzne. Tiež platí, že koncové body nových teleportov musia ležať vo vnútri medzi štartom a cieľom úsečky.

Poznamenajme, že je zaručené, že nezávisle od toho, ako sú teleporty pridané, súťažiaci sa dostane do cieľa.

Súťažná úloha:

Napíšte program, ktorý pre dané pozície N teleportov a daný počet M nových teleportov, ktoré je možné pridať, vypočíta maximálny počet bodov, ktoré je možné získať.

Ohraničenia:

$1 \leq N \leq 1\,000\,000$	Počet počiatočných teleportov.
$1 \leq M \leq 1\,000\,000$	Maximálny počet teleportov, čo smieme pridať.
$1 \leq W_X < E_X \leq 2\,000\,000$	Vzdialenosti západného a východného koncového bodu teleportu X od začiatku úsečky.

Vstup:

Váš program musí zo štandardného vstupu načítať nasledujúce údaje:

- Riadok 1 obsahuje celé číslo N, počet počiatočných teleportov.
- Riadok 2 obsahuje celé číslo M, maximálny počet teleportov, ktoré je možné pridať.
- Každý z nasledujúcich N riadkov popisuje jeden teleport. Presnejšie i-ty riadok popisuje i-ty teleport. Každý riadok pozostáva z dvoch celých čísel: W_i a E_i oddelených medzerou. Tieto predstavujú vzdialenosť od štartovacieho bodu úsečky k západnému a východnému koncovému bodu teleportu.

Žiadne dva koncové body nemajú tú istú pozíciu. Úsečka začína v bode 0 a končí v bode 2 000 001

Výstup:

Váš program musí na štandardnom výstupe vypísať jediný riadok, ktorý obsahuje jedno celé číslo, maximálny počet získaných bodov.

Bodovanie:

V testovacích vstupoch s celkovou hodnotou do 30 bodov platí $N \leq 500$ a $M \leq 500$.

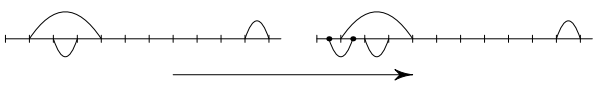
Príklad:

Vstup

```
3
1
10 11
1 4
2 3
```

Výstup

```
6
```



Prvý obrázok predstavuje úsečku s troma originálnymi teleportami. Druhý obrázok zobrazuje tú istú úsečku po pridaní nového teleportu s koncovými bodmi v 0.5 a 1.5.

Po pridaní nového teleportu, ako je ukázané na obrázku, cesta súťažiaceho bude nasledujúca:

- Štartuje v pozícii 0 smerom na východ.
- Dostane sa do pozície 0.5 a je teleportovaný do pozície 1.5, získava 1 bod.
- Pokračuje smerom na východ a dostane sa do pozície 2; je teleportovaný do pozície 3 (už má 2 body).
- Dostane sa do pozície 4 a je teleportovaný do pozície 1, získava 1 bod (už má 3 body).
- Dostane sa do pozície 1.5 a je teleportovaný do pozície 0.5, získava 1 bod (už má 4 body).
- Dostane sa do pozície 1 a je teleportovaný do pozície 4, získava 1 bod (už má 5 bodov).
- Dostane sa do pozície 10 a je teleportovaný do pozície 11, získava 1 bod (už má 6 bodov).
- Pokračuje, pokiaľ nedosiahne koniec úsečky a končí s celkovým skóre 6 bodov.

Vstup

```
3
3
5 7
6 10
1999999 2000000
```

Výstup

```
12
```

Záhrada na priamke

Ramzes II. sa práve vrátil domov z víťaznej bitky. Na oslavu svojho víťazstva sa rozhodol postaviť honosnú záhradu. Jeho záhradu má tvoriť jeden dlhý rad rastlín, ktorý bude siahať od jeho paláca v Luxore až ku chrámu v Karnaku. V záhrade majú byť iba lotosy a papyrusy, ktoré sú symbolom Horného a Dolného Egypta.

Záhrada musí obsahovať práve N rastlín. Taktiež musí byť vyvážená: v ľubovoľnej súvislej časti záhrady sa počet lotosov a počet papyrusov nesmú líšiť o viac než 2.

Záhrada sa dá reprezentovať ako reťazec znakov `L` a `P`. Napríklad pre $N = 5$ existuje 14 vyvážených záhrad. V abecednom poradí to sú tieto: `LLPLP`, `LLPPL`, `LPLLP`, `LPLPL`, `LPLPP`, `LPPLL`, `LPPLP`, `PLLPL`, `PLLPP`, `PLPLL`, `PLPLP`, `PLPPL`, `PPLLP` a `PPLPL`.

Vyvážené záhrady určitej dĺžky sa dajú usporiadať podľa abecedy a následne očíslovať začínajúc číslom 1. Napríklad pre $N = 5$ je záhrada s číslom 12 reprezentovaná ako `PLPPL`.

Súťažná úloha:

Napíšte program, ktorý pre zadaný počet rastlín N, celé číslo M a reťazec reprezentujúci vyváženú záhradu vypočíta číslo záhrady **modulo zadané celé číslo** M.

Poznamenajme, že hodnota M zjednodušuje výpočet, ale inak nemá vplyv na riešenie úlohy.

Ohraničenia:

$1 \leq N \leq 1\,000\,000$
$7 \leq M \leq 10\,000\,000$

Bodovanie:

V testovacích vstupoch, ktorých celková hodnota je 40 bodov, N nepresiahne hodnotu 40.

Vstup:

Váš program musí zo štandardného vstupu načítať nasledujúce údaje:

- Riadok 1 obsahuje jediné celé číslo N, počet rastlín v záhrade.
- Riadok 2 obsahuje jediné celé číslo M.
- Riadok 3 obsahuje reťazec reprezentujúci vyváženú záhradu z N znakov `L` (lotus) alebo `P` (papyrus).

Výstup:

Váš program musí na štandardný výstup vypísať jediný riadok obsahujúci celé číslo z intervalu 0 až $M - 1$ (vrátane): číslo, ktorým je daná záhrada očíslovaná, **modulo** M.

Príklad:

Vstup

```
5
7
PLPPL
```

Výstup

```
5
```

V skutočnosti má táto záhrada číslo 12. *Výstup je hodnota* 12 *modulo* 7, *čo je* 5.

Vstup

```
12
10000
LPLLPLPPLPLL
```

Výstup

```
39
```

RNDr. Michal Forišek, Slovenská komisia OI
Dvadsiaty tretí ročník Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava, 2009
141 strán, náklad 300 výtlačkov
Neprešlo jazykovou úpravou
ISBN 978-80-8072-095-7