Dvadsiaty siedmy ročník Olympiády v informatike

SLOVENSKÁ INFORMATICKÁ SPOLOČNOSŤ

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

Na obálke ročenky je dlhý reťazec textu. Ak sa ho pokúsite prečítať, zistíte, že síce zmysel príliš nedáva, ale čítať sa dá a znie skoro ako slovenčina. Ide o výstup z tzv. Markovovského zdroja.

Podobný reťazec sa dá vyrobiť nasledovne: začneme s ľubovoľnými tromi znakmi. Nájdeme si v 26. ročenke OI jeden náhodný výskyt tejto trojice znakov a pozrieme sa, aký znak za nimi nasleduje. Ten pridáme aj na koniec nášho reťazca. A takto pokračujeme dokola, vždy pozrieme na doteraz posledné tri znaky reťazca a tie nám náhodným výberom určia, čo za ne pridáme. Všimnite si, že čím je v slovenčine daná kombinácia písmen častejšia, tým má väčšiu pravdepodobnosť, že ju použijeme.

Napríklad za „`krá`“ asi doplníme niektoré z písmen „`ľ`“, „`s`“, či „`t`“ (ale zrejme nie „`r`“, hoci je to inak časté písmeno). A ak by sme doplnili „`ľ`“, tak v nasledujúcom kroku za „`ráľ`“ náhodne doplníme buď medzeru alebo „`o`“,

Markovovské zdroje majú svoje využitie napr. v kompresii textu, kryptoanalýze, bioinformatike a mnohých ďalších oblastiach.

RNDr. Michal Forišek PhD., Slovenská komisia OI
Dvadsiaty siedmy ročník Olympiády v informatike
ISBN 978-80-8072-122-0

Obsah

O priebehu 27. ročníka Olympiády v informatike

V školskom roku 2011/12 na Slovensku prebehol už dvadsiaty siedmy ročník Olympiády v informatike (OI). Do súťaže sa zapojilo 71 žiakov v kategórii A (starší) a 57 v kategórii B (mladší).

Najlepších 27 riešiteľov kategórie A sa zúčastnilo celoštátneho kola, ktoré sa konalo v Rakoviciach, okres Piešťany. Úspešní riešitelia celoštátneho kola potom na tradičnom týždňovom výberovom sústredení bojovali o účasť na medzinárodných súťažiach a prípravných akciách. Celý ročník bol korunovaný milým úspechom, keď sa v septembri 2012 na Medzinárodnej olympiáde v informatike (IOI) podarilo Eduardovi Batmendijnovi získať pre Slovensko zlatú medailu.

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, PhD., podpredseda, KI FMFI UK, Bratislava
- RNDr. Andrej Blaho, PhD., FMFI UK, Bratislava
- Mgr. Ján Katrenič, PhD., PF UPJŠ, Košice
- Mgr. Juliana Šišková, PhD., FMFI UK, Bratislava
- PaedDr. Ivan Brodenec, KI FPV UMB, krajský predseda pre BB
- RNDr. Eva Hanulová, Gym. J. Hronca, krajská predsedkyňa pre BA
- RNDr. Rastislav Krivoš-Belluš, PhD.,
  Ústav informatiky PF UPJŠ, krajský predseda pre KE
- prof. Ing. Veronika Stoffová, CSc.,
  KI PF UJS, krajská predsedkyňa pre NR
- Mgr. Mária Majherová, PhD.,
  Katedra matematiky FHPV PU, krajská predsedkyňa pre PO
- Ing. Andrea Julény, KI FMech TnUAD, krajská predsedkyňa pre TN
- doc. RNDr. Mária Lucká, CSc.,
  KMI PdF TU, krajská predsedkyňa pre TT
- RNDr. Peter Varša, PhD., KI FRI ŽU, krajský predseda pre ZA

Zástupcom IUVENTY zabezpečujúcim organizačnú stránku súťaže bol PhDr. Peter Barát.

Tento ročník OI bol hlavne v znamení úspešnej medzinárodnej prípravy. Okrem tradičného česko-poľsko-slovenského prípravného sústredenia (CPSPC) sa niekoľkí naši súťažiaci mali možnosť zúčastniť prípravných sústredení v zime v Švajčiarsku a v lete (vďaka prispeniu Slovenskej informatickej spoločnosti) vo Švédsku. Veríme, že táto medzinárodná príprava prospela ako samotným súťažiacim v príprave, tak aj nám organizátorom pri našej snahe držať krok so svetovou špičkou.

Michal Forišek, podpredseda SK OI

Zadania domáceho kola kategórie A

A-I-1 Bezpečná planéta

Planéty vo vesmíre môžeme rozdeliť na tri druhy:
- *typ 0: neobyvateľné* planéty, na ktorých človek okamžite zahynie,
- *typ 1: nehostinné* planéty, na ktorých človek chvíľu prežije, ale nie dlhodobo,
- *typ 2: prívetivé* planéty, na ktorých človek môže spokojne žiť.

Všetky rasy žijúce vo vesmíre cestujú medzi planétami prostredníctvom siete **jednosmerných** teleportov.

Planétu voláme *bezpečná*, ak platí, že je typu 2, a takisto sú typu 2 všetky planéty, na ktoré sa z nej vieme dostať pomocou cestovania teleportami (možno aj viacerými po sebe). Ak teda vysadíte človeka na bezpečnej planéte, máte istotu, že všade, kam odtiaľ docestuje, môže spokojne žiť.

Planétu voláme *znesiteľná*, ak síce nie je bezpečná, ale dá sa z nej pomocou teleportov (možno viacerých po sebe) dostať na bezpečnú planétu bez toho, aby sme museli navštíviť planétu typu 0. Ak teda vysadíte človeka na znesiteľnej planéte a dáte mu vhodné inštrukcie ako cestovať, časom docestuje živý a zdravý na nejakú bezpečnú planétu. Všimnite si, že planéta typu 0 nemôže nikdy byť znesiteľná.

Súťažná úloha:

Daný je počet planét n, počet teleportov m, pre každú planétu jej typ a pre každý teleport odkiaľ kam vedie. Zistite, ktoré planéty sú znesiteľné a ktoré bezpečné.

Formát vstupu:

Prvý riadok vstupu obsahuje dve celé čísla n a m. Planéty si očíslujeme od 1 po n. Druhý riadok vstupu obsahuje n medzerami oddelených celých čísel z množiny $\{0, 1, 2\}$: postupne pre každé i typ planéty číslo i. Zvyšok vstupu tvorí m riadkov, každý z nich popisuje jeden teleport. Popis teleportu pozostáva z dvoch čísel planét: odkiaľ a kam vedie.

Vo vstupoch, za ktoré bude dokopy 5 bodov, bude $1 \leq n \leq 500$ a $0 \leq m \leq 10\,000$. V ostatných vstupoch bude $1 \leq n \leq 100\,000$ a $0 \leq m \leq 200\,000$.

Formát výstupu:

Vypíšte dva riadky. V prvom z nich reťazec „`bezpecne:"` a za ním postupnosť čísel bezpečných planét. Čísla vypíšte v usporiadanom poradí a pred každým z nich vypíšte práve jednu medzeru. V druhom riadku vypíšte reťazec „`znesitelne:"` a postupnosť čísel znesiteľných planét. Použite rovnaký formát ako v prvom riadku. (Ak je niektorá z postupností prázdna, bezprostredne za dvojbodkou má nasledovať koniec riadku.)

Príklad:

Vstup

```
7 8
2 1 2 2 2 2 0
1 2
2 3
3 4
4 5
5 3
2 6
6 7
7 6
```

Výstup

```
bezpecne: 3 4 5
znesitelne: 1 2
```

A-I-2 Naložená loď

Na planéty, ktoré ešte nemajú teleportové terminály, ich treba priviezť medziplanetárnou kozmickou loďou. Keďže ešte nik nevyvinul technológiu na automatické pilotovanie takejto lode, musí ísť s loďou aj posádka. A tá potrebuje jesť. Tvojou úlohou bude presne naplniť celý potravinový nákladný priestor lode balíčkami s jedlom.

Problém je v tom, že balíčky sa vyrábajú na Zemi, zatiaľ čo loď je na obežnej dráhe. Balíčky teda treba dostať zo Zeme na kozmickú loď. To sa robí tak, že sa zoberie niekoľko balíčkov, naložia sa do kapsuly, tú sa vystrelí zo Zeme na obežnú dráhu, a tam si ju už posádka lode odchytí.

Na Zemi majú k dispozícii n typov kapsúl. Kapsula i-teho typu má kapacitu presne a_i balíčkov – nezmestí sa do nej viac a zároveň ich kvôli správnemu vyváženiu a doletu nesmie byť ani menej. Kapsúl každého typu je k dispozícii dostatočne veľa.

Napíš program, ktorý načíta počet typov kapsúl a ich kapacity a vypočíta **najmenší počet** kapsúl potrebný na naloženie kozmickej lode. Súčet kapacít použitých kapsúl musí byť **presne rovný** kapacite k nákladného priestoru lode.

Formát vstupu:

Prvý riadok vstupu obsahuje celé číslo n. Druhý riadok obsahuje n celých čísel $a_1, \ldots, a_n$ oddelených medzerami. Tretí riadok obsahuje kapacitu nákladného priestoru lode k.

Môžete predpokladať, že $a_1 < \cdots < a_n$ a že $a_1 = 1$ (teda vždy existuje spôsob, ako presne celú loď naplniť).

- Vo vstupoch za 3 body bude platiť $n \leq 5$, $a_n \leq 2000$ a $k \leq 30$.
- Vo vstupoch za ďalšie 4 body bude platiť $n \leq 30$, $a_n \leq 2000$ a $k \leq 1\,000\,000$.
- Vo vstupoch za posledné 3 body bude platiť $n \leq 30$ a $k \leq 10^{18}$. Spomedzi týchto vstupov jeden bod bude za vstupy s $a_n \leq 100$, druhý za vstupy s $a_n \leq 300$ a tretí za vstupy s $a_n \leq 2000$.

Všimnite si, že na uloženie premennej k potrebujete použiť 64-bitovú premennú: `long long` v C/C++, `int64` v Pascale. Takéto veľké premenné budete tiež potrebovať na počty kapsúl v optimálnom riešení.

Formát výstupu:

V prvom riadku vypíšte najmenší počet kapsúl potrebný na presné naloženie našej kozmickej lode.

V druhom riadku vypíšte jedno ľubovoľné riešenie, ktoré použije tento počet kapsúl. Presnejšie, vypíšte n celých čísel $b_1, \ldots, b_n$ oddelených medzerami, pričom b_i je počet použitých kapsúl veľkosti a_i.

Príklady:

Vstup

```
4
1 10 100 1000
147
```

Výstup

```
12
7 4 1 0
```

Použijeme 1 kapsulu s kapacitou 100, 4 s kapacitou 10 a 7 s kapacitou 1. To je dokopy 1+4+7=12 kapsúl.

Vstup

```
3
1 8 10
16
```

Výstup

```
2
0 2 0
```

Optimálne riešenie je použiť 2 kapsuly s kapacitou 8.

A-I-3 Skladník

Ignác bol skladníkom. Pil alkoholické nápoje, používal vulgarizmy a v pracovnej dobe zväčša spal. Niet divu, že sa mu šéf už dlhé roky vyhrážal, že ho nahradí počítačom. Až jedného dňa tá chvíľa naozaj prišla: Šéf kúpil počítač, posadil ho na vrátnicu skladu a Ignáca prepustil. Tým dosiahol presne rovnaký stav ako doteraz, len počítaču už nemusel platiť žiadnu mzdu.

Netrvalo však dlho a šéf si uvedomil, že počítač dokonca môžu aj zapnúť a používať. Potrebujú však na to vhodný softvér.

Podúloha a) *(4 body)* Napíšte čo najefektívnejší program, ktorý bude spravovať inventár skladu. Na vstupe budú prichádzať informácie o tom, koľko kusov ktorého typu tovaru pribudlo alebo ubudlo. Po každom načítaní novej informácie by váš program mal vypísať dva údaje: celkový počet kusov objektov v sklade a počet rôznych typov objektov v sklade.

Podúloha b) *(6 bodov)* Napíšte ešte jeden program, ktorý tiež bude spravovať inventár skladu. Jeho vstup bude vyzerať rovnako ako vstup prvého programu. Po každom načítaní novej informácie by tento program mal vypísať dva údaje: názov typu veci, z ktorého je aktuálne v sklade najviac kusov, a tento počet kusov. (Ak je viac možných typov vecí, vypíšte ten, ktorý je prvý v abecednom poradí.)

Formát vstupu:

V každom riadku vstupu je najskôr jedno nenulové celé číslo δ_i a potom jeden reťazec s_i obsahujúci nanajvýš ℓ znakov anglickej abecedy. Význam tohto riadku je, že počet vecí typu s_i v sklade sa zmenil o δ_i.

Pri písaní programov môžete predpokladať, že $\ell \leq 50$, že počet typov vecí v sklade nikdy neprekročí 100 000 a že celkový počet kusov vecí v sklade nikdy neprekročí 10^{18} (a teda sa zmestí do bežnej celočíselnej premennej). Tiež môžete

predpokladať, že nikdy nedostanete inštrukciu, ktorá by počet vecí nejakého typu zmenila na záporný.

Príklad:

vstup

```
+3 koberec
+2 buldozer
+1 buldozer
+7 zeriav
-3 buldozer
-5 zeriav
```

výstup, podúloha a)

```
kusov: 3, typov: 1
kusov: 5, typov: 2
kusov: 6, typov: 2
kusov: 13, typov: 3
kusov: 10, typov: 2
kusov: 5, typov: 2
```

výstup, podúloha b)

```
koberec 3
koberec 3
buldozer 3
zeriav 7
zeriav 7
koberec 3
```

Hodnotenie:

Každá podúloha sa hodnotí samostatne.

Ako súčasť svojho riešenia odhadnite, ako **najdlhšie** (t.j. v najhoršom možnom prípade) môže každému z vašich programov trvať spracovanie n-tej inštrukcie. (Pri odhade môžete použiť premennú t pre aktuálny počet typov vecí v sklade, premennú k pre aktuálny počet kusov vecí v sklade a premennú ℓ pre maximálnu dĺžku názvu veci.) Táto časová zložitosť bude hlavným kritériom hodnotenia. Snažte sa teda, aby váš program každú informáciu zaručene spracoval rýchlo.

Pripomíname: Ak používate v programe netriviálne algoritmy alebo dátové štruktúry (napr. rôzne súčasti STL v C++), súčasťou popisu algoritmu musí byť dostatočný popis ich implementácie, prípadne popis implementácie príbuznej dátovej štruktúry s rovnakou časovou zložitosťou operácií.

A-I-4 Zlomkové programy

V tomto ročníku olympiády sa budeme v každom súťažnom kole stretávať so zlomkovými programmi. V študijnom texte uvedenom za zadaním tejto úlohy je popísané, ako tieto programy fungujú.

Súťažná úloha:

a) (5 bodov) Na vstupe je číslo n tvaru $2^x 3^y 5$, pričom $x, y \geq 0$.
Napíšte program, ktorý ho prerobí na číslo 5, ak $x = y$, resp. na číslo 7, ak $x \neq y$.

b) (5 bodov) Na vstupe je číslo n tvaru $2^x 3$, pričom $x \geq 0$.
Napíšte program, ktorý ho prerobí na číslo tvaru 2^y, kde $y = \lfloor x/2 \rfloor$.

Pred tým, než svoje riešenie odovzdáte, si ho môžete otestovať. Na adrese `http://oi.sk/zlomky/` nájdete interpreter zlomkových programov.

Študijný text

Zlomkové programy predstavujú jeden veľmi jednoduchý spôsob, ako počítať niektoré funkcie na prirodzených číslach. Samotný zlomkový program je veľmi jednoduchý: je to konečná postupnosť zlomkov, teda kladných racionálnych čísel $(z_1, \ldots, z_k)$.

Výpočet zlomkového programu prebieha v krokoch. Počas výpočtu si udržiavame jediné celé číslo, tzv. *aktuálnu hodnotu* a. Na začiatku výpočtu na vstupe n je $a = n$. Každý krok výpočtu vyzerá nasledovne: Nájdeme najmenšie i také, že $a \cdot z_i$ je celé číslo, a zmeníme aktuálnu hodnotu na $a \cdot z_i$. Ak také i neexistuje, výpočet končí.

Príklad 1. Ukážeme si program, ktorý pre vstup $n = 2^x$ (kde $x \geq 0$) vyrobí výstup 3^y, kde $y = x \bmod 3$.

Jedným takýmto programom je postupnosť $(1/8, 9/4, 3/2)$. Slovne si priebeh výpočtu tohto programu môžeme popísať nasledovne: kým sa to dá, zmenšuj x o 3. Keď sa to už nedá, máme v a číslo 1, 2, alebo 4, vyrobíme z neho teda 1, 3, alebo 9. Napríklad pre $n = 1024 = 2^{10}$ sa hodnota a bude meniť nasledovne: $2^{10} \xrightarrow{1} 2^7 \xrightarrow{1} 2^4 \xrightarrow{1} 2 \xrightarrow{3} 3$. (Číslo nad šípkou je poradovým číslom zlomku, ktorým sme a v danom kroku prenásobili.)

Všimnite si, že záleží na poradí zlomkov. Napríklad program $(9/4, 3/2, 1/8)$ by z čísla 2^x vyrobil číslo 3^x. Zlomok $1/8$ by sa pri výpočtoch tohto programu nikdy nepoužil. Iné vyhovujúce programy sú $(1/8, 3/2)$, $(3/2, 1/27)$ a $(1/27, 3/2)$. Rozmyslite si, prečo každý z nich tiež rieši zadanú úlohu.

Príklad 2. Čo urobí program $(2/2)$ na vstupe $n = 4$? A čo na vstupe $n = 7$?

Obľúbenou chybou je odpovedať, že na vstupe $n = 4$ sa tento program zacyklí, ale na vstupe $n = 7$ sa program zastaví, lebo 7 nie je deliteľné dvomi. Správna odpoveď ale je, že v **oboch** prípadoch bude program bežať do nekonečna. Zaujímajú nás totiž len hodnoty zlomkov, nie ich zápis. Zlomok $2/2$ predstavuje racionálne číslo 1, a $7 \cdot (2/2) = 7 \cdot 1$ je celé číslo.

Aby ste sa vo svojich riešeniach takto nepoplietli, odporúčame uvádzať všetky zlomky v základnom tvare. Pre takto zapísanú postupnosť zlomkov potom už platí, že v každom kroku hľadáme najmenšie i, pre ktoré menovateľ i-teho zlomku delí aktuálnu hodnotu.

Príklad 3. Ukážeme si program, ktorý pre vstup $n = 2^{x+1}$ (kde $x \geq 0$) vyrobí výstup 3^{x+2}.

Číslo na vstupe je určite párne a nie je deliteľné žiadnym prvočíslom iným ako 2. Použijeme prvočíslo 11 ako značku, že už nie sme na začiatku výpočtu. V prvom kroku teda prepíšeme číslo 2^{x+1} na $2^x 3^2 11$. Toto dosiahneme zlomkom $3^2 \cdot 11/2 = 99/2$.

Ak už vidíme v prvočíselnom rozklade aktuálnej hodnoty prvočíslo 11, znamená to, že prvý krok máme úspešne za sebou. Môžeme teda spokojne „meniť dvojky na trojky“. To vieme dosiahnuť napríklad postupnosťou zlomkov $(3 \cdot 13)/(2 \cdot 11)$ a $11/13$. (Všimnite si, že nestačí použiť jeden zlomok $33/22$. Ak vám nie je jasné, prečo, prečítajte si ešte raz príklad 2.)

Časom sa takto dostaneme k číslu $3^{x+2}11$. V tejto situácii už stačí len vydeliť aktuálnu hodnotu číslom 11 a môžeme skončiť.

Pozor treba dať na to, že vyššie popísané zlomky treba dať do správneho poradia: $(39/22, 11/13, 1/11, 99/2)$. V prvom kroku výpočtu sa zjavne použije posledný zlomok. Od tejto chvíle je aktuálna hodnota deliteľná číslom 11 alebo 13. Striedavo sa používajú prvé dva zlomky, až kým sa nedostaneme do situácie $a = 3^{x+2}11$. Vtedy sa už prvý ani druhý zlomok použiť nedá. Použije sa preto tretí, čím dosiahneme $a = 3^{x+2}$ a výpočet zjavne končí.

Poznámka. V riešeniach podobných tomu v príklade 3 nie je nutné čitatele a menovatele zlomkov roznásobovať. Pokojne uveďte svoje riešenie v tvare: $\big((3 \cdot 13)/(2 \cdot 11),\ 11/13,\ 1/11,\ (3^2 \cdot 11)/2\big)$.

Zadania domáceho kola kategórie B

B-I-1 Teploty

Janka pripravuje softvér pre letisko, ktorý bude vypisovať štatistiku o teplote vzduchu nad pristávacou dráhou. Pri dráhe má senzor, ktorý pravidelne pošle do jej počítača aktuálnu teplotu. Jej program musí vždy, keď príde nová hodnota, vypísať tri údaje: najmenšiu, priemernú a najväčšiu teplotu za posledných t meraní.

Napíšte program, ktorý bude tieto tri údaje počítať a vypisovať. Pre vás sme už pre jednoduchosť merania teploty pripravili vo forme vstupného súboru.

Formát vstupu:

V prvom riadku vstupného súboru sú dve celé čísla n a t: celkový počet meraní a dĺžka intervalu, ktorý nás zaujíma. V každom z nasledujúcich n riadkov je jedno reálne číslo s presne dvomi desatinnými miestami: nameraná teplota v stupňoch Celzia. Tieto záznamy sú uvedené v poradí, v akom boli namerané. Všetky teploty sú z rozsahu od $-80.00\,^{\circ}\mathrm{C}$ po $60.00\,^{\circ}\mathrm{C}$.

(Pre zaujímavosť: Vstupy `3.txt` a `5.txt` obsahujú merania približne zodpovedajúce reálnemu vývoju teplôt niekedy počas roku 2011. Teploty pre tieto vstupy boli merané zhruba raz za 10 sekúnd.)

Formát výstupu:

Pre každé meranie začínajúc t-tym (vtedy máme prvýkrát k dispozícii t hodnôt) a končiac posledným n-tým vypíšte jeden riadok a v ňom postupne tri údaje: najmenšie z posledných t meraní, priemer posledných t meraní a najväčšie z posledných t meraní.

Najmenšiu a najväčšiu teplotu vypisujte na aspoň dve desatinné miesta, priemernú na aspoň štyri. Drobné zaokrúhľovacie chyby budeme ignorovať, netrápte sa nimi. V Pascale môžete použiť na výpis priemernej teploty napríklad príkaz „`writeln(priemerna_teplota:0:4);`“, v C/C++ zase napríklad príkaz „`printf("%.4f",priemerna_teplota);`“.

Príklad:

Vstup

```
7 3
-23.15
-23.21
-23.23
-23.24
-23.12
-23.19
-23.20
```

Výstup

```
-23.23 -23.1967 -23.15
-23.24 -23.2267 -23.21
-23.24 -23.1967 -23.12
-23.24 -23.1833 -23.12
-23.20 -23.1700 -23.12
```

Prvá priemerná hodnota (−23.1967) je priemerom prvých troch teplôt zo vstupu (−23.15, −23.21 a −23.23). Druhá priemerná hodnota je priemerom druhej, tretej a štvrtej nameranej teploty. A tak ďalej.

B-I-2 Sudoku

6	2		5	3			1	4
			2		8			
		4	6		1	2		9
8			9		5			7
	4	6	7		2	3	9	
1	9		4		3			5
		5	8		4	1		
			1		6			
9	8			5			4	2

zadanie sudoku

6	2	8	5	3	9	7	1	4
7	1	9	2	4	8	5	3	6
3	5	4	6	7	1	2	8	9
8	3	2	9	1	5	4	6	7
5	4	6	7	8	2	3	9	1
1	9	7	4	6	3	8	2	5
2	6	5	8	9	4	1	7	3
4	7	3	1	2	6	9	5	8
9	8	1	3	5	7	6	4	2

jeho riešenie

Sudoku už dnes pozná snáď každý. Teda okrem Kleofáša, ktorý posledné tri roky prežil v lese s vlkmi. Zadaním sudoku je štvorec 9×9, v ktorého niektorých políčkach sú cifry od 1 po 9. Úlohou riešiteľa je vyplniť všetky zvyšné políčka, a to tak, aby sa v každom riadku, v každom stĺpci aj v každom zvýraznenom štvorci 3×3 každá cifra vyskytovala práve raz. Vpravo vidíte príklad zadania a správneho riešenia. Kleofáš sa začal učiť, ako riešiť sudoku. Už objavil tri pravidlá. Všetky si ukážeme na zadaní na obrázku hore.

- *Prvé pravidlo:* Pozrime sa na štvrtý stĺpec.
  Je v ňom len jedno voľné políčko. Tam musí byť cifra 3.
- *Druhé pravidlo:* Všimnime si políčko v spodnom riadku v šiestom stĺpci.
  Jediná cifra, ktorá tam ešte môže byť, je cifra 7.

- *Tretie pravidlo:* Skúsme do siedmeho riadku umiestniť cifru 9.
  Musí byť v piatom stĺpci, nikam inam ju už dať nesmieme.

Každé pravidlo sa dá použiť na ľubovoľný riadok, stĺpec, či zvýraznený štvorec 3×3. V treťom pravidle môžeme skúšať umiestniť ľubovoľnú cifru.

Napíšte program, ktorý bude riešiť sudoku. Váš program to samozrejme môže robiť ľubovoľným spôsobom, ale na plný počet bodov stačí, ak bude vedieť správne používať Kleofášove tri pravidlá.

Formát vstupu a výstupu:

V prvom riadku vstupného súboru je celé číslo t – počet zadaní sudoku, ktoré musíte všetky vyriešiť. Každé zadanie je popísané 9 riadkami. V každom riadku je 9 znakov. Každý znak je cifra od 1 do 9, alebo bodka predstavujúca prázdne políčko. Každé zadanie má jednoznačné riešenie.

Do výstupného súboru vypíšte zaradom riešenia všetkých zadaní.

Na vyriešenie súboru `1.txt` stačí používať prvé pravidlo na riadky a stĺpce.
Na vyriešenie súboru `2.txt` stačí používať prvé pravidlo na riadky, stĺpce a zvýraznené štvorce 3×3.
Na vyriešenie súboru `3.txt` stačí používať prvé dve pravidlá na riadky a stĺpce.
Na vyriešenie súboru `4.txt` stačí používať všetky tri pravidlá na riadky a stĺpce.
Na vyriešenie súboru `5.txt` stačí používať všetky tri pravidlá na riadky, stĺpce a zvýraznené štvorce 3×3.

Príklad:

Vstup

```
1
9421.83.6
8..94...2
..5.6.4.8
.18479625
5.......3
.6432518.
..1.3.2..
457.9...1
2..5.7964
```

Výstup

```
942158376
876943512
135762498
318479625
529681743
764325189
691834257
457296831
283517964
```

Na vyriešenie tohto zadania sudoku stačí používať prvé dve pravidlá na riadky a stĺpce.

B-I-3 Uhádni deň

„Kedy máš narodeniny?“ opýtala sa Anička.
„V máji,“ odpovedal Miško, „ale presný deň musíš uhádnuť!“
„Mal si tento rok narodeniny v utorok?“
„Áno.“
„Je to neskôr ako dvanásteho?“
„Nie.“
„A je to jednociferné číslo?“
„Nie.“
„Tak potom už viem! Desiateho!“ uhádla Anička.

Súťažná úloha:

Podúloha a) *(3 body)*

Anička deň Miškovych narodenín uhádla už po tretej otázke. Ale to len preto, že mala šťastie. Ak by mal Miško narodeniny inokedy, mohlo sa jej stať, že by musela položiť otázok oveľa viac.

Nájdite nejakú stratégiu, ako sa pýtať otázky tak, aby ste vedeli *zaručiť*, že hľadaný deň uhádnete po najviac 5 otázkach. Každá otázka musí byť taká, aby na ňu Miško vedel odpovedať „áno“ alebo „nie“.

(Stratégiu stačí popísať slovne, napr.: „Najskôr sa ho opýtam, či mal tento rok narodeniny cez víkend. Ak áno, tak druhá otázka bude ... a ak nie, tak druhá otázka bude ...“)

Podúloha b) *(4 body)*

Aj Anička chce, aby Miško uhádol, kedy má narodeniny. Prezradila mu, že je to v decembri. Jej hra však má trochu iné pravidlá: Miško sa nemôže otázky pýtať postupne. Musí ich najskôr všetky napísať na papier, a až potom mu Anička všetky zodpovie. (Opäť, každá otázka musí mať odpoveď „áno“ alebo „nie“.)

Poraďte Miškovi, aké otázky má napísať na papier a ako potom podľa Aničkiných odpovedí určí deň jej narodenín. Použite čo najmenej otázok.

Podúloha c) *(3 body)*

Nájdite optimálne riešenie predchádzajúcej podúlohy a dokážte, že je optimálne. Inými slovami, zdôvodnite, že keby Miško položil menej otázok, tak nemôže mať istotu, že deň Aničkiných narodenín uhádne.

B-I-4 Čierne štvorce

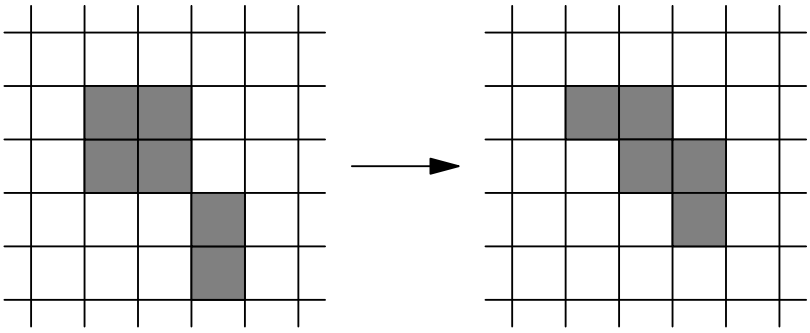
Keď Peťko pozeral priamy prenos z parlamentu, všimol si jedného poslanca. Ten väčšinu času spal. Len keď sa blížilo hlasovanie, zdvihol hlavu, opýtal sa kamaráta naľavo, kamaráta pred sebou a potom podľa toho zahlasoval. A spal ďalej.

Keď sa tak nad tým Peťko zamyslel, napadla mu zákerná otázka: čo by sa stalo, keby tak fungoval nie jeden poslanec, ale úplne všetci? A keďže to je otázka veľmi zložitá, zjednodušil si ju do nasledujúcej podoby:

Predstavme si nekonečnú štvorcovú sieť. Na začiatku sú skoro všetky štvorce, ktoré ju tvoria, biele – to sú poslanci, ktorí sú proti aktuálnemu návrhu. Len n štvorcov je čiernych – to sú poslanci, ktorí sú za.

Raz za minútu prebehne všeobecná diskusia. V nej si každý poslanec k svojmu názoru zistí ešte dva ďalšie: názor poslanca naľavo a názor poslanca pred ním. Ten názor, ktorý má medzi týmito troma väčšinu, bude náš poslanec zastávať počas nasledujúcej minúty.

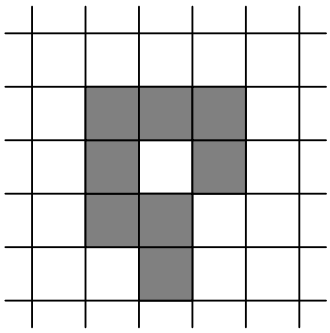
To isté v reči farebných štvorcov: novú farbu štvorca zistíme tak, že sa pozrieme na staré farby jeho, štvorca naľavo od neho a štvorca pod ním. Ako jeho novú farbu vyberieme tú, ktorú vidíme aspoň dvakrát. Tieto zmeny sa udejú naraz pre všetky štvorce. Na obrázku vidíte príklad toho, ako sa po jednej minúte zmení situácia.



Súťažná úloha:

Podúloha a) *(2 body)*

Na obrázku je jedna možná začiatočná situácia. Nakreslite, ako bude vyzerať o 1, 2, 3, 4 a 5 minút. (Môžete si buď pomôcť programom, alebo odpovede zostrojiť ručne.)



Podúloha b) *(2 body)*

V príklade v zadaní sa počet čiernych štvorcov zmenšil zo 6 na 5.

Nájdite jedno začiatočné rozmiestnenie čiernych štvorcov, pre ktoré sa po prvej minúte počet čiernych štvorcov nezmenší. (Čiernych štvorcov musí byť konečne veľa a nesmie ich byť nula.)

Podúloha c) *(2 body)*

Nájdite jedno rozmiestnenie 7 čiernych štvorcov také, že ešte aj po 6 minútach bude aspoň jeden štvorec čierny.

Podúloha d) *(4 body)*

Hovoríme, že čierne štvorce *držia pokope*, ak sa z každého na každý dá prejsť tak, že ideme len po čiernych štvorcoch a v každom kroku sa pohneme o políčko jedným z ôsmich základných smerov – ako kráľ v šachu. (Teda stačí, aby štvorce po kope držali rohom, ako na prvom obrázku vľavo.)

Nech n a k sú prirodzené čísla také, že platí $k \leq n$. Pre ktoré dvojice (n, k) sa dá rozmiestniť presne n čiernych štvorcov tak, aby držali pokope a zároveň platilo, že po k minútach budú prvýkrát všetky štvorce biele?

Zadania krajského kola kategórie A

A-II-1 Bezpečné medziplanetárne cestovanie

Planéty vo vesmíre môžeme rozdeliť na tri druhy:
- *typ 0: neobývateľné* planéty, na ktorých človek okamžite zahynie,
- *typ 1: nehostinné* planéty, na ktorých človek chvíľu prežije, ale nie dlhodobo,
- *typ 2: prívetivé* planéty, na ktorých človek môže spokojne žiť.

Všetky rasy žijúce vo vesmíre cestujú medzi planétami prostredníctvom siete **obojsmerných** teleportov. (Jednosmerné teleporty zrušili po tom, ako sa školský výlet z planéty QYX nevedel dva roky vrátiť späť domov.)

Postupnosť na seba nadväzujúcich teleportov voláme *cesta*. Cesta je pre ľudí *bezpečná*, ak neprechádza cez žiadnu planétu typu 0. Ak z nehostinnej planéty p vedie bezpečná cesta na nejakú prívetivú planétu, hovoríme, že z planéty p sa *človek vie zachrániť*.

Teleport je *kritický*, ak by jeho porucha spôsobila, že sa zmenší počet planét, z ktorých sa človek vie zachrániť. Inými slovami, teleport t je kritický, ak existuje planéta typu 1, z ktorej sa vieme dostať bezpečnou cestou na (aspoň jednu) planétu typu 2, ale pri každej takejto ceste musíme použiť teleport t.

Súťažná úloha:

V prvom riadku vstupu je počet planét n a počet teleportov m. Planéty majú čísla od 1 po n. V druhom riadku je pre každú planétu zadaný jej typ. Následne je pre každý teleport zadaná dvojica planét, ktoré spája. Medzi každou dvojicou planét vedie najviac jeden teleport.

Nájdite a vypíšte všetky kritické teleporty.

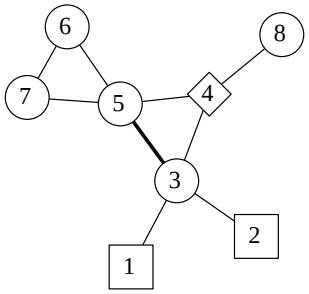
Príklad:

Vstup

```
8 9
2 2 1 0 1 1 1 1
1 3
2 3
3 4
3 5
4 5
5 6
6 7
5 7
4 8
```

Výstup

```
3 5
```



Vstup je znázornený na obrázku napravo.

Planéty typu 0, 1 a 2 sú zakreslené ako kosoštvorec, kruhy a štvorce.

Človek sa vie zachrániť z nehostinných planét 3, 5, 6 a 7. Ak by sa ale pokazil teleport medzi planétami 3 a 5, už by sa nedalo zachrániť z planét 5, 6 a 7. Tento teleport je teda kritický. Na obrázku je hrubou čiarou.

Žiaden iný teleport už kritický nie je. Špeciálne si všimnite, že nám neprekáža pokazenie teleportu medzi planétami 4 a 8. Z planéty 8 sa tak či tak zachrániť nedalo.

Hodnotenie:

- Plných 10 bodov môže získať riešenie, ktoré zvládne efektívne vyriešiť aj vstup so stotisíc planétami a miliónmi teleportov.
- Až 7 bodov sa dá získať za riešenie, ktoré zvládne efektívne vyriešiť vstup s tisíc planétami a pár tisícmi teleportov.
- Každé korektné riešenie, bez ohľadu na to, ako pomalé bude, môže získať aspoň 4 body.

A-II-2 Vlámačka à la Banach-Tarski

Po krátkom pobyte na slobode sa šialený robot Roberto opäť vydal na zločinecké chodníčky. Vlámal sa do laboratória profesora Farnswortha a chce si odtiaľ odniesť v obrovskom vreci profesorovu zbierku modelov vesmírnych lodí, aby ju speňažil na čiernom trhu. Ako však práve zistil, zbierka je natoľko rozsiahla, že ju celú nedokáže uniesť.

Okrem rôznych neužitočných vynálezov sa v laboratóriu nachádza aj duplikátor modelov vesmírnych lodí. Tento stroj dokáže vyrobiť presnú kópiu ľubovoľného modelu, no potrebuje na to istý čas. Vzniknutá kópia je na nerozoznanie od originálu – má rovnakú hmotnosť aj cenu na čiernom trhu.

Roberto sa bojí odhalenia, preto sa ponáhľa a nechce použiť duplikátor viac ako jedenkrát. Poraďte mu, či a ktorý model vesmírnej lode má zduplikovať, aby vedel ukradnúť modely s čo najvyššou celkovou cenou.

Súťažná úloha:

V prvých dvoch riadkoch vstupu je zadaná Robertova nosnosť m a počet modelov n. Súčet hmotností modelov, ktoré Roberto vynesie, nesmie presiahnuť m.

Nasleduje n riadkov, každý obsahuje popis jedného modelu vesmírnej lode – jeho hmotnosť w_i a cenu c_i. Modely sú očíslované od 1 po n v poradí, v akom sú na vstupe.

Môžete predpokladať, že hmotnosti modelov aj Robertova nosnosť sú **rozumne malé kladné celé čísla**. (Presnejšie údaje nájdete na konci zadania v časti Hodnotenie.)

Vypíšte číslo modelu, ktorý má Roberto zduplikovať, prípadne správu, že nemá zduplikovať žiaden model. Ak existuje viacero možností vedúcich k optimálnemu riešeniu, vypíšte ľubovoľnú z nich.

Následne vypíšte najvyššiu možnú celkovú cenu modelov, ktoré potom dokáže Roberto naraz vyniesť.

Príklady:

Vstup

```
5
2
2 1
4 3
```

Výstup

```
nezduplikuj nic
najlepsia cena 3
```

Nezáleží na tom, či niečo zduplikujeme, aj tak si vezmeme iba jeden kus – model s váhou 4 a cenou 3.

Vstup

```
5
4
4 1
1 3
3 5
2 5
```

Výstup

```
zduplikuj model 4
najlepsia cena 13
```

Zduplikujeme model $(2, 5)$ *a potom si vezmeme modely* $(1, 3)$, $(2, 5)$ *a* $(2, 5)$. *Ich celková hmotnosť je 1+2+2=5, čo ešte Roberto unesie, a ich celková cena je 3+5+5=13.*

Hodnotenie:

- Plných 10 bodov môže získať riešenie, ktoré zvládne efektívne vyriešiť aj vstup s tisíc modelmi a Robertovou nosnosťou desaťtisíc.
- Až 8 bodov sa dá získať za riešenie, ktoré zvládne efektívne vyriešiť vstup so sto modelmi a Robertovou nosnosťou tisíc.
- Jedným bodom môže byť zohľadnená aj pamäťová zložitosť vášho riešenia.

Pozor! Sústreďte sa na to, aby vaše riešenie bolo v prvom rade úplne korektné a vždy našlo úplne najlepšiu možnú celkovú cenu. Riešenia, ktoré majú principiálne nesprávny algoritmus, ktorý občas najlepšiu cenu nenájde, budú bodované veľmi prísne!

A-II-3 Parazity

„Za všetko môžu parazity!“ hlásala titulná strana Galaktického biologického spravodajského týždenníka (GBST). Vedcom sa totiž podarilo zistiť, že rôzne parazity vedia ovplyvňovať myslenie svojich hostiteľov vo svoj prospech a dokonca sú hnacím motorom evolúcie. Ďalším prekvapením bolo, že sa parazitom páčia špeciálne typy DNA.

DNA je postupnosť tzv. dusíkatých báz. U pozemských organizmov sú štyri (adenín, cytozín, guanín a tymín), ale mimozemské organizmy ich môžu mať viac. Všetky možné dusíkaté bázy vo vesmíre si očíslujeme od 1 po n. Následne môžeme každú DNA reprezentovať ako postupnosť celých čísel z tohto rozsahu.

Parazitom sa najviac páčia také súvislé úseky DNA, ktoré obsahujú (aspoň raz) každú z n možných dusíkatých báz. Komora Slovenských Parazitológov (KSP) sa teraz rozhodla preskúmať DNA rôznych vesmírnych organizmov a zistiť, ktorá z nich sa ako veľmi parazitom páči.

Súťažná úloha:

V prvom riadku vstupu je zadaný počet dusíkatých báz n a dĺžka skúmanej DNA d. V druhom riadku je popis DNA: postupnosť d celých čísel. Každé z týchto čísel je z rozsahu od 1 po n. Vypočítajte, koľko je v zadanej DNA súvislých úsekov, ktoré obsahujú každú dusíkatú bázu aspoň raz.

Príklady:

Vstup	Výstup
1 4 1 1 1 1	10

Každý úsek je jednoznačne určený miestami, kde začína a končí. Aj ak rôzne úseky obsahujú tú istú postupnosť dusíkatých báz, zarátame každý z nich.

Vstup	Výstup
3 5 1 2 1 3 2	5

Sú to nasledujúce úseky: (1,2,1,3), (1,2,1,3,2), (2,1,3), (2,1,3,2) a (1,3,2).

Vstup	Výstup
4 5 1 2 4 4 2	0

V danej postupnosti sa nenachádza báza 3, preto neexistuje žiaden úsek obsahujúci každú bázu aspoň raz.

Hodnotenie:

- Optimálne riešenie, za ktoré udelíme 10 bodov, zvládne efektívne vyriešiť aj vstup s miliónom rôznych dusíkatých báz a DNA dĺžky niekoľko miliónov.
- Až 8 bodov sa dá získať za riešenie fungujúce pre ľudí ($n = 4$), ak zvládne efektívne vyriešiť aj vstup s DNA dĺžky niekoľko miliónov.
- Medzi 6 a 8 bodmi môžu získať riešenia, ktoré si za sekundu poradia so stotisícprvkovou DNA – podľa toho, ako veľké n zvládajú.
- Každé korektné riešenie, bez ohľadu na to, ako pomalé bude, môže získať aspoň 3 body.

A-II-4 Zlomkové programy, maximum a sčítanie

Študijný text k tejto úlohe nájdete na strane 10 tejto ročenky.

Súťažná úloha:

a) (3 body) Na vstupe je číslo n tvaru $2^x 3^y$, pričom $x, y \geq 0$. Napíšte program, ktorý ho prerobí na číslo $5^{\max(x,y)}$.

Teda napríklad:
- z čísla $n = 144 = 2^4 3^2$ by mal vyrobiť číslo $5^4 = 625$,
- z čísla $n = 729 = 2^0 3^6$ by mal vyrobiť číslo $5^6 = 15625$.

b) (7 bodov) Na vstupe je číslo n tvaru $2^x 3^y 7$, pričom $x, y \geq 0$. Napíšte program, ktorý ho prerobí na číslo $2^x 3^y 5^{x+y}$.

Teda napríklad:
- z čísla $n = 1008 = 2^4 3^2 7$ by mal vyrobiť číslo $2^4 3^2 5^6$,
- z čísla $n = 5103 = 2^0 3^6 7$ by mal vyrobiť číslo $3^6 5^6$.

Hodnotenie:

Pri hodnotení úlohy sa bude mierne prihliadať aj na *časovú zložitosť* vašich zlomkových programov – teda na počet krokov výpočtu v závislosti od parametrov x a y. Ak bude váš program potrebovať *rádovo* viac krokov ako vzorový, môžete získať nanajvýš 2 body v prvej, resp. nanajvýš 6 bodov v druhej podúlohe.

Zadania krajského kola kategórie B

B-II-1 Letenka

Monika opäť raz cestuje kamsi na opačný koniec sveta. Potrebovala by si kúpiť letenku. Samozrejme, čo najlacnejšiu. Avšak prestupovať je otrava, preto je Monika ochotná prestupovať najviac dvakrát.

Súťažná úloha:

Napíšte program, ktorý načíta ceny jednotlivých letov a nájde Monike tri možné cesty: najlacnejší priamy let, najlacnejší let s najviac jedným prestupom a najlacnejší let s najviac dvomi prestupmi.

Formát vstupu:

Každé letisko má tzv. IATA kód, tvorený tromi veľkými písmenami. Napr. bratislavské letisko je `BTS`, košické je `KSC` a popradské je `TAT`. V prvom riadku vstupného súboru sú dva názvy letísk: odkiaľ a kam Monika cestuje.

V druhom riadku vstupného súboru je číslo ℓ: počet priamych letov, na ktoré si Monika môže kúpiť lístok.

Nasleduje ℓ riadkov, každý z nich popisuje jeden let. Popis letu obsahuje dva kódy letísk (odkiaľ a kam letí) a cenu letenky v eurách (celé číslo od 1 do 10 000).

Lety sú jednosmerné, teda napr. letom „`BTS KSC 47`“ sa nedá dostať z Košíc do Bratislavy. Pre každý let bude platiť, že koncové letisko je rôzne od začiatočného. Pre každú dvojicu letísk môže existovať viacero letov z prvého na druhé z nich. Tieto lety môžu mať navzájom rôzne ceny.

Nech n je počet rôznych letísk, ktoré sa aspoň raz vyskytnú na vstupe. Ako najväčší možný vstup si môžete predstaviť vstup, v ktorom $n = 10\,000$ a $\ell = 1\,000\,000$. Ak to pri svojom riešení potrebujete, môžete teda predpokladať, že rádovo n^2 celých čísel sa do pamäte ešte zmestí.

Formát výstupu:

Vypíšte presne tri riadky. V prvom riadku uveďte cenu najlacnejšieho priameho letu, v druhom cenu najlacnejšieho letu s max. 1 prestupom a v treťom cenu najlacnejšieho letu s max. 2 prestupmi. Ak neexistuje žiadne riešenie, namiesto ceny letu vypíšte reťazec „`neexistuje`“.

Príklady:

Vstup	Výstup
VIE ZRH 2 VIE MUC 23 MUC ZRH 24	neexistuje 47 47

Priamy let neexistuje.

Najlacnejší let z Viedne (VIE) do Zurichu (ZRH) s najviac jedným prestupom je cez Mníchov (MUC) za 23+24 = 47.

Tento let je zároveň najlacnejším aj vtedy, ak dovolíme dva prestupy – lebo aj tak nič iné nemáme na výber.

Vstup	Výstup
ZRH VIE 8 ZRH VIE 1200 ZRH VIE 470 VIE ZRH 10 ZRH MUC 120 ZRH FRA 220 MUC FRA 30 MUC VIE 220 FRA VIE 110	470 330 260

B-II-2 Benátky

Hlavnú ulicu v Benátkach tvorí n domov stojacich jeden vedľa druhého v rade. Rôzne domy môžu mať rôzne počty poschodí. Domy v Benátkach sú stavané tak, že sa medzi nimi dá na ľubovoľnom poschodí prechádzať. To aby si domorodci nezamočili nohy, keď idú niekam na opačný koniec ulice. Aj po strechách budov sa dá chodiť.

Problém je ale v tom, že Benátky sa potápajú. Historické zdroje uvádzajú čísla medzi 0.4 a 2.4 mm/rok. Jeden však nikdy nevie, či zajtra nezačnú klesať rádovo rýchlejšie. No a časom to povedie k tomu, že niektoré domy skončia celé pod vodou a Hlavná ulica sa rozpadne na niekoľko ostrovov. A to bude raj pre gondolierov, ktorí budú potom všetkých obyvateľov medzi ostrovami prevážať.

Súťažná úloha:

Na vstupe dostanete počet domov n a pre každý dom počet poschodí, ktoré má (celé číslo od 1 po n).

Napíšte čo najefektívnejší program, ktorý zistí všetky nasledujúce údaje:

- Na koľko ostrovov bude rozdelená ulica, keď voda zatopí prvé poschodia všetkých domov?
- Na koľko ostrovov bude rozdelená ulica, keď voda zatopí druhé poschodia všetkých domov?
- ...
- Na koľko ostrovov bude rozdelená ulica, keď voda zatopí n-té poschodia všetkých domov?

(Ak voda práve zatopila k poschodí každého domu, strechy k-poschodových domov sú ešte suché.)

Pozor! Existujú rozlične efektívne riešenia tejto úlohy. Prvé, ktoré nájdete, nemusí nutne byť to najlepšie.

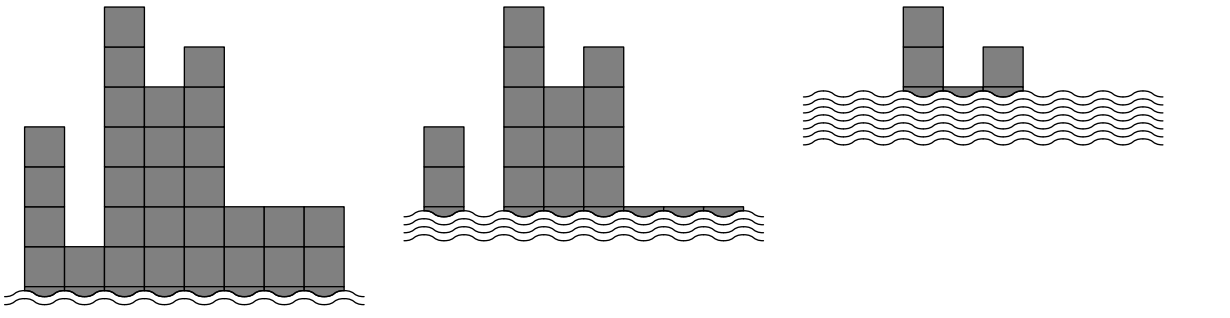
Formát vstupu a výstupu:

V prvom riadku vstupu je celé číslo n. V druhom riadku sú medzerami oddelené počty poschodí v domoch na Hlavnej ulici, v poradí v akom po sebe na ulici nasledujú. Výška každej budovy je celé číslo od 1 po n.

Na výstup vypíšte n čísel, pričom i-te z nich má byť počet ostrovov vo chvíli, kedy voda zatopí spodných i poschodí každého domu.

Príklad:

Vstup	**Výstup**
8 4 1 7 5 6 2 2 2	1 2 2 2 1 2 1 0



Na obrázku vľavo je situácia v súčasnosti, ktorú popisujú vstupné dáta z príkladu. Celá ulica je zatiaľ jedným veľkým ostrovom.

Na obrázku uprostred je situácia, keď už voda zatopila dve poschodia. Vidíme, že ulicu teraz tvoria 2 ostrovy, preto je druhé číslo vo výstupe 2.

Na obrázku vpravo je situácia po zatopení piatich poschodí. Opäť máme len 1 ostrov, preto je piate číslo vo výstupe 1.

B-II-3 Aničkine darčeky

Anička mala nedávno narodeniny. Keďže Miško uhádol presný dátum, zorganizoval jej veľkú párty a pozval všetkých jej kamarátov. Tí sú všetci slušne vychovaní a nezabudli so sebou priniesť každý po jednom darčeku.

Keď už svitalo a odišiel aj posledný hosť, dojatá Anička si ešte raz poobzerala všetky tie nádherné darčeky. Každému z nich určila hodnotu a podľa nej ich usporiadala do dlhého radu, ktorý končil až kdesi v kuchyni.

Keď sa ale potom uložila spať, prisnilo sa jej, že príde veľká bieda. Aničku ale len tak niečo nezlomí. Rozhodla sa, že ak ozaj nastanú zlé časy a bude potrebovať určitý obnos peňazí, vyberie si jeden darček, ktorý predá. Zároveň jej ale nesmie byť ľúto, že stačilo predať aj niečo menej hodnotné.

Súťažná úloha:

Podúloha a) *(6 bodov)*

V pamäti už je načítané pole celých čísel A, v ktorom sú uložené ceny všetkých n darčekov. Toto pole je *usporiadané* od najmenšej ceny po najväčšiu.

Vašou úlohou je napísať *čo najefektívnejšiu funkciu*, ktorá dostane ako parametre pole A, počet darčekov n a sumu p, ktorú Anička potrebuje. Táto funkcia musí nájsť *najlacnejší* darček, ktorého cena je *väčšia alebo rovná* ako p. Návratovou hodnotou vašej funkcie by mala byť cena dotyčného darčeka, prípadne číslo -1, ak žiaden darček nie je dostatočne drahý.

Dobrá rada: Anička peniaze potrebuje čo najrýchlejšie. Keby vaša funkcia musela prechádzať celé pole, asi by dovtedy Anička umrela od hladu. Snažte sa nájsť riešenie, ktoré odpoveď vypočíta rádovo rýchlejšie.

Upozornenie: Naozaj uveďte presnú implementáciu v nejakom vami zvolenom programovacom jazyku. Riešenia obsahujúce len slovný popis algoritmu budú hodnotené veľmi prísne!

Príklad: Majme $n = 12$ darčekov.
Ich ceny nech sú $A = (2, 3, 3, 3, 5, 8, 9, 10, 23, 32, 47, 99)$.

- Pre hodnotu $p = 4$ by vaša funkcia mala vrátiť hodnotu 5.
- Pre hodnotu $p = 5$ by vaša funkcia mala tiež vrátiť hodnotu 5.
- Pre hodnotu $p = 17$ by vaša funkcia mala vrátiť hodnotu 23.
- No a pre hodnotu $p = 100$ by vaša funkcia mala vrátiť hodnotu -1 (nemáme darček s cenou ≥ 100)

Podúloha b) *(1 bod)*

Na koľko políčok poľa A sa (v najhoršom možnom prípade) pozrie tvoj algoritmus riešiaci podúlohu a)?
Nepotrebujeme presné číslo, stačí nám vedieť, ako rádovo závisí tento počet od čísla n.

Podúloha c) *(3 body)*

Na koľko políčok poľa A sa (v najhoršom možnom prípade) musí pozrieť úplne optimálny algoritmus?
Svoje tvrdenie dokážte.

Kostra riešenia podúlohy a):

V Pascale by riešenie podúlohy a) malo vyzerať približne nasledovne:

```
function najdi_darcek(var A : array of longint; n, p : longint) : longint;
var odpoved : longint;
    { sem napis deklaracie ostatnych premennych }
begin
    { ceny darcekov su A[0] az A[n-1] }
    { sem napis program, ktory vypocita cenu darceka do premennej odpoved }
    najdi_darcek := odpoved;
end;
```

A v C zase napríklad nasledovne:

```
int najdi_darcek(int *A, int n, int p) {
    // ceny darcekov su A[0] az A[n-1]
    // sem napis program, ktory vypocita cenu darceka do premennej odpoved
    return odpoved;
}
```

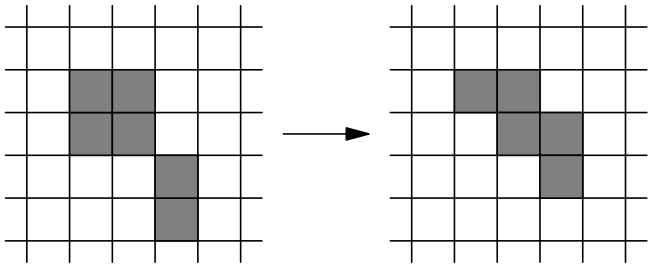
A v C++ trebárs takto:

```
int najdi_darcek(const vector<int> &A, int p) {
    // ceny darcekov su A[0] az A[ A.size()-1 ]
    // sem napis program, ktory vypocita cenu darceka do premennej odpoved
    return odpoved;
}
```

B-II-4 Čierne štvorce sú tu opäť

Rovnako ako v domácom kole, aj túto sadu úloh ukončíme jednou o prefarbovaní štvorcov. Najskôr si ale pripomenieme, ako naše prefarbovanie vlastne funguje.

Predstavme si nekonečnú štvorcovú sieť. Na začiatku sú skoro všetky štvorce, ktoré ju tvoria, biele, len n ich je čiernych. Každú minútu pre každý štvorec vypočítame jeho novú farbu: Pozrieme sa na staré farby jeho, štvorca naľavo od neho a štvorca pod ním. Ako jeho novú farbu vyberieme tú, ktorú vidíme aspoň dvakrát. Tieto zmeny sa udejú naraz pre všetky štvorce. Na obrázku je príklad toho, ako sa po jednej minúte zmení situácia.



Súťažná úloha:

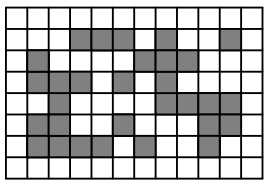
Podúloha a) *(2 body za hociktoré jedno rozmiestnenie / 3 body za obe)*

Nájdite jedno rozmiestnenie presne 7 čiernych štvorcov, pre ktoré platí: O 3 minúty ešte bude aspoň jeden štvorec čierny, ale o 4 minúty už budú úplne všetky štvorce v celej rovine biele.

Nájdite iné rozmiestnenie presne 7 čiernych štvorcov, pre ktoré platí: O 6 minút bude práve jeden štvorec v celej rovine čierny. Tento štvorec nesmie ležať na žiadnej z pozícií, kde boli čierne štvorce na začiatku.

Podúloha b) *(2 body)*

Marika nemá nekonečnú štvorcovú sieť, preto si túto hru skúša na papieri, kde si nakreslila obdĺžnikovú oblasť rozmerov 8×12. V tej si na začiatku vyfarbila nejaké čierne štvorce tak, aby žiaden z nich neležal na okraji. Príklad takéhoto vyfarbenia:



Môže sa Marike niekedy stať, že o pár minút bude niektorý zo štvorcov na okraji jej oblasti čierny?
(Marika počíta nové farby štvorcov na okraji tak, ako keby okolo jej papiera boli ešte ďalšie biele štvorce.)

Podúloha c) *(3 body)*

Vie Marika zafarbiť na čierno niektoré štvorce vo vnútri svojej oblasti tak, aby aj po 100 minútach menenia farieb mala vo svojej oblasti aspoň jeden čierny štvorec? Ak áno, nájdite jedno také začiatočné zafarbenie, ak nie, zdôvodnite, prečo to nejde.

Podúloha d) *(2 body)*

Na záver sa vráťme k nekonečnej štvorcovej sieti. Ak ste úspešne vyriešili všetky doterajšie podúlohy, ste pripravení na záverečnú otázku.

Nájdite nejaké (nie nutne najmenšie) prirodzené číslo k také, aby platilo: Ak na začiatku zafarbíme na čierno ľubovoľných 47 štvorcov, tak po k minútach už zaručene budú všetky štvorce v celej rovine biele.

Samozrejme, svoje tvrdenie patrične dokážte.

Zadania celoštátneho kola kategórie A

A-III-1 Odpoveď

Na úvod tohto zadania si pripomeňme, že *medián* postupnosti nepárnej dĺžky je jej prostredný prvok podľa veľkosti. Inými slovami, je to ten jej prvok, ktorý by bol presne uprostred, ak by sme danú postupnosť usporiadali. Napríklad mediánom postupnosti $(3, 1, 4, 1, 5)$ je číslo 3 a mediánom postupnosti $(9, 2, 6, 5, 3, 5, 8)$ je číslo 5.

Po dlhé roky sa vo vesmíre hľadala odpoveď na základnú otázku života, vesmíru a vôbec. Nakoniec túto odpoveď počítač na to určený aj vypočítal. Samotná odpoveď však bola pomerne prekvapivá: bolo ňou číslo 42. A až vtedy mnohí pochopili, že radšej mali chcieť poznať samotnú otázku.[1]

Ako z tejto kaše von? Jeden nápad mali členovia Klubu Starostlivých Pisárov (skrátene KSPáci): možno bolo v skutočnosti číslo 42 len časťou dlhšej odpovede, ktorú sa nepodarilo zachytiť. KSPáci našťastie po dlhé roky starostlivo zapisovali všetky dôležité čísla, na ktoré vo vesmíre narazili: od počtu zelených opíc až po veľkosť atómov v pravom zadnom kúte galaxie. V postupnosti, ktorú zapísali, sa číslo 42 nachádzalo práve raz. No a KSPáci by teraz chceli nájsť podrobnejšiu odpoveď na otázku života, vesmíru a vôbec. Nazdávajú sa, že ide o súvislú podpostupnosť zaznamenaných čísel, ktorej mediánom je práve naša stará známa hodnota 42. Existuje ale vôbec takáto podpostupnosť? A nie je ich náhodou viac?

Súťažná úloha:

Dané je celé číslo n a následne postupnosť tvorená n celými číslami. Práve jedno z týchto čísel je rovné 42.

(Ostatné čísla môžu byť veľké, ale zmestia sa do bežnej celočíselnej premennej. Môžete napr. predpokladať, že sú z rozsahu od -10^9 po 10^9.)

Napíšte program, ktorý spočíta, koľko má táto postupnosť *súvislých* podpostupností, ktoré majú *nepárnu dĺžku* a zároveň *medián rovný 42*.

[1] Ak vám tieto slová nič nehovoria, odporúčame vám knihu od Douglasa Adamsa s názvom Stopárov sprievodca po galaxii (v origináli The Hitchhiker's Guide to the Galaxy). S riešením tejto úlohy vám znalosť tejto knihy samozrejme nijako nepomôže.

Príklad:

Vstup

```
10
-5 13 42 88 37 41 43 100 50
72
```

Výstup

```
6
```

Je to týchto 6 podpostupností:
(42),
$(42, 88, 37)$,
$(42, 88, 37, 41, 43)$,
$(13, 42, 88)$,
$(13, 42, 88, 37, 41, 43, 100)$ *a*
$(-5, 13, 42, 88, 37, 41, 43, 100, 50)$.

Hodnotenie:

Úmyselne neuvádzame obmedzenie na veľkosť premennej n. Váš program dostane tým viac bodov, čím lepšia bude jeho časová zložitosť v závislosti od n.

A-III-2 U obchodníka s rozličným kradnutým tovarom

Vďaka vašim riešeniam z krajského kola sa šialenému robotovi Robertovi podarilo nepozorovane ukradnúť za plné vrece modelov vesmírnych lodí. Po tomto trúfalom kúsku sa Roberto náhlil na čierny trh, kde sa chystá toto vrece predať. Chce zaň dostať presne s peňazí. (Číslo s je kladné a celé.)

Na trhu sa platí pomocou n druhov mincí. (Každý druh mincí má kladnú celočíselnú hodnotu. Všetky tieto hodnoty sú pomerne malé.)

Samozrejme, platenie funguje ako v bežnom obchode: Platiaci nemusí zaplatiť presnú sumu. Môže zaplatiť aj viac a následne si nechať vydať sumu, o ktorú zaplatil viac ako mal. Aj Roberto, aj obchodník na trhu majú z každého druhu mince dostatočne veľa kusov.

Roberto sa i tentoraz ponáhľa, preto chce, aby celá transakcia prebehla čo najrýchlejšie. Požaduje, aby platba sumy s prebehla takým spôsobom, pri ktorom bude celkový počet kusov mincí použitých pri platení a vydávaní najmenší možný. Keďže to vôbec nie je ľahká úloha, Roberto sa obrátil na vás. Poraďte mu, lebo inak vás bonzne polícii, že ste mu minule pomáhali pri krádeži.

Súťažná úloha:

V prvom riadku vstupu je zadaná celočíselná suma s, ktorú chce Roberto dostať, a počet druhov mincí n. Druhý riadok obsahuje kladné celé čísla $c_1, c_2, \ldots, c_n$: hodnoty jednotlivých druhov mincí. Tieto hodnoty sú usporiadané vzostupne. Navyše platí $c_1 = 1$, aby sa každá suma určite dala zaplatiť.

Vypíšte dva riadky obsahujúce po n čísel. V prvom riadku vypíšte pre každý druh mince počet takých mincí, ktoré má použiť obchodník, v druhom riadku počty mincí, ktoré má Roberto obchodníkovi vydať. Poradie vypísaných hodnôt má zodpovedať poradiu zo vstupu. Ak existuje viacero riešení s optimálnym celkovým počtom použitých mincí, nájdite ľubovoľné z nich.

Príklady:

Vstup	**Výstup**
`8 4` `1 2 5 10`	`0 0 0 1` `0 1 0 0`

Suma 8 sa jednou mincou zjavne zaplatiť nedá, no dvomi to už ide: obchodník zaplatí 10 a Roberto mu vydá 2.

Vstup	**Výstup**
`10 3` `1 5 6`	`0 2 0` `0 0 0`

Hodnotenie:

Sústreďte sa na to, aby vaše riešenie bolo v prvom rade korektné – teda vždy našlo optimálny spôsob platenia. Riešenia, ktoré nebudú korektné, budú posudzované veľmi prísne! U väčšiny typov riešení je veľmi dôležitou súčasťou riešenia **dôkaz jeho správnosti**. Riešenia, v ktorých dôkaz nebude uvedený, môžu (podľa toho, nakoľko je/nie je zjavný) stratiť značnú časť bodov.

- 10 bodov sa dá získať za riešenie, ktoré zvládne efektívne vyriešiť vstupy s $n \leq 100$, $c_n \leq 1000$ a $s \leq 10^{18}$.
- Až 8 bodov sa dá získať za riešenie, ktoré zvládne efektívne vyriešiť vstupy s $n \leq 100$, $c_n \leq 1000$ a $s \leq 10^6$.
- Až 5 bodov sa dá získať za riešenie, ktoré zvládne efektívne vyriešiť vstupy s $n \leq 50$, $c_n \leq 1000$ a $s \leq 1000$.
- Každé korektné riešenie môže získať aspoň 3 body.

A-III-3 Zlomkové programy sa lúčia

Študijný text k tejto úlohe nájdete na strane 10 tejto ročenky.

Súťažná úloha:

Súťažná úloha sa skladá z troch podúloh, označených „a“, „b1“ a „b2“. U podúloh b1 a b2 sa očakáva, že si vyberiete a vyriešite jednu z nich.
Ak sa pokúsite vyriešiť aj podúlohu b1, aj podúlohu b2, započíta sa vám tá z nich, za ktorú dostanete viac bodov. Za ľubovoľné korektné riešenie podúlohy b2 budú udelené aspoň 4 body. Ak teda viete vyriešiť podúlohu b2, nie je potrebné uvádzať riešenie podúlohy b1.

a) (3 body) Na vstupe je číslo n tvaru $2^x 3^y 5^z$, pričom $x, y, z \geq 0$.
Napíšte program, ktorý ho prerobí na číslo $7^{\text{median}(x,y,z)}$.

Mediánom troch čísel je prostredné podľa veľkosti. Teda napríklad:
$\text{median}(2, 5, 8) = 5$, $\text{median}(8, 2, 5) = 5$ a $\text{median}(3, 3, 7) = 3$.
Váš program má teda napr. z čísla $n = 2^8 3^2 5^5$ vyrobiť číslo 7^5.

b1) (3 body) Na vstupe je číslo n tvaru $2^x \cdot 47$, pričom $x \geq 0$.
Napíšte program, ktorý ho prerobí na číslo $3^{(x^2)}$.

Teda napríklad:
- z čísla $n = 47 = 2^0 \cdot 47$ by mal vyrobiť číslo $3^0 = 1$,
- z čísla $n = 2^4 \cdot 47$ by mal vyrobiť číslo 3^{16}.

b2) (7 bodov) Na vstupe je číslo n tvaru 2^x, pričom $x \geq 0$.
Napíšte program, ktorý ho prerobí na číslo 3^y, kde $y = \lceil \sqrt{x} \rceil$.

Značenie $\lceil z \rceil$ označuje hornú celú časť čísla z: najmenšie celé číslo, ktoré je aspoň z. Napr. $\lceil 4.7 \rceil = 5$, $\lceil 47 \rceil = 47$, $\lceil \sqrt{16} \rceil = \lceil 4 \rceil = 4$ a $\lceil \sqrt{22} \rceil = 5$.

Váš program má teda napríklad z čísla $n = 2^{16}$ vyrobiť číslo 3^4 a z čísla $n = 2^{22}$ vyrobiť číslo 3^5.

Hodnotenie:

Pri hodnotení úlohy sa bude prihliadať aj na *časovú zložitosť* vašich zlomkových programov – teda na počet krokov výpočtu v závislosti od parametrov x, y a z. Ak bude váš program potrebovať *rádovo* viac krokov ako vzorový, môžete získať nanajvýš 2 body v podúlohe a, nanajvýš 2 body v podúlohe b1, resp. 4-6 bodov v podúlohe b2.

A-III-4 Tučný Santa

Onedlho bude apríl. A ako viete, v apríli už supermarkety začínajú predávať výrobky s vianočnou tematikou a na severnom póle začínajú mať plné ruky práce s prípravou Vianoc.

Momentálne Santa Claus a elfovia riešia dôležitý problém, ako sa dostať od kozuba (príp. okna či dverí, keď sa do miestnosti nedá inak dostať) ku vianočnému stromčeku. A to veru nie je vôbec ľahké, pretože ľudia majú teraz plné obývačky nábytku a harabúrd. Keďže Santa nedávno trocha pribral a získal tak krásny obdĺžnikový pôdorys, je naozaj zázrak, že sa ku stromčeku dokáže dostať. (Prezradíme vám tajomstvo: občas si pri tom pomôže vianočnou mágiou.)

Vašou úlohou bude pomôcť Santovi nájsť **najkratšiu** cestu k stromčeku.

Súťažná úloha:

Obývačka je obdĺžnikova miestnosť, ktorú si môžeme predstaviť ako mriežku tvorenú $r_1 \times s_1$ štvorcovými políčkami. Niektoré políčka sú prázdne, iné obsahujú prekážky. Políčka sú očíslované od $(0, 0)$ v ľavom hornom po $(r_1 - 1, s_1 - 1)$ v pravom dolnom rohu miestnosti.

Santu si môžeme predstaviť ako obdĺžnik rozmerov $r_2 \times s_2$. Santa je vždy umiestnený tak, aby zakrýval presne $r_2 \times s_2$ políčok miestnosti. Pohybovať sa vie len v štyroch smeroch rovnobežných s osami mriežky, a to vždy o 1 políčko. Nemôže sa otáčať, teda strana Santu, ktorá má dĺžku r_2, je vždy rovnobežná so stranou miestnosti, ktorá má dĺžku r_1.

Santa začína v ľavom hornom rohu, kde zaberá políčka od $(0, 0)$ po $(r_2 - 1, s_2 - 1)$. Aby mohol umiestniť darčeky pod stromček, potrebuje sa **na najmenší možný počet krokov** dostať do protiľahlého rohu, teda zaberať políčka od $(r_1 - r_2, s_1 - s_2)$ po $(r_1 - 1, s_1 - 1)$.

Počas svojej cesty nesmie Santa nikdy stúpiť na prekážku.

V niektorých situáciách (ktoré majú pre vás hodnotu 5 bodov) vie Santa používať mágiu. Pomocou mágie vie naraz presunúť nanajvýš m prekážok do inej dimenzie, a teda stáť aj na políčkach, kde tieto prekážky ležali. Mágiu vie Santa plynule presúvať z jednej prekážky na druhú, takže ak mu nájdete trasu, kde sa bude po každom kroku prekrývať s nanajvýš m prekážkami, bude po nej vedieť prejsť. Počas svojej cesty nesmie Santa nikdy opustiť obývačku, a to ani za pomoci mágie.

Formát vstupu:

V prvom riadku vstupu je päť celých čísel oddelených medzerami. Sú to rozmery obývačky r_1, s_1, rozmery Santu r_2, s_2 a množstvo dostupnej mágie m.

Nasleduje popis obývačky: r_1 riadkov, z ktorých každý obsahuje reťazec tvorený s_1 znakmi. Znak '`.`' (bodka) označuje voľné políčko, znak '`X`' (veľké iks) prekážku.

Môžete predpokladať, že na začiatku pod Santom nie je žiadna prekážka.

Formát výstupu:

Vypíšte jeden riadok a v ňom jeden reťazec znakov: postupnosť krokov, ktoré má Santa urobiť, aby sa dostal do cieľa svojej cesty. Na popis cesty použijeme svetové strany: presun o riadok dohora budeme označovať '`S`' (sever), presun o stĺpec doľava '`Z`' (západ); opačné smery budú '`J`' (juh) a '`V`' (východ).

Ak neexistuje žiadna postupnosť krokov, ktorá Santu dovedie do protiľahlého rohu, vypíšte jeden riadok a v ňom text „`Santa je chudak.`“ (bez úvodzoviek).

Ak existuje viacero vyhovujúcich postupností, vypíšte najkratšiu z nich. Ak stále existuje viacero vyhovujúcich postupností, môžete vypísať ľubovoľnú.

(Zdôrazňujeme, že vo vstupoch s $m > 0$ *nie je potrebné* hľadať postupnosť, kde Santa použije čo najmenej mágie celkovo, resp. čo najmenej mágie naraz. Jediné, čo je dôležité, je minimalizovať počet krokov, ktoré Santa spraví.)

Obmedzenia:

Testovacie vstupy sú rozdelené do 15 sád. Za každú sadu, ktorú váš program celú správne vyrieši, dostanete jeden bod. Vo všetkých testovacích vstupoch platí $1 \leq r_2 \leq r_1 \leq 2000$, $1 \leq s_2 \leq s_1 \leq 2000$ a $0 \leq m \leq r_2 s_2$. V žiadnom testovacom vstupe nebude súčasne platiť $r_1 = r_2$ aj $s_1 = s_2$. (Santa je vždy menší ako obývačka.) Na plný počet bodov by váš program mal vedieť správne vyriešiť úplne všetky takéto vstupy.

Nasledujú podrobné informácie o tom, ako vyzerajú testovacie vstupy.
Obzvlášť upozorňujeme, že ak váš program správne vyrieši všetky vstupy s $m = 0$, získa aspoň 10 bodov.

číslo sady	1-10	11	12-15
max. množstvo mágie m	0	1	$r_2 s_2$

číslo sady	1-2	3	4-6, 12	7-10, 11, 13-15
max. rozmery Santu $r_2 \times s_2$	1×1	10×10	$10 \times s_1$	$r_1 \times s_1$

číslo sady	1, 4	2, 5, 7	8, 13
max. rozmery izby $r_1 \times s_1$	20×20	200×200	1000×1000

číslo sady	9, 14	3, 6, 10-12, 15
max. rozmery izby $r_1 \times s_1$	1500×1500	2000×2000

Príklady:

Vstup

```
2 2 1 1 0
..
..
```

Výstup

```
VJ
```

Aj výstup „JV“ je správny.

Vstup

```
5 8 2 2 0
........
....XXXX
..X.....
........
.....X..
```

Výstup

```
JJJVVVSVVVJ
```

V tomto prípade ide o jedinú najkratšiu postupnosť pohybov.

Vstup

```
5 8 2 2 0
........
....XXXX
.XX.....
........
.....X..
```

Výstup

```
Santa je chudak.
```

Oproti predchádzajúcemu príkladu pribudla prekážka na $(2, 1)$. Tú Santa nemá ako obísť.

Vstup

```
5 8 2 2 1
........
....XXXX
.XXXX...
..X...X.
.....X..
```

Výstup

```
JJJVVVVSVVJ
```

Teraz tých prekážok pribudlo ešte viac, Santa však vie použiť mágiu. Všimnite si, že nevyhovuje cesta „JJJVVVVVV“ – totiž krok pred cieľom by Santa musel stáť naraz na dvoch prekážkach a na to nemá dosť mágie.

A-III-5 Ministerstvo

Ministerstvo pre boj s byrokraciou má n úradníkov. Tí sú označení číslami $1, 2, \ldots, n$. Číslo 1 má samotný minister, ostatné čísla nemajú žiaden špeciálny význam.

Každý úradník okrem ministra má svojho šéfa. Úradník a je nadriadeným úradníka b, ak je a buď priamo šéfom b, alebo je nadriadeným šéfa b. (Inými slovami, vtedy, ak je a šéfom b, alebo šéfom šéfa b, alebo ...) Ministerstvo má stromovú štruktúru, teda žiaden úradník nie je nadriadeným sebe samému.

Každý úradník vrátane ministra šéfuje oddeleniu ministerstva. Do tohto oddelenia patrí on sám a tiež všetci úradníci, ktorým je nadriadeným. (Občas sa stane, že oddelenie je tvorené len jedným človekom.)

Bojovníkom proti byrokracii treba pravidelne zvyšovať mzdu, aby dobre bojovali. To minister dosahoval nasledovnými príkazmi:

1. Ak je mzda úradnika u ostro menšia ako x, treba ju zvýšiť o y.
2. Ak je priemerná mzda oddelenia, ktorému šéfuje úradník u, ostro menšia ako x, treba každému úradníkovi z oddelenia zvýšiť mzdu o y.

Potom ale prišli voľby a s nimi aj nový minister. Mladý, neskúsený, plný ideálov. Inými slovami, fakt nevedel, ako to chodí. A preto začal vydávať aj príkazy tretieho typu:

3. Nájdi najmenšiu mzdu úradníka v oddelení, ktorému šéfuje úradník u. Túto mzdu nastav každému v tomto oddelení.

Čo čert nechcel, upratovačka vyhodila papier, na ktorom mala mzdová účtareň aktuálne mzdy. Jediné, čo zostalo, sú platy úradníkov na začiatku roka a zoznam ministerských príkazov v chronologickom poradí. Vašou úlohou je zistiť aktuálne platy.

Súťažná úloha:

Napíšte program, ktorý dostane popis ministerstva, začiatočný plat každého úradníka a postupnosť ministerských príkazov a vypíše plat každého úradníka po vykonaní všetkých príkazov.

Na získanie 12 bodov nie je nutné implementovať spracovanie príkazov nového ministra.

Formát vstupu:

V prvom riadku vstupu je číslo n, udávajúce počet úradníkov. Nasleduje n riadkov, pričom i-ty z nich popisuje úradníka číslo i. Každý z týchto riadkov obsahuje nasledujúce údaje: začiatočný plat z_i, počet priamych podriadených p_i a zoznam ich čísel. Všetky údaje sú oddelené medzerami.

(Môžete predpokladať, že popis hierarchie ministerstva je korektný. Každý úradník okrem ministra sa ako priamy podriadený vo vstupe vyskytne práve raz, a to tak, aby nevznikol žiaden cyklus.)

Nasleduje riadok s číslom q, udávajúcim počet ministerských príkazov. Zvyšok vstupu tvorí q riadkov, každý z nich popisuje jeden príkaz, v poradí, v akom boli vykonané. Popis príkazu má jeden z týchto tvarov:

- „1 u x y“: zodpovedá možnému zvýšeniu platu úradníka u
- „2 u x y“: zodpovedá možnému zvýšeniu platu celého oddelenia vedeného úradníkom u
- „3 u“: zodpovedá príkazu nového ministra

Formát výstupu:

Vypíšte n riadkov, v každom jedno celé číslo. Číslo v i-tom riadku má byť záverečný plat i-teho úradníka.

Obmedzenia:

Testovacie vstupy sú rozdelené do 15 sád. Za každú sadu, ktorú váš program celú správne vyrieši, dostanete jeden bod. Vo všetkých testovacích vstupoch platí pre začiatočné platy $0 \leq z_i \leq 1000$.
Vo všetkých testovacích vstupoch platí v každom príkaze: $1 \leq u \leq n$, $0 \leq x \leq 10^9$, $0 \leq y \leq 1000$.

Rôzne testovacie vstupy majú rôzny počet úradníkov n a rôzny počet príkazov q. Líšia sa tiež v nasledujúcich parametroch: hĺbke hierarchie h, počte vykonaní druhého príkazu v_2 a tom, či sa vo vstupe vyskytujú aj príkazy nového ministra.

(Hĺbka ministra je 0, jeho priami podriadení majú hĺbku 1, ich priami podradení 2, atď. Hĺbka hierarchie je maximum z hĺbok úradníkov, ktorí ju tvoria. Hodnota v_2 udáva len počet tých príkazov druhého typu, ktoré naozaj zmenia platy úradníkov.)

Nasledujúca tabuľka prehľadnou formou udáva, nanajvýš aké parametre majú ktoré testovacie vstupy.

číslo sady	1-3	4	5-6	7-11	12	13-15
n	1000	100000	100000	100000	100000	100000
q	1000	100000	100000	100000	100000	100000
h	1000	50	50	100000	100000	100000
v_2	1000	50	100000	50	100000	100000
príkazy „3 u“	nie	nie	nie	nie	nie	áno

Príklady:

Vstup

```
7
10 2 2 6
5 2 3 4
1 0
1 1 5
1 0
2 1 7
1 0
5
2 4 1 5
1 5 2 7
2 2 20 20
2 2 20 100
2 6 2 2
```

Výstup

```
10
25
21
21
28
4
3
```

Prvá operácia sa nevykoná, lebo priemer daného oddelenia je 1, *čo nie je menej ako* 1. *(Dané oddelenie tvoria úradníci 4 a 5. Obaja majú v tejto chvíli plat 1.) Druhá a tretia operácia sa vykonajú. Štvrtá operácia sa nevykoná. Piata sa vykoná, lebo priemer je* 1.5, *čo je menej ako* 2.

Vstup

```
3
10 2 2 3
4 0
7 0
1
3 1
```

Výstup

```
4
4
4
```

Minister má plat 10, jeho podriadení majú platy 4 a 7. Vykoná sa inštrukcia 3. typu pre ministra. Najnižší plat v jeho oddelení je 4, ten teda dostanú všetci.

Riešenia domáceho kola kategórie A

A-I-1 Bezpečná planéta

Všimnime si, že definíciu bezpečnej planéty môžeme preformulovať nasledovne: bezpečné sú tie planéty, z ktorých sa *nedá* docestovať na planétu typu 0 alebo 1. Inými slovami, planéta *nie je* bezpečná práve vtedy, ak sa z nej *dá* docestovať na planétu typu 0 alebo 1.

Množinu všetkých planét, ktoré *nie sú* bezpečné, teda vieme zostrojiť jednoduchým prehľadávaním zadanej siete teleportov. Začneme vo všetkých planétach typu 0 a 1 a odtiaľ pokračujeme proti smeru teleportov – ak planéta x nie je bezpečná a existuje teleport z y na x, tak ani y nie je bezpečná.

Množinu bezpečných planét následne získame ako doplnok množiny planét, ktoré nie sú bezpečné.

Následne zostrojíme všetky znesiteľné planéty. Na to stačí použiť druhé prehľadávanie, opäť proti smeru teleportov. Tentokrát začneme na všetkých bezpečných planétach a pokračovať smieme len cez planéty typu 1 a 2. Takto zjavne nájdeme všetky planéty, odkiaľ vie človek dosiahnuť bezpečnú planétu.

V našom programe používame v oboch prípadoch prehľadávanie do šírky. Časová aj pamäťová zložitosť nášho riešenia je zjavne lineárna od veľkosti vstupu, teda $\Theta(n + m)$.

Existuje aj alternatívne, komplikovanejšie riešenie s rovnakou časovou zložitosťou. Množinu bezpečných planét vieme zostrojiť aj tak, že v zadanom grafe nájdeme silno súvislé komponenty. Planéta je bezpečná práve vtedy, ak jej komponent obsahuje samé planéty typu 2 a navyše platí, že každý teleport vedúci z tohto komponentu vedie na bezpečnú planétu.

Listing programu (C++)

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

int N, M;
vector<int> typy;
vector< vector<int> > teleporty;

vector<bool> prehladaj(const vector<int> start, bool moze0) {
    vector<bool> navstivil(N+1,false);
    queue<int> Q;
```

```
    for (unsigned i=0; i<start.size(); ++i) {
        navstivil[ start[i] ] = true;
        Q.push( start[i] );
    }
    while (!Q.empty()) {
        int kde = Q.front(); Q.pop();
        for (unsigned i=0; i<teleporty[kde].size(); ++i) {
            int kam = teleporty[kde][i];
            if (!moze0 && typy[kam]==0) continue;
            if (navstivil[kam]) continue;
            navstivil[kam] = true;
            Q.push(kam);
        }
    }
    return navstivil;
}

int main() {
    cin >> N >> M;
    typy.resize(N+1);
    for (int n=1; n<=N; ++n) cin >> typy[n];
    teleporty.resize(N+1);
    while (M--) {
        int x, y;
        cin >> x >> y;
        teleporty[y].push_back(x);
    }

    // prve prehladavanie: z planet typu 0 a 1 cez cokolvek
    vector<int> start1;
    for (int n=1; n<=N; ++n) if (typy[n]<2) start1.push_back(n);
    vector<bool> nebezpecne = prehladaj(start1,true);

    // druhe prehladavanie: z bezpecnych planet cez typ 1 a 2
    vector<int> start2;
    for (int n=1; n<=N; ++n) if (!nebezpecne[n]) start2.push_back(n);
    vector<bool> znesitelne = prehladaj(start2,false);

    // vypis vysledku
    cout << "bezpecne:";
    for (int n=1; n<=N; ++n) if (!nebezpecne[n]) cout << " " << n;
    cout << endl << "znesitelne:";
    for (int n=1; n<=N; ++n) if (nebezpecne[n] && znesitelne[n]) cout << " " << n;
    cout << endl;
}
```

A-I-2 Naložená loď

Prvé riešenie, ktoré väčšine ľudí napadne, je priamočiary „pažravý“ postup. Kým sme ešte neposlali všetky balíčky, opakujeme: nájdeme najväčšiu kapsulu, ktorú ešte vieme naplniť, naplníme ju a pošleme.

Problém je, že toto riešenie nemusí použiť najmenší možný počet kapsúl. Pozorní riešitelia si isto všimli, že aj pre jeden z príkladov uvedených v zadaní existuje lepšie riešenie ako to, ktoré vyrobí náš pažravý postup. Budeme na to teda musieť ísť ináč.

Rozumne málo balíčkov:

Ukážeme si najskôr riešenie, ktoré bude fungovať pre rozumne malý počet balíčkov k, a potom ukážeme, ako toto riešenie vylepšiť, aby fungovalo aj pre obrovské počty balíčkov. Toto prvé riešenie bude založené na myšlienke dynamického programovania: Označme $M[x]$ najmenší počet kapsúl potrebný na prepravenie presne x balíčkov. Pre tieto hodnoty ľahko nájdeme rekurzívny vzťah: Ak mám prepraviť x balíčkov, musím si vybrať nejakú kapsulu i a tou poslať a_i balíčkov (pričom musí byť $a_i \leq x$). Následne mi zostane ešte $x - a_i$ neposlaných balíčkov, a na tie treba ešte $M[x - a_i]$ kapsúl.

Formálne môžeme tento vzťah zapísať nasledovne:

$$M[x] = 1 + \min_{a_i \leq x} M[x - a_i],$$

pričom nesmieme zabudnúť na začiatočnú podmienku $M[0] = 0$ (na 0 balíčkov zjavne stačí 0 kapsúl). Minimum je v tom vzťahu preto, že my máme na výber, akú kapsulu použijeme, a teda si vyberieme tú, ktorá je v danej situácii pre nás najvýhodnejšia.

Pomocou tohto vzťahu vieme konkrétnu hodnotu $M[x]$ spočítať v čase $O(n)$, ak už poznáme všetky hodnoty $M[0..x-1]$. Ak teda budeme postupne počítať hodnoty $M[1]$, $M[2]$, ..., $M[k]$, dostaneme program, ktorý bude mať časovú zložitosť $O(nk)$.

Práve popísaný program nám síce spočíta hodnotu $M[k]$, teda optimálny počet kapsúl, ktoré treba na poslanie k balíčkov, ale to nám nestačí. My navyše potrebujeme aj zostrojiť jeden konkrétny rozvrh, pri ktorom použijeme práve toľko kapsúl. Ako na to? Zjavne si stačí pre každé x zapamätať to i, pre ktoré je $M[x-a_i]$ minimálne – teda poradové číslo kapsuly, ktorú je najlepšie použiť, ak máme presne x balíčkov. Ak je viac možností pre i, zapamätáme si ľubovoľnú jednu z nich.

Pomocou takto zapamätaných informácií už ľahko zostrojíme jedno optimálne riešenie. Začneme s k balíčkami. Pozrieme sa na optimálnu veľkosť kapsuly pre k balíčkov a použijeme ju. Tým sa nám počet balíčkov zmenší na nejaké k'. Preň sa opäť pozrieme na optimálnu veľkosť kapsuly, použijeme ju, a tak ďalej, až kým počet balíčkov neklesne na nulu. Časová zložitosť tohto zostrojenia riešenia je zjavne $O(k)$, lebo v každom kroku zostávajúci počet balíčkov klesá a každý krok vykonáme v konštantnom čase.

Táto fáza je teda zanedbateľná oproti výpočtu hodnôt $M[0..k]$. Celková časová zložitosť nášho algoritmu zostáva rovná $O(nk)$.

Veľmi veľa balíčkov:

Predchádzajúce riešenie pri obmedzení $n \leq 30$ prestáva byť prakticky použiteľné už pri $k = 10^8$. Naše vylepšenie tohto riešenia bude založené na myšlienke, že pre obrovské k sa nám určite oplatí použiť veľakrát najväčšiu kapsulu.

Ako takéto tvrdenie ale dokázať? Predstavme si, že máme napríklad dve kapsuly: jedna na 5 balíčkov, druhá na 7. Čo o nich vieme povedať? Určite sa nám neoplatí použiť 7-krát tú menšiu – namiesto toho by sme použili 5-krát tú väčšiu a tým ušetrili dve kapsuly. Zovšeobecnením tohto príkladu sa dá dokázať nasledujúce tvrdenie: Pre každé $i < n$ platí, že kapsulu číslo i použijeme v každom optimálnom riešení menej ako a_n-krát.

Toto tvrdenie sa však dá ešte zosilniť. Oprieme sa o pomocné tvrdenie z teórie čísel: Nech $s_1, \ldots, s_t$ sú prirodzené čísla. Potom nejaká ich neprázdna podmnožina má súčet deliteľný číslom t. Dôkaz: Uvažujme $t + 1$ súčtov: 0, s_1, $s_1 + s_2$, až $s_1 + \cdots + s_t$. Niektoré dva z nich, nech sú to $s_1 + \cdots + s_i$ a $s_1 + \cdots + s_j$ (pre nejaké $0 \leq i < j \leq t$), dávajú nutne po delení t rovnaký zvyšok. Potom ale $s_{i+1} + \cdots + s_j$ je deliteľné t, a teda sme našli vyhovujúcu podmnožinu.

Čo z tohto tvrdenia pre nás vyplýva? Že v každom optimálnom riešení nanajvýš $(a_n - 1)$-krát použijeme inú ako najväčšiu kapsulu. Totiž ak by sme použili a_n menších kapsúl, podľa práve dokázaného tvrdenia existuje ich podmnožina, ktorej súčet je deliteľný a_n. Namiesto dotyčných menších kapsúl by potom bolo výhodnejšie niekoľkokrát použiť tú najväčšiu.

Hľadanie optimálneho riešenia teda môžeme rozdeliť na dva kroky: Najskôr si pre každý počet balíčkov od 0 po trebárs a_n^2 spočítame, ako ho najlepšie poslať pomocou kapsúl iných ako posledná. Medzi takto spočítanými riešeniami sa určite nachádza aj časť toho celkovo optimálneho. A potom už len vyskúšame všetky možné počty použitia najväčšej kapsuly, ktoré pripadajú do úvahy, a vyberieme z nich ten najlepší. Takéto riešenie má časovú zložitosť $O(na_n^2)$.

Listing programu (C++)

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


int main() {
    // nacitanie vstupu a vypocet hranice, po ktoru pobezi dyn. prog.
    int N; cin >> N;
    vector<long long> A(N);
    for (int n=0; n<N; ++n) cin >> A[n];
    long long K; cin >> K;

    long long K1 = A[N-1]*A[N-1];
    if (K<K1) K1=K;
```

```
    // vypocet optimalnych rieseni pre male pocty balickov
    vector<long long> best(K1+1,1LL<<62), how(K1+1,-1);
    best[0]=0;
    for (int k=1; k<=K1; ++k)
        for (int n=0; n<N-1; ++n)
            if (A[n] <= k)
                if (best[k-A[n]]+1 < best[k])
                    best[k]=best[k-A[n]]+1, how[k]=n;

    // vypocet optimalneho riesenia pre K balickov
    long long bestsum = K+1, K2=-1;
    for (int k=0; k<=K1; ++k)
        if ((K-k)%A[N-1]==0)
            if (best[k]+(K-k)/A[N-1] < bestsum)
                bestsum = best[k]+(K-k)/A[N-1], K2=k;

    // vypis riesenia
    cout << bestsum << endl;
    vector<long long> B(N,0);
    B[N-1] = (K-K2) / A[N-1];
    while (K2) { ++B[how[K2]]; K2 -= A[how[K2]]; }
    for (int n=0; n<N; ++n) cout << B[n] << (n==N-1 ? "\n" : " ");
}
```

A-I-3 Skladník

Riešenie prvej podúlohy:

Hlavnú oštaru nám robí skutočnosť, že veci na vstupe majú mená – reťazce, ktoré musíme spracovať. Keby to boli namiesto reťazcov malé prirodzené čísla, stačilo by v prvej podúlohe použiť obyčajné pole. Takto ale potrebujeme dátovú štruktúru, ktorá nám umožní efektívne meniť množinu reťazcov a pre každý reťazec v nej si pamätať nejaké údaje navyše. (V našom prípade pôjde o jedno celé číslo: aktuálny počet výskytov dotyčného predmetu v sklade.)

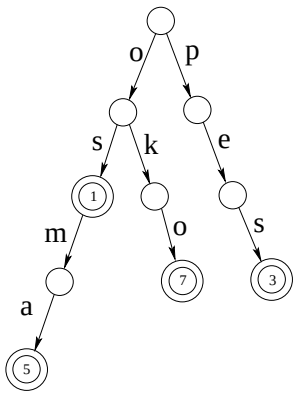
Jedno možné efektívne riešenie je použiť dátovú štruktúru nazývanú písmenkový strom, po anglicky *trie*. (Slovo *trie* by sa správne malo čítať *trí*, lebo názov je odvodený od slova *retrieval*. Mnohí ľudia však preferujú výslovnosť *tráj*, aby ho odlíšili od všeobecného stromu (*tree* sa tiež číta *trí*).)

Písmenkový strom je zakorenený strom, v ktorom má každý vrchol najviac 26 synov. Hrany do synov sú označené rôznymi písmenkami (od `a` po `z`). Každému vrcholu v zodpovedá jedna cesta z koreňa nadol, a tou je jednoznačne určené slovo, zodpovedajúce danému vrcholu. (Toto slovo si „prečítame“ na hranách, po ktorých ideme z koreňa do v.)

Do písmenkového stromu vieme ľahko uložiť množinu slov. Jednoducho vytvoríme všetky vrcholy, ktoré treba na to, aby sme si mohli na ceste z koreňa

dole „prečítať“ každé z našich slov, a následne označíme tie vrcholy, kde niektoré z uložených slov končí. V našom prípade dokonca ani nepotrebujeme nič explicitne označovať. Stačí si v každom vrchole trie pamätať jedno celé číslo: počet vecí so zodpovedajúcim menom, ktoré máme v sklade. Vo vrcholoch, ktorých reťazce nepopisujú veci v sklade, budú jednoducho nuly.

Príklad takéhoto stromu si môžete pozrieť na obrázku 1. Z každého vrcholu vedie v skutočnosti dodola až 26 hrán – tie, ktoré nevedú nikam (NULL pointre), sme pre prehľadnosť nekreslili. Dvojitým krúžkom sú znázornené vrcholy, kde niektoré zo slov končí. V nich sú uvedené čísla, ktoré zodpovedajú počtom daných vecí v sklade. V ostatných vrcholoch sú uložené nuly.



Obr. 1: Trie zodpovedajúci skladu, v ktorom je: 1× os, 5× osma, 7× oko a 3× pes.

Okrem samotného písmenkového stromu na vyriešenie prvej podúlohy už nepotrebujeme skoro nič. Stačia nám dve celočíselné premenné, v ktorých si pamätáme aktuálny počet rôznych typov vecí a aktuálny počet kusov vecí v sklade. Tieto vieme vždy po spracovaní riadku vstupu v konštantnom čase prepočítať. Pri práci s písmenkovým stromom vieme ľubovoľný reťazec spracovať v čase priamo úmernom jeho dĺžke, teda $O(\ell)$. Taká je teda aj celková časová zložitosť spracovania každého riadku vstupu.

Listing programu (C++)

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

struct trie {
    struct __vrchol {
        int pocet;
        __vrchol *syn[26];
        __vrchol() { pocet=0; memset(syn,0,sizeof(syn)); }
    };

    __vrchol *root;
    int celkovo_typov;
    long long celkovo_veci;

    trie() { root = new __vrchol(); celkovo_typov = celkovo_veci = 0; }

    void update(const string &S, int add) {
        // najdeme a ak treba vyrobime zodpovedajuci vrchol
        __vrchol *kde = root;
        for (unsigned i=0; i<S.size(); ++i) {
```

```
            int idx = S[i]-'a';
            if (! kde->syn[idx]) kde->syn[idx] = new __vrchol();
            kde = kde->syn[idx];
        }
        // upravime pamatanu hodnotu
        celkovo_veci += add;
        if (kde->pocet == 0) ++celkovo_typov;
        kde->pocet += add;
        if (kde->pocet == 0) --celkovo_typov;
    }
};

int main() {
    trie T;
    int zmena;
    string nazov;
    while (cin >> zmena >> nazov) {
        T.update(nazov,zmena);
        cout << "kusov: " << T.celkovo_veci;
        cout << ", typov: " << T.celkovo_typov << endl;
    }
}
```

(Práve uvedené riešenie ešte nemá optimálnu pamäťovú zložitosť. Vylepšiť ho vieme nasledovne: vždy, keď zo skladu nejakú vec úplne odstránime, zmažeme tie vrcholy písmenkového stromu, ktoré sú v danej chvíli zbytočné. Takto dosiahneme riešenie, ktorého pamäťová zložitosť je $O(\ell t)$, teda lineárna od súčtu dĺžok názvov vecí aktuálne prítomných v sklade. Program bude kvôli úspore miestom uvedený len v elektronickej verzii týchto riešení.)

Riešenie druhej podúlohy:

Jednou možnosťou, ako túto podúlohu riešiť, je použiť riešenie z prvej podúlohy a len si navyše pamätať, ktorej veci je momentálne v sklade najviac kusov. Takéto riešenie však príliš efektívne nebude. Ako vyzerá jeho najhorší prípad? Ten nastane zakaždým, keď zo skladu odnesú niekoľko kusov tej veci, ktorej je v danej chvíli najviac. Keďže o počtoch ostatných vecí nič netušíme, musíme ich prezrieť všetky. V najhoršom možnom prípade teda budeme musieť nájsť najväčší spomedzi t záznamov, čo sa dá spraviť s časovou zložitosťou $O(\ell t)$.

Ako sa dá takémuto zlému prípadu vyhnúť? Záznamy o veciach sa oplatí *udržiavať usporiadané* podľa aktuálneho počtu kusov. Pri každej operácii potom obetujeme trochu času na „upratanie“ – preusporiadanie záznamov tak, aby boli naďalej usporiadané. Odmenou nám bude záruka, že každú požiadavku spracujeme rozumne rýchlo, bez nutnosti prezerať zbytočne veľa záznamov.

Aké teda požiadavky máme na novú dátovú štruktúru? Potrebujeme vedieť efektívne robiť dve veci: Prvou je nájsť záznam zodpovedajúci konkrétnemu typu a zmeniť v ňom počet kusov. No a druhou je nájdenie záznamu, ktorý má

momentálne najväčší počet kusov. Najjednoduchším riešením bude k písmenkovému stromu pridať ešte jednu dátovú štruktúru: *haldu*.[2]

V halde budeme mať pre každý typ vecí jeden záznam, a v ňom si budeme pamätať názov daného typu a počet jeho kusov. Záznamy v halde budú usporiadané podľa zadania – teda primárne podľa počtu kusov a sekundárne podľa abecedy. Navyše budú naše dve dátové štruktúry medzi sebou prepojené: v každom vrchole písmenkového stromu si budeme pamätať, kde je zodpovedajúci záznam v halde (ak existuje) a v každom zázname haldy si budeme pamätať, ktorému vrcholu písmenkového stromu zodpovedá.

Spracovanie každej inštrukcie potom bude vyzerať nasledovne:

1. Načítame inštrukciu.
2. Podľa názvu typu vecí nájdeme zodpovedajúci vrchol v písmenkovom strome.
3. Pomocou hodnoty, ktorú si v danom vrchole pamätáme, nájdeme zodpovedajúci záznam v halde.
4. Príslušne upravíme počet výskytov daného typu vecí.
5. Zmenou v predchádzajúcom kroku sme mohli porušiť podmienku haldy. Zmenený záznam preto v prípade potreby presunieme haldou dohora alebo dodola tak, aby sme opäť dostali platnú haldu.

Najpomalšou časťou riešenia je posledný krok, v ktorom upravujeme haldu. Počet záznamov v halde je t, jej hĺbka je teda $O(\log t)$. Toľkokrát môže byť potrebné zmenený záznam presunúť o úroveň vyššie či nižšie. Zakaždým musíme daný záznam porovnať so záznamom bezprostredne nad ním a dvomi bezprostredne pod ním, a na každé toto porovnanie treba $O(\ell)$ času – v prípade rovnosti počtov totiž môže byť potrebné porovnať aj názvy. Celková časová zložitosť úpravy haldy je teda $O(\ell \log t)$.

Listing programu (C++)

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

struct __vrchol {
    __vrchol *syn[26], *otec;
    int index_halda;
    __vrchol(__vrchol* otec=NULL) : otec(otec) {
```

[2] Popis haldy neuvádzame. Záujemcom odporúčame napríklad text o haldách na adrese `http://foja.dcs.fmph.uniba.sk/ads/materialy.php`.

```
        index_halda=-1; memset(syn,0,sizeof(syn));
    }
};

struct __zaznam {
    __vrchol *trie_vrchol;
    string nazov;
    int pocet;
    __zaznam(__vrchol *t, string n, int p) : trie_vrchol(t), nazov(n), pocet(p) {}
};

struct trie {
    __vrchol *root;

    trie() { root = new __vrchol(); }

    __vrchol *find(const string &S) {
        // najde (a ak treba, vyrobi) vrchol zodpovedajuci retazcu S
        __vrchol *kde = root;
        for (unsigned i=0; i<S.size(); i++) {
            int idx = S[i]-'a';
            if (! kde->syn[idx]) kde->syn[idx] = new __vrchol(kde);
            kde = kde->syn[idx];
        }
        return kde;
    }
};

bool operator< (const __zaznam &A, const __zaznam &B) {
    if (A.pocet != B.pocet) return A.pocet < B.pocet; return A.nazov > B.nazov;
}

struct halda_a_trie {
    trie T;
    vector<__zaznam> H; // halda

    void vymen(int x, int y) {
        // vymeni v halde zaznamy na poziciach x a y, zaroven patricne upravi trie
        __vrchol *tx = H[x].trie_vrchol, *ty = H[y].trie_vrchol;
        swap(H[x],H[y]);
        tx->index_halda = y; ty->index_halda = x;
    }

    int uprac_haldu(int x) {
        // presuva prvok x dohora alebo dodola podla potreby,
        // az kym nie je halda napravena
        while (true) {
            int y=x;
            if (x>0 && H[(x-1)/2]<H[y]) y=(x-1)/2;
            if (2*x+1 < H.size() && H[y]<H[2*x+1]) y=2*x+1;
            if (2*x+2 < H.size() && H[y]<H[2*x+2]) y=2*x+2;
            if (y != x) { vymen(x,y); x=y; } else break;
        }
        return x;
    }

    void update(const string &nazov, int add) {
        __vrchol *kde = T.find(nazov);
        // ak treba, pridame novy zaznam do haldy,
        // nasledne upravime zaznam v halde
        if (kde->index_halda==-1) {
            kde->index_halda=H.size();
            H.push_back(__zaznam(kde,nazov,0));
        }
```

```
        H[ kde->index_halda ].pocet += add;
        int x = uprac_haldu(kde->index_halda);
        // ak sme zmensili pocet na nulu, vymazeme zaznam z haldy
        if (H[x].pocet==0) {
            vymen(x,H.size()-1);
            H[H.size()-1].trie_vrchol->index_halda=-1;
            H.pop_back();
            uprac_haldu(x);
        }
    }
};

int main() {
    halda_a_trie HT;
    int zmena;
    string nazov;
    while (cin >> zmena >> nazov) {
        HT.update(nazov,zmena);
        if (HT.H.empty() || HT.H[0].pocet==0) cout << endl;
        else cout << HT.H[0].nazov << " " << HT.H[0].pocet << endl;
    }
}
```

Alternatívnym riešením by bolo namiesto haldy použiť vyvažovaný binárny strom. V jazykoch, ktoré majú takúto dátovú štruktúru v štandardnej knižnici, sa toto riešenie implementuje ľahšie. Jeho časová zložitosť je rovnaká.

Ešte iné efektívne riešenie odovzdal Andrej Mariš. Podobne ako v prvej podúlohe si budeme údaje pamätať v písmenkovom strome. Navyše si v každom jeho vnútornom vrchole budeme pamätať maximum spomedzi počtov výskytov vecí, ktorých názvy daným vrcholom prechádzajú. Takéto riešenie má časovú zložitosť $O(\ell\sigma)$ na jednu operáciu, kde σ je počet rôznych písmen v abecede. (Napr. ak používame len malé písmená anglickej abecedy, tak $\sigma = 26$). Šikovnejšou implementáciou tohoto riešenia by sa jeho časová zložitosť dala zlepšiť na $O(\ell \log \sigma)$.

A-I-4 Zlomkové programy

Testovanie rovnosti:

Na vstupe je číslo n tvaru $2^x 3^y 5$, pričom $x, y \geq 0$. Našou úlohou je napísať program, ktorý ho prerobí na číslo 5, ak $x = y$, resp. na číslo 7, ak $x \neq y$.

Základom programu bude zlomok $1/6$. Zakaždým, keď ten použijeme vo výpočte, zmenšíme tým x aj y o jedna. Ak teda na začiatku $x = y$, nič iné ani nepotrebujeme: po x použitiach zlomku $1/6$ dostaneme číslo 5 a môžeme skončiť.

Čo sa stane, ak $x \neq y$? Program, obsahujúci jediný zlomok 1/6, by sa po konečnom počte krokov zasekol, lebo by sa mu už minuli prvočísla jedného typu. Ak by napríklad bolo $x > y$, program by skončil s hodnotou $2^{x-y}5$.

Ak nastane táto (alebo opačná) situácia, potrebujeme v prvočíselnom rozklade nášho čísla vyrobiť 7 a následne sa zbaviť všetkého okrem tejto 7. Prvý krok zabezpečia zlomky $7/(2 \cdot 5)$ a $7/(3 \cdot 5)$ a následný druhý krok zlomky 1/2 a 1/3. (Tieto musia byť uvedené až na konci programu, aby sa nepoužili skôr.)

Celý program teda vyzerá nasledovne:

$$\left(\frac{1}{6} \ , \ \frac{7}{10} \ , \ \frac{7}{15} \ , \ \frac{1}{2} \ , \ \frac{1}{3} \right)$$

Príklad výpočtu pre $n = 2^4 3^4 5$:

$$2^4 3^4 5 \xrightarrow{1} 2^3 3^3 5 \xrightarrow{1} 2^2 3^2 5 \xrightarrow{1} 2^1 3^1 5 \xrightarrow{1} 5$$

Príklad výpočtu pre $n = 2^2 3^5 5$:

$$2^2 3^5 5 \xrightarrow{1} 2^1 3^4 5 \xrightarrow{1} 3^3 5 \xrightarrow{3} 3^2 7 \xrightarrow{5} 3^1 7 \xrightarrow{5} 7$$

Delenie dvoma:

Na vstupe je číslo n tvaru $2^x 3$, pričom $x \geq 0$. Našou úlohou je napísať program, ktorý ho prerobí na číslo tvaru 2^y, kde $y = \lfloor x/2 \rfloor$.

Toto samozrejme nepôjde robiť tak, že priamo zmeníme mocninu 2, ktorá naše číslo delí. Jednoduchšie by bolo použiť ako „pomocnú premennú“ exponent nejakého iného prvočísla, napríklad čísla 7.

Mohli by sme teda napríklad v programe použiť zlomok 7/4. Každé jeho použitie zníži mocninu 2 o dve a zvýši mocninu 7 o jedno. Časom by sme sa takto dopracovali buď k aktuálnej hodnote $2^1 7^y 3$ alebo $7^y 3$, podľa parity x.

Zlomok 7/4 však priamo použiť nemôžeme. Prečo? Lebo potrebujeme, aby náš výpočet skončil. Ale ak na konci opäť vyrobíme číslo, ktoré je mocninou 2, výpočet by neskončil, lebo by sa znovu dal použiť tento zlomok.

Tu prichádza k slovu číslo 3, ktorým je vstup deliteľný. Prítomnosť prvočísla 3 budeme chápať ako signál, že ešte sme v prvej fáze výpočtu, kedy premieňame 2ky na 7ky. Namiesto menovateľa 4 budeme teda používať menovateľ $4 \cdot 3 = 12$. Zlomok 7/12 však tiež ešte nie je presne to, čo potrebujeme – dal by sa použiť len raz, lebo jeho použitím odstránime prvočíslo 3 z rozkladu aktuálnej hodnoty. Potrebujeme ešte vedieť toto odstránené prvočíslo vrátiť späť.

Fungovať bude napríklad nasledujúca konštrukcia: namiesto jedného zlomku $7/4$ použijeme dvojicu zlomkov $(7 \cdot 5)/(4 \cdot 3)$ a $3/5$. Postupné použitie týchto dvoch zlomkov bude mať rovnaký efekt ako použitie zlomku $7/4$, ale uskutočniť sa môže len vtedy, keď je na začiatku aktuálna hodnota deliteľná 3-kou.

A už stačí len doriešiť upratovanie. Za tieto dva zlomky pridáme zlomky $1/6$ a $1/3$, ktoré sa zbavia deliteľa 3 a prípadného posledného deliteľa 2, ak x bolo nepárne. Takto dostávame program, ktorý z čísla $2^x 3$ vyrobí číslo 7^y, kde $y = \lfloor x/2 \rfloor$. A už stačí len na konci zmeniť 7ky na 2ky zlomkom $2/7$.

Celý program teda vyzerá nasledovne:

$$\left(\frac{35}{12} \ , \ \frac{3}{5} \ , \ \frac{1}{6} \ , \ \frac{1}{3} \ , \ \frac{2}{7} \right)$$

Príklad výpočtu pre $n = 2^7 3$:

$$\begin{aligned} 2^7 3 &\xrightarrow{1} 2^5 7^1 5 \xrightarrow{2} 2^5 7^1 3 \xrightarrow{1} 2^3 7^2 5 \xrightarrow{2} 2^3 7^2 3 \xrightarrow{1} 2^1 7^3 5 \\ &\xrightarrow{2} 2^1 7^3 3 \xrightarrow{3} 7^3 \qquad \xrightarrow{4} 2^1 7^2 \qquad \xrightarrow{4} 2^2 7^1 \qquad \xrightarrow{4} 2^3 \end{aligned}$$

Riešenia domáceho kola kategórie B

B-I-1 Teploty

Na zisk 6 bodov stačilo naprogramovať najjednoduchšie možné riešenie: každý z $n - t + 1$ riadkov výstupu vyrobíme tak, že prejdeme všetky zodpovedajúce záznamy, nájdeme ich minimum, súčet a maximum a na záver priemer vypočítame ako súčet deleno t. Časová zložitosť takéhoto riešenia je teda približne priamo úmerná hodnote $t(n - t)$. (Matematicky to zapisujeme: časová zložitosť je $O(t(n - t))$.)

Takéto riešenie dokonca mohlo **trpezlivému** riešiteľovi stačiť aj na 10 bodov. Odhadnime si totiž, ako dlho budeme musieť čakať na výsledok pre najväčší vstup. Výstup pre vstup `5.txt` má 500 001 riadkov, a pre každý z nich potrebujeme spracovať 500 000 hodnôt. To znamená, že náš program spraví rádovo $500\,000^2 \approx 2.5 \cdot 10^{11}$ logických krokov.

Pri súčasných počítačoch je celkom dobrý odhad, že za sekundu stihne vykonať asi miliardu inštrukcií, čo zhruba zodpovedá 10^8 jednoduchým logickým krokom. Náš hrubý odhad teda hovorí, že na priemernom súčasnom počítači by malo vyriešenie vstupu `5.txt` byť zvládnuteľné rádovo za hodinu. (Od efektívnosti počítača, zvoleného programovacieho jazyka a šikovnosti implementácie závisí, či to bude 13 minút alebo 11 hodín. Ale tak či onak to bude dostatočne rýchlo.)

Trpezlivosti sa dalo pomôcť rôznymi vylepšeniami. Niektoré z nich fungujú len občas – napríklad riešenie, v ktorom minimum/maximum prerátame len vtedy, keď to treba (hodnota, ktorá práve opustila spracúvaný interval, je rovná dovtedajšiemu minimu/maximu). V tomto vzorovom riešení si však popíšeme niekoľko algoritmov, u ktorých máme istotu, že budú vždy efektívnejšie ako riešenie „hrubou silou“.

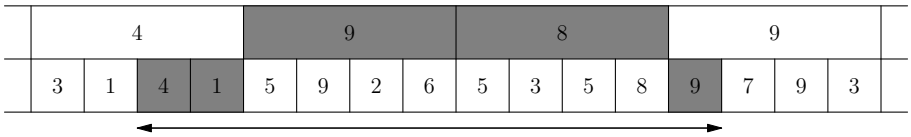
Zaručene lepšie riešenie pre priemer:

Priemernú teplotu vieme ľahko počítať omnoho efektívnejšie. Predstavme si, že sme práve spočítali priemer teplôt $a_1, \ldots, a_t$. Teda poznáme súčet $s = a_1 + \cdots + a_t$. Ako vypočítame priemer pre nasledujúce meranie? Na ten potrebujeme poznať súčet $a_2 + \cdots + a_{t+1}$. Lenže ten vieme vypočítať v konštantnom čase: ako $(s - a_1 + a_{t+1})$.

Zaručene lepšie riešenie pre minimum a maximum:

Ako zrýchliť výpočet minima a maxima? Treba nejako zabrániť tomu, aby sme zakaždým prechádzali všetkých t posledných meraní. Jednou možnou cestou je rozdeliť si vstup na bloky nejakej pevne zvolenej veľkosti. Každému bloku len raz spočítame minimum a maximum a zapamätáme si ich. A potom vždy, keď potrebujeme nájsť napr. maximum spomedzi nejakého úseku hodnôt, tak sa pozrieme na maximá blokov, ktoré sú celé vo vnútri nášho úseku. Po jednom prvku potrebujeme prejsť maximálne jeden blok na začiatku a jeden na konci.

Celé to je názorne ukázané na nasledujúcom obrázku:



V spodnom riadku sú jednotlivé namerané teploty. Toto pole sme si „nasekali“ na bloky dĺžky 4. Tie sú znázornené v hornom riadku – pre každý blok máme jedno políčko a v ňom maximum spomedzi hodnôt v danom bloku. Šípkami je znázornený interval, ktorého maximum nás zaujíma. Na zistenie tohoto maxima sa stačí pozrieť na farebne zvýraznené políčka.

Ako zvoliť dĺžku bloku? Ak si zvolíme primalé bloky, bude ich treba prejsť veľa. A ak si zvolíme bloky priveľké, bude na začiatku aj na konci úseku treba pozerať veľa jednotlivých prvkov. Dobrou voľbou je zlatá stredná cesta: hodnota $b = \lfloor \sqrt{t} \rfloor$. Úsek dĺžky t bude obsahovať najviac b celých blokov, a taktiež bude treba prezrieť najviac $b-1$ prvkov na začiatku a najviac $b-1$ prvkov na konci za posledným celým blokom. Dokopy nám teda na nájdenie maxima pre konkrétny úsek stačí pozrieť na menej ako $3\sqrt{t}$ hodnôt.

Dostávame teda riešenie s časovou zložitosťou $O(\sqrt{t}(n-t))$. Toto riešenie už aj najväčší zo zadaných vstupov vyrieši za pár sekúnd.

Listing programu (Pascal)

```
var n, t, bl : longint; { pocet merani, dlzka useku, dlzka bloku }
    teploty, bloky_max, bloky_min : array of longint;

function max(x,y : longint) : longint; begin if x>y then max:=x else max:=y; end;
function min(x,y : longint) : longint; begin if x<y then min:=x else min:=y; end;

procedure nacitaj;
var i : longint;
    x : double;
begin
   read(n,t);
   bl := trunc(sqrt(t));
   setlength(teploty,n);
```

```
   setlength(bloky_max,(n div bl)+1);
   setlength(bloky_min,(n div bl)+1);
   for i:=0 to n-1 do begin
      read(x); teploty[i]:=round(x*100);
      if i mod bl=0 then begin
         bloky_max[i div bl] := teploty[i];
         bloky_min[i div bl] := teploty[i];
      end else begin
         bloky_max[i div bl] := max( bloky_max[i div bl], teploty[i] );
         bloky_min[i div bl] := min( bloky_min[i div bl], teploty[i] );
      end;
   end;
end;

procedure max_min_useku(zac, kon : longint; var odpoved_max, odpoved_min : longint);
begin
   odpoved_max := -10000; odpoved_min := 10000;
   { spracujeme prvky po najblizsi zaciatok bloku }
   while (zac<kon) and (zac mod bl > 0) do begin
      odpoved_max := max( odpoved_max, teploty[zac] );
      odpoved_min := min( odpoved_min, teploty[zac] );
      inc(zac);
   end;
   { spracuvame cele bloky }
   while (zac+bl <= kon) do begin
      odpoved_max := max( odpoved_max, bloky_max[zac div bl] );
      odpoved_min := min( odpoved_min, bloky_min[zac div bl] );
      inc(zac,bl);
   end;
   { spracujeme prvky po koniec useku }
   while (zac<kon) do begin
      odpoved_max := max( odpoved_max, teploty[zac] );
      odpoved_min := min( odpoved_min, teploty[zac] );
      inc(zac);
   end;
end;

procedure vyries;
var i,mx,mn : longint;
    sucet : int64;
begin
   sucet := 0; for i:=0 to t-2 do sucet:=sucet+teploty[i];
   for i:=0 to n-t do begin
      { spocitame odpovede pre usek zacinajuci na indexe i }
      sucet:=sucet+teploty[i+t-1];
      if i>0 then sucet:=sucet-teploty[i-1];
      max_min_useku(i,i+t,mx,mn);
      writeln( (mn/100):0:2, '␣', (sucet/(t*100)):0:4, '␣', (mx/100):0:2 );
   end;
end;

begin
   nacitaj;
   vyries;
end.
```

Iné ľahké vylepšenie:

Taktiež sa na urýchlenie výpočtu minima/maxima dalo využiť to, že hodnoty na vstupe sú len z malého rozsahu. Keď všetky teploty vynásobíme číslom 100, dostaneme celé čísla od -8000 do 6000. A tie môžeme použiť ako indexy do

poľa, v ktorom si budeme pamätať, ktorú hodnotu koľkokrát práve spracúvaný úsek obsahuje. Takto dosiahneme situáciu, kedy väčšinou vieme minimum/maximum povedať v konštantnom čase. Len občas (práve vtedy, keď dovtedajšie minimum/maximum prestane ležať v aktuálne spracúvanom úseku) potrebujeme prechádzať naším pomocným poľom a nájsť najbližšiu väčšiu/menšiu hodnotu, ktorá sa v aktuálnom úseku vyskytuje. (Všimnite si, že počet krokov, ktoré pri hľadaní nového minima/maxima spravíme, nezávisí od t.)

Nad rámec riešenia tejto úlohy:

Pokročilé dátové štruktúry

Veľmi stručnú a zároveň efektívnu implementáciu vieme dosiahnuť v programovacích jazykoch, ktoré nám ponúkajú dátovú štruktúru „usporiadaná množina“. (Napr. `set` v C++, `TreeSet` v Jave.) Táto dátová štruktúra je zväčša implementovaná ako vyvažovaný binárny strom. Ak si v nej budeme pamätať t aktuálne spracúvaných meraní, budeme vedieť v čase $O(\log t)$ zistiť aktuálne minimum/maximum, a taktiež v čase $O(\log t)$ prejsť na nasledujúci úsek meraní.

Listing programu (C++)

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

int get() { double x; cin >> x; return int(round(100*x)); }

int main() {
    int N, T; cin >> N >> T;
    vector<int> vstup;
    for (int n=0; n<N; ++n) vstup.push_back(get());
    multiset<int> usek;
    long long sucet = 0;
    for (int t=0; t<T-1; ++t) { usek.insert(vstup[t]); sucet += vstup[t]; }
    for (int t=T-1; t<N; ++t) {
        usek.insert(vstup[t]); sucet += vstup[t];
        if (t>=T) { usek.erase(usek.find(vstup[t-T])); sucet -= vstup[t-T]; }
        cout << *usek.begin()/100. << " ";
        cout << sucet/(100.*T) << " " << *usek.rbegin()/100. << "\n";
    }
}
```

Rovnako efektívne riešenie vieme dostať pomocou intervalového stromu. Záujemcom odporúčame prečítať si riešenie úlohy A-I-1 z 25. ročníka OI. Ide vlastne o vylepšenie vyššie popísaného riešenia s blokmi: pri intervalovom strome vstup rozdelíme na $n/2$ blokov veľkosti 2, tie pospájame do $n/4$ väčších blokov veľkosti 4, tie do blokov veľkosti 8, a tak ďalej.

Optimálne riešenia

Existuje dokonca aj riešenie, ktoré má aj v najhoršom možnom prípade časovú zložitosť $O(n)$ – každý možný vstup teda spracuje v čase lineárnom od jeho veľkosti. Detaily tohoto riešenia nebudeme uvádzať. Prípadným záujemcom poradíme, že je myšlienkovo príbuzné riešeniu z časti „Iné ľahké vylepšenie“. Namiesto poľa so všetkými hodnotami si ale treba vo vhodnej dátovej štruktúre (deque, teda tzv. obojsmerná fronta) pamätať len tie hodnoty, ktoré majú teoretickú šancu niekedy sa stať maximom úseku.

Iné riešenie s celkovou časovou zložitosťou $O(n)$ vyzerá nasledovne: rozdelíme si vstup na bloky dĺžky t a v každom bloku pre každý prefix aj pre každý sufix spočítame jeho minimum/maximum. Pomocou týchto údajov vieme teraz minimum ľubovoľného úseku dĺžky t určiť v konštantnom čase.

B-I-2 Sudoku

Riešenie tejto úlohy neobsahuje žiadne hlboké myšlienky. Na získanie bodov stačilo správne pochopiť, zovšeobecniť a implementovať pravidlá uvedené v zadaní. Popíšeme si, ako to malo celé vyzerať.

Prvé pravidlo je zbytočné:

To samozrejme nie je úplne pravda. Implementuje sa najľahšie a pár bodov sa zaň dalo získať. Ak ale plánujeme získať všetkých 10 bodov, nemá zmysel strácať čas implementovaním prvého pravidla. Prečo? Preto, že vždy, keď niečo vieme odvodiť prvým pravidlom, vieme to isté odvodiť aj druhým pravidlom – ak sú v danom riadku/stĺpci/štvorci plné všetky políčka okrem jedného, keď sa na dotyčné prázdne políčko pozrieme, budeme už mať len jednu možnosť.

Implementácia druhého a tretieho pravidla:

Tieto pravidlá sa síce dajú implementovať každé zvlášť, avšak môžeme si ich obe zjednodušiť tým, že si budeme pamätať vhodné pomocné údaje. Presnejšie, pre každé políčko si budeme pamätať, ktoré cifry sa na ňom ešte môžu vyskytnúť bez toho, aby vznikol priamy konflikt. Na toto je najjednoduchšie použiť pole boolovských premenných – presnejšie teda trojrozmerné pole: pre každý riadok r, stĺpec s a cifru c si budeme pamätať, či môžeme cifru c umiestniť na pozíciu (r, s).

(Pre už zaplnené políčka nastavíme všetkým cifrám, vrátane tej na ňom použitej, hodnotu `false`. Takto ich odlíšime od políčok, kde už ostáva len jedna možnosť, ale ešte sme ju tam neumiestnili.)

Hodnoty v poli je ľahké udržiavať: vždy, keď umiestnime nejakú cifru, stačí prejsť jej riadok, stĺpec a štvorec 3×3 a „vyškrtať ju“ – teda zakázať jej výskyty na všetkých spomínaných políčkach.

Ako bude vyzerať inicializácia nášho poľa? To bude tiež jednoduché: stačí na začiatku nastaviť, že každá cifra môže byť všade, a potom postupne po jednej popridávať cifry zo vstupu.

Druhé pravidlo teraz môžeme implementovať nasledovne: zaradom prejdeme všetky políčka, pre každé z nich skontrolujeme, či náhodou nemá práve jednu možnú cifru, a ak áno, tak ju tam umiestnime. No a takisto pri treťom pravidle sa stačí pozerať do nášho pomocného poľa a skontrolovať, či má daná cifra v danom riadku/stĺpci/štvorci už len jednu možnú pozíciu.

Listing programu (Pascal)

```
program sudoku;

var moze : array[1..9,1..9,1..9] of boolean; { moze byt na pozicii A,B cifra C? }
    riesenie : array[1..9,1..9] of longint;

procedure inicializuj;
var i,j,k : longint;
begin
   for i:=1 to 9 do for j:=1 to 9 do for k:=1 to 9 do moze[i,j,k]:=true;
   for i:=1 to 9 do for j:=1 to 9 do riesenie[i,j]:=0;
end;

procedure umiestni_cifru(r,s,c : longint);
var i,j,br,bs : longint;
begin
   riesenie[r,s]:=c;
   for i:=1 to 9 do begin
      moze[i,s,c]:=false; moze[r,i,c]:=false; moze[r,s,i]:=false;
   end;
   br:=(r-1) div 3; bs:=(s-1) div 3;
   for i:=1 to 3 do for j:=1 to 3 do moze[3*br+i,3*bs+j,c]:=false;
end;

procedure nacitaj;
var i,j : longint;
    s : string;
begin
   for i:=1 to 9 do begin
      readln(s);
      for j:=1 to 9 do if s[j]<>'.' then umiestni_cifru(i,j,ord(s[j])-48);
   end;
end;

procedure vypis;
var i,j : longint;
begin
   for i:=1 to 9 do begin
      for j:=1 to 9 do write(riesenie[i,j]);
      writeln;
   end;
end;
```

```
function hotovo : boolean;
var i,j : longint;
begin
   hotovo := true;
   for i:=1 to 9 do for j:=1 to 9 do if riesenie[i,j]=0 then hotovo := false;
end;

procedure druhe_pravidlo;
var r,s,c,kolko,co : longint;
begin
   for r:=1 to 9 do for s:=1 to 9 do begin
      kolko := 0;
      for c:=1 to 9 do if moze[r,s,c] then begin inc(kolko); co:=c; end;
      if kolko=1 then umiestni_cifru(r,s,co);
   end;
end;

procedure tretie_pravidlo;
var rr,ss,r,s,c,kolko,kder,kdes : longint;
begin
   for c:=1 to 9 do begin
      { riadky }
      for r:=1 to 9 do begin
         kolko := 0;
         for s:=1 to 9 do if moze[r,s,c] then begin inc(kolko); kdes:=s; end;
         if kolko=1 then umiestni_cifru(r,kdes,c);
      end;
      { stlpce }
      for s:=1 to 9 do begin
         kolko := 0;
         for r:=1 to 9 do if moze[r,s,c] then begin inc(kolko); kder:=r; end;
         if kolko=1 then umiestni_cifru(kder,s,c);
      end;
      { stvorce 3x3 }
      for rr:=0 to 2 do for ss:=0 to 2 do begin
         kolko := 0;
         for r:=1 to 3 do for s:=1 to 3 do begin
            if moze[3*rr+r,3*ss+s,c] then begin
               inc(kolko); kder:=3*rr+r; kdes:=3*ss+s;
            end;
         end;
         if kolko=1 then umiestni_cifru(kder,kdes,c);
      end;
   end;
end;

var t : longint;

begin
   readln(t);
   while t>0 do begin
      dec(t);
      inicializuj;
      nacitaj;
      while not hotovo do begin druhe_pravidlo; tretie_pravidlo; end;
      vypis;
   end;
end.
```

B-I-3 Uhádni deň

Na úvod riešenia si treba uvedomiť, že sa zaobídeme aj bez komplikovaných otázok ako táto: „Mal si tento rok narodeniny v utorok?“ Namiesto nej totiž môžeme položiť rovnocennú otázku: „Máš narodeniny v jeden z nasledovných dní: 3, 10, 17, 24, 31?“ Každú otázku vieme preformulovať na ekvivalentnú otázku o číslach dní. Stačí nám teda riešiť jednoduchšiu úlohu: na čo najmenej otázok uhádnuť číslo od 1 do 31.

Podúloha a):

Dobrá stratégia pre Aničku (neskôr si dokonca ukážeme, že najlepšia možná) je pýtať sa tak, aby sa jej po každej otázke približne na polovicu zmenšil počet možností, ktoré ešte pripadajú do úvahy.

Začať môže napríklad otázkou: „Je deň tvojich narodenín číslo od 1 do 15?“

Ak dostane odpoveď „áno“, zmenšil sa počet možností z 31 na 15. A ak dostane odpoveď „nie“, prichádzajú už do úvahy len čísla 16 až 31, čo je 16 možností.

A rovnako môže Anička pokračovať ďalej – vždy rozdelí čísla, ktoré ešte prichádzajú do úvahy, na dve rovnako veľké kôpky a opýta sa, či je hľadané číslo na prvej kôpke. (Ak má nepárny počet možností, prvá kôpka bude mať o číslo menej ako druhá.) Ak napríklad začala vyššie uvedenou otázkou a dostala odpoveď „nie“, v druhom kole môže položiť otázku „Je deň tvojich narodenín číslo od 16 do 23?“.

Ako veľa otázok bude Aničke stačiť pri tomto postupe? Po prvej otázke jej ostávalo najviac 16 možností. Po druhej tento počet klesne na 8, po tretej na 4, po štvrtej na 2 a po prípadnej piatej otázke (ak ju ešte treba) už určite ostala len jedna možnosť – tá správna.

Ak teda Miško dovolí Aničke položiť päť otázok, má Anička pri použití našej stratégie istotu, že jeho deň narodenín uhádne.

Podúloha b):

Napriek tomu, že je Miškova situácia ťažšia, aj jemu stačí položiť len päť otázok. Ako to má spraviť? Jednoducho sa zakaždým šikovne opýta naraz všetky otázky, ktoré by sa v danú chvíľu mohla opýtať Anička.

Prvú otázku položí Miško rovnakú ako Anička.

Pozrime sa teraz na Aničkinu druhú otázku. Tá mohla vyzerať dvomi spôsobmi: ak na prvú otázku dostala odpoveď „áno“, opýtala by sa druhú otázku „Je deň tvojich narodenín číslo od 1 do 7?“, ak nie, tak otázku „Je deň tvojich narodenín číslo od 16 do 23?“.

Miškova druhá otázka teraz môže vyzerať nasledovne: „Je deň tvojich narodenín medzi číslami (1 až 7) a (16 až 23)?“

A rovnako pokračujeme aj ďalej. Tretia Miškova otázka by vyzerala takto: „Je deň tvojich narodenín medzi číslami (1 až 3), (8 až 11), (16 až 19) a (24 až 27)?“

Z odpovedí na všetkých päť otázok vie Miško zakaždým jednoznačne zostrojiť hľadané číslo.

Iný pohľad na toto isté riešenie:

Číslo x, ktoré sa Miško snaží uhádnuť, vieme zapísať v dvojkovej sústave. Keďže je z rozsahu od 1 do 31, určite nám na to stačí päť bitov. (Kratšie čísla si doplníme zľava nulami, teda napr. 6 zapíšeme ako 00110, 18 zapíšeme ako 10010.)

No a Miškovu stratégiu, ktorú sme vyššie popísali, môžeme teraz formulovať jednoduchšie: každou otázkou sa dozvieme jeden bit hľadaného čísla.

Napríklad otázku „Je deň tvojich narodenín medzi číslami (1 až 7) a (16 až 23)?“ môžeme preformulovať takto: „Má číslo x druhý bit zľava rovný nule?“

Podúloha c):

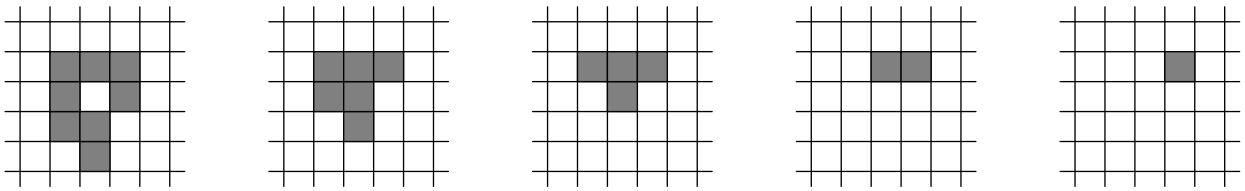
Teraz potrebujeme dokázať, že Miškovi nemôže stačiť menej ako päť otázok. Dôkaz vôbec nebude závisieť na tom, ako vyzerajú Miškove otázky – využijeme len to, že na primálo otázok dostaneme primálo odpovedí.

Predstavme si, že Miško napísal na papier len štyri otázky. Anička teraz zobrala jeho papier a ide k nim napísať svoje odpovede. V tejto chvíli je len $2^4 = 16$ možností, aké odpovede Anička napíše. Lenže dní, ktoré pripadajú do úvahy, je až 31, no a $31 > 16$. Preto určite (podľa Dirichletovho princípu) budú existovať dva rôzne dni, pre ktoré Anička napíše na Miškov papier tú istú postupnosť odpovedí. No a práve tým vzniká problém – ak sa Anička narodila v jeden z týchto dní, tak Miškovi ostane viac ako jedna možnosť. Miško teda nebude mať istotu, že vie, kedy sa Anička narodila.

(Rovnako by dôkaz fungoval, ak by Miško položil menej ako štyri otázky. Rozmyslite si tiež, že z tohto dôkazu vyplýva aj to, že aj Anička na uhádnutie Miškovych narodenín potrebuje aspoň päť otázok.)

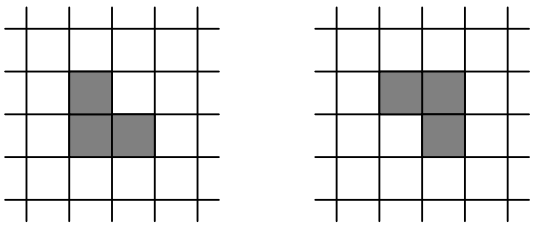
B-I-4 Čierne štvorce

Podúloha a):
Situácia z obrázku v zadaní sa počas nasledujúcich minút vyvíjala nasledovne:



Po piatich minútach už boli všetky štvorce biele.

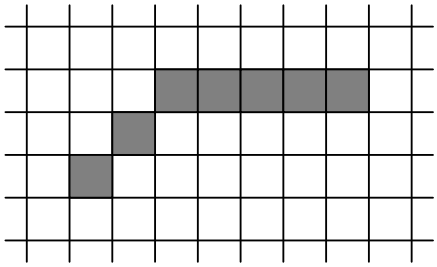
Podúloha b):
Po príklad takého rozmiestnenia netreba chodiť ďaleko – najmenšie má len tri štvorce! Na obrázku vľavo je pôvodné rozmiestnenie, vpravo jeho vzhľad o minútu neskôr.



Podúloha c):
Po chvíli experimentovania ľahko odhalíme viaceré také rozmiestnenia štvorcov. Asi najjednoduchším je jednoducho 7 čiernych štvorcov v rade vedľa seba.

Podúloha d):
Dá sa to pre každé n a k. Z predchádzajúcej podúlohy už vieme, že ak umiestnime k štvorcov do radu, práve po k minútach dostaneme všetky štvorce biele. Následne stačí správne pridať ďalších $n-k$ štvorcov tak, aby to celé držalo pokope a aby ich pridanie neovplyvnilo tých prvých k. Na to ich napríklad stačí vhodne diagonálne pripojiť k prvým k štvorcom. Potom hneď po prvej minúte všetky zmiznú a ostane len $k-1$ z pôvodných k štvorcov.

Na nasledujúcom obrázku je príklad pre $n = 7$ a $k = 5$.



Riešenia krajského kola kategórie A

A-II-1 Bezpečné medziplanetárne cestovanie

Stručná myšlienka vzorového riešenia:

Zo zadaného grafu odstránime nepoužiteľné vrcholy a hrany. V tom, čo nám ostalo, nájdeme mosty. Navyše si vhodne predpočítame, ako sú rozložené vrcholy typu 2, aby sme pre každý most vedeli povedať, či sú všetky na tej istej jeho strane (vtedy je kritický) alebo nie.

Spoločný úvod pre všetky riešenia:

Keďže nás zaujímajú bezpečné cesty a žiadna bezpečná cesta cez planétu typu 0 nevedie, tak ich budeme ignorovať. A tak isto budeme ignorovať teleporty, ktoré majú niektorý koniec na takejto planéte. Najľahšie je zahodiť ich priamo pri načítaní vstupu.

Navyše ľahko spravíme jedno ďalšie predspracovanie vstupu: rozdelíme ho na komponenty súvislosti. Zjavne stačí každý komponent riešiť samostatne. V ďalšom texte riešenia budeme preto predpokladať, že spracúvaná sieť teleportov je súvislá.

Ešte si môžeme všimnúť, že ak niektorý komponent súvislosti obsahuje samé planéty typu 1, tak žiaden teleport v ňom nie je kritický – totiž ani teraz sa zo žiadnej z týchto planét nedá zachrániť. Taktiež žiadne kritické teleporty nemôže obsahovať komponent obsahujúci samé planéty typu 2 – tam sa nie je odkiaľ zachraňovať. Takéto komponenty teda môžeme rovno ignorovať, zjednoduší to neskôr rozbor prípadov.

Riešenie takmer hrubou silou:

Skúsme teraz nájsť algoritmus, ktorý pre daný teleport t povie, či je kritický. (Pritom predpokladáme, že komponent súvislosti, obsahujúci tento teleport, má v sebe aspoň jednu planétu typu 2.)

Ľahko môžeme vidieť, že pokiaľ obidva konce teleportu sú na planéte typu 2, tak teleport určite kritický nie je. Totiž keby nejaká zachraňujúca cesta viedla cez neho, tak určite existuje aj taká cesta, ktoré skončí už pred týmto teleportom.

Ak je na niektorom konci teleportu planéta p typu 1, tak potrebujeme overiť, či sa z nej aj po odstránení teleportu t dá zachrániť. To môžeme spraviť tak, že z planéty p spustíme prehľadávanie do hĺbky a zakážeme mu prejsť cez teleport

t. Ak sa toto prehľadávanie dostane na planétu typu 2, tak sa vieme zachrániť. (Ak teleport t viedol medzi dvomi planétami typu 1, postupne spustíme dve nezávislé prehľadávania. Ak sa pri čo len jednom z nich nevieme zachrániť, t je nutne kritický.)

Teraz si už len stačí uvedomiť, že pre žiadne iné planéty už nepotrebujeme overovať možnosť záchrany. Totiž keby odstránenie teleportu t pokazilo možnosť záchrany pre nejakú inú planétu q typu 1, tak určite pokazí aj možnosť záchrany pre planétu p typu 1, ktorá je „na tom istom konci“ teleportu t ako q. (Detaily tohto ľahkého dôkazu sporom prenechávame na čitateľa.)

Overenie kritickosti teleportu nás teda prinajhoršom stojí dve prehľadávania do hĺbky, čiže jeho časová zložitosť v sieti s n planétami a m teleportami bude $O(m+n)$. Keby sme takto overovali každý teleport, tak máme výsledný algoritmus s časovou zložitosťou $O((m+n)^2)$.

Rýchlejšie riešenie:

Majme ľubovoľnú súvislú sieť teleportov. Pozrime sa na konkrétny teleport. Ak je aj po jeho odstránení sieť teleportov súvislá (teda vieme prejsť z ľubovoľnej planéty na ľubovoľnú inú), tak dotyčný teleport evidentne nemôže byť kritický. Kritické teleporty stačí teda hľadať medzi tými, ktorých odstránenie rozpojí sieť teleportov na dve časti. V grafovej terminológii sa takéto hrany grafu volajú *mosty*.

Mostov v grafe nemôže nikdy byť príliš veľa: ak má graf n vrcholov, môže mať nanajvýš $n-1$ mostov. Súčasťou vzorového riešenia bude nájdenie všetkých mostov v zadanom grafe. Aj bez toho však vieme zlepšiť predchádzajúce riešenie. Na to nám bude stačiť, ak o väčšine hrán budeme vedieť povedať, že mostom *nie sú*.

Ako na to? Na našom súvislom grafe spustíme ľubovoľné prehľadávanie. Pre každý vrchol si zapamätáme hranu, ktorou sme ho počas prehľadávania prvýkrát objavili. Takto dostaneme jednu možnú *kostru* nášho grafu. (Kostra n-vrcholového grafu je strom tvorený $n-1$ jeho hranami.)

No a teraz si už len stačí uvedomiť, že každý most musí byť súčasťou práve nájdenej kostry. Totiž keď z nášho grafu vyhádžeme trebárs aj všetky ostatné hrany, stále bude držať pokope. Lenže kostrových hrán je len $n-1$. A tu sme práve ušetrili! Namiesto toho, aby sme ako kritický teleport testovali každú z m hrán, stačí nám ich otestovať $n-1$. Tento vylepšený algoritmus má teda časovú zložitosť $O(n(m+n))$.

Vzorové riešenie:

Ako sme si už ukázali, nutnou (ale nie postačujúcou!) podmienkou na to, aby teleport bol kritický, je, že musí byť mostom. No a most je kritickým teleportom vtedy a len vtedy, ak po jeho odobraní vzniknú dva komponenty: jeden ktorý obsahuje iba planéty typu 1, a druhý ktorý obsahuje aspoň jednu planétu typu 2.

Náš algoritmus najprv nájde všetky mosty a potom zistí, ktoré z nich sú kritické.

Na hľadanie mostov použijeme štandardný algoritmus, ktorý sa dá popísať nasledovne:

Z nejakého vrcholu pustíme prehľadávanie do hĺbky. Toto prehľadávanie nám hrany rozdelí na dva druhy: *stromové a spätné*. (Stromové hrany sú tie, ktorými sme objavili nejaký nový vrchol, spätné sú tie ostatné.)

Stromové hrany nám, ako u každého prehľadávania, vytvoria kostru nášho grafu. Pri prehľadávaní do hĺbky budeme tejto kostre hovoriť *DFS strom*. Rozmyslite si, že v DFS strome každá spätná hrana vedie medzi nejakým vrcholom a jeho predkom v DFS strome. (Túto vlastnosť napr. kostra určená prehľadávaním do šírky nemá!)

Už vieme, že mosty sú niektoré zo stromových hrán. Potrebujeme vedieť nejako identifikovať, ktoré z nich to sú a ktoré nie. Uvažujme teda nejakú stromovú hranu uv, ktorou sme objavili vrchol v. Kedy je táto hrana mostom? Vtedy a len vtedy, keď z podstromu s koreňom vo vrchole v žiadna spätná hrana nevedie von (teda do vrcholu u alebo jeho niektorého predka). Na základe tohto pozorovania teraz sformulujeme náš algoritmus.

Každému vrcholu x vieme priradiť hĺbku v DFS strome, označíme ju $h(x)$. Navyše pre každý vrchol spočítame nasledovnú veličinu $d(x)$: najmenšiu hĺbku, kam sa vieme dostať pomocou niekoľkých stromových hrán smerujúcich nadol a následne najviac jednej spätnej hrany.

Ak máme spočítané toto, tak mosty vieme nájsť nasledovne: Spätná hrana určite most nebude (keď ju odstránime, tak stále existuje cesta pomocou stromových hrán). Ak pre vrchol x, ktorý objavila stromová hrana, platí $h(x) > d(x)$, tak táto stromová hrana tiež nie je most (vieme ju nahradiť niekoľkými stromovými a spätnou). Ináč daná stromová hrana most je.

Zostáva popísať ako spočítame hodnoty $h(x)$ a $d(x)$. Hodnoty $h(x)$ vieme triviálne počítať pri objavovaní vrcholov. Hodnotu $d(x)$ spočítame ako minimum z hodnôt $d(x)$ synov vrcholu x v DFS strome a hĺbok vrcholov, kam vedú spätné hrany z vrcholu x.

Tento postup sa dá použiť priamo po načítaní grafu zo vstupu: postupne spúšťame prehľadávania do hĺbky, čím rozdelíme zadaný graf na komponenty

a zároveň v každom komponente nájdeme DFS strom a na ňom všetky mosty. Keďže každé prehľadávanie má časovú zložitosť priamo úmernú veľkosti príslušného komponentu, má táto časť riešenia celkovú časovú zložitosť $O(m+n)$.

Teraz ešte potrebujeme spočítať podmienky pre kritickosť teleportu. Počas prehľadávania do hĺbky pre každý vrchol x spočítame dve hodnoty $c_1(x)$ a $c_2(x)$: počty vrcholov typu 1 a typu 2 v podstrome, ktorý visí pod ním v DFS strome (vrátane samotného vrcholu x). Keď takto spracujeme celý komponent, v koreni jeho DFS stromu dostaneme celkové počty C_1 a C_2 vrcholov typu 1 a 2 v ňom. Ak $C_1 = 0$ alebo $C_2 = 0$, komponent môžeme odignorovať. Inak platí, že most je kritický, ak sú všetky vrcholy typu 2 na tej istej jeho strane – teda ak platí ($c_2(x) = 0$ alebo $c_2(x) = C_2$).

Celé riešenie má zjavne optimálnu časovú aj pamäťovú zložitosť $O(m+n)$.

Listing programu (C++)

```
#include <cstdio>
#include <vector>
#include <algorithm>
using namespace std;
#define INF 1234567890

struct Vrchol {
  Vrchol() : depth(-1), best(INF), c1(0), c2(0) {}
  vector<int> next;
  int type, depth, best, c1, c2;
};

// Vrcholy grafu
vector<Vrchol> v;
// Kandidati na mosty: (index hrany, index vrcholu kam vedie)
vector<pair<int, int> > mosty;

// Prehlada vrchol x, vrati trojicu best, c1, c2 daneho vrchola,
// pripadne depth, 0, 0 ak sme tam uz boli
pair<int, pair<int, int> > dfs(int x, int depth, int odkial) {
  if (v[x].depth != -1) return make_pair(v[x].depth, make_pair(0, 0));
  v[x].depth = depth;
  for (int i = 0; i < v[x].next.size(); i++) {
    if (v[x].next[i] == odkial) continue;
    pair<int, pair<int, int> > ret = dfs(v[x].next[i], depth+1, x);
    v[x].best = min(v[x].best, ret.first);
    v[x].c1 += ret.second.first;
    v[x].c2 += ret.second.second;
  }
  if (v[x].type == 1) v[x].c1++;
  if (v[x].type == 2) v[x].c2++;
  if (v[x].best >= v[x].depth && depth != 0) mosty.push_back(make_pair(odkial,x));
  return make_pair(v[x].best, make_pair(v[x].c1, v[x].c2));
}

int main() {
  int n, m;
  scanf("%d %d", &n, &m);
```

```
  v.resize(n);
  for (int i = 0; i < n; i++) scanf("%d", &v[i].type);
  for (int i = 0; i < m; i++) {
    int a, b; scanf("%d %d", &a, &b); a--; b--;
    if (v[a].type == 0) continue;
    if (v[b].type == 0) continue;
    v[a].next.push_back(b);
    v[b].next.push_back(a);
  }
  for (int i = 0; i < n; i++) {
    mosty.clear();
    dfs(i, 0, -1);
    int C1 = v[i].c1,  C2 = v[i].c2;
    for (int j = 0; j < mosty.size(); j++) {
      if (v[mosty[j].second].c1 > 0 && v[mosty[j].second].c2 == 0 && C2 > 0) {
        printf("%d %d\n", mosty[j].first+1, mosty[j].second+1);
      }
      if (C1 - v[mosty[j].second].c1 > 0 && C2 - v[mosty[j].second].c2 == 0
          && v[mosty[j].second].c2 > 0) {
        printf("%d %d\n", mosty[j].first+1, mosty[j].second+1);
      }
    }
  }
}
```

A-II-2 Vlámačka à la Banach-Tarski

Na začiatok na chvíľu zabudnime na možnosť používať duplikátor. Dostaneme tak úlohu, ktorá je vo svete známa pod názvom *0-1 knapsack problem* – chceme vybrať niektoré z modelov tak, aby súčet ich hmotností nepresiahol m a zároveň sa snažíme maximalizovať súčet ich cien.

Táto úloha je NP-úplná, teda nepoznáme algoritmus, ktorý by ju riešil v čase polynomiálnom od počtu modelov na vstupe. V našom prípade však máme v zadaní zaručené, že hmotnosti jednotlivých modelov sú celé kladné čísla a že celková Robertova nosnosť je rozumne malá. To nám umožní použiť pseudopolynomiálny algoritmus založený na myšlienke dynamického programovania so zložitosťou $O(mn)$.

Definujme si $P[i, j]$ ako najvyššiu cenu, ktorú vieme získať ukradnutím niektorých modelov, ak máme na výber len z prvých i modelov na vstupe a súčet hmotností tých, ktoré vyberieme, môže byť najviac j. Riešením, ktoré chceme nájsť, je potom hodnota $P[n, m]$.

Hodnoty $P[0, j]$ sú pre všetky j rovné 0, keďže nemáme k dispozícii nijaký model.

Pre $i \geq 1$ vypočítame $P[i, j]$ takto: Označme si hmotnosť i-teho modelu w_i a jeho cenu c_i. Ak tento model nevezmeme, najlepšia cena, ktorú vieme dosiah-

nuť, je $P[i-1,j]$. A najlepšia cena, ktorú vieme dosiahnuť, ak ho vezmeme, je $c_i + P[i-1, j-w_i]$. To preto, lebo dostaneme cenu c_i za práve vybranú vec, a následne môžeme z prvých $i-1$ modelov vyberať len tak, aby sme nepresiahli hmotnosť $j - w_i$). Spomedzi týchto dvoch možností si samozrejme vyberieme tú lepšiu.[3] Hodnota $P[i,j]$ sa teda rovná $\max(\ c_i + P[i-1, j-w_i],\ P[i-1,j]\ )$.

Všimnite si, že na výpočet $P[i,j]$ nám stačí poznať len $P[i-1,j]$ a $P[i-1, j-w_i]$. Preto ak budeme počítať hodnoty P postupne od menších i k väčším, budeme už mať v čase výpočtu $P[i,j]$ potrebné hodnoty pripravené. Takto dostávame riešenie s časovou zložitosťou $O(mn)$ – každú z $O(mn)$ hodnôt $P[i,j]$ vypočítame v konštantnom čase.

Ak by sme si pamätali všetky hodnoty P, potrebovali by sme $O(mn)$ pamäte. To však nie je nutné – počas výpočtu hodnôt P pre prvých i modelov si stačí pamätať len hodnoty P pre prvých $i-1$ modelov. Naša vzorová implementácia si to dokonca pamätá v jedinom poli a postupne od väčších j k menším nahrádza hodnoty $P[i-1,j]$ hodnotami $P[i,j]$. (Rozmyslite si, že si nikdy neprepíšeme informáciu, ktorú by sme ešte neskôr potrebovali.) Pamäťová zložitosť je teda $O(m)$.

Pomalšie riešenie pôvodnej úlohy:

Teraz už priamo vieme navrhnúť funkčné a vcelku efektívne riešenie pôvodnej úlohy. Stačí vyskúšať všetkých n možností, na ktorú vec použiť duplikátor. Akonáhle duplikátor použijeme, stojíme pred obyčajnou úlohou 0-1 knapsack s $n+1$ modelmi. Stačí nám teda n-krát vyriešiť 0-1 knapsack a spomedzi všetkých nájdených riešení si vybrať to najlepšie. Takéto riešenie má časovú zložitosť $O(n^2m)$ a pamäťovú zložitosť $O(m+n)$.

Vzorové riešenie pôvodnej úlohy:

Rýchlejšie riešenie úlohy s duplikátorom dostaneme vhodným rozšírením dynamického programovania, ktoré používame v zjednodušenej úlohe. Definujme si $Q[i,j]$ ako najvyššiu cenu, ktorú vieme získať ukradnutím niektorých modelov, ak máme na výber len z prvých i modelov na vstupe, súčet ich hmotností môže byť najviac j a navyše môžeme raz použiť duplikátor.

Hodnoty $Q[0,j]$ sú, rovnako ako $P[0,j]$, rovné 0.

Pozrime sa teraz, ako vyjadriť $Q[i,j]$ pre $i \geq 1$. Znova si označme hmotnosť i-teho modelu w_i a jeho cenu c_i. Oproti výpočtu $P[i,j]$ nám pribudla tretia

[3] Drobný technický detail: Na výber máme len vtedy, ak daný model ešte unesieme, teda ak $w_i \leq j$. V opačnom prípade nám ostáva jediná možnosť – i-ty model nevziať. V programe toto ošetríme vhodnou podmienkou.

možnosť: tento model môžeme zduplikovať a zobrať obe kópie. Takto získame riešenie, ktorého optimálna cena bude $2c_i + P[i-1, j-2w_i]$. (Spomedzi zvyšných $i-1$ modelov chceme vybrať najdrahšiu podmnožinu, ktorej hmotnosť je nanajvýš $j-2w_i$. Duplikátor sme už použili, preto pri výbere týchto $i-1$ modelov ho už použiť nesmieme.) Hodnotu $Q[i, j]$ teda vypočítame nasledovne:

$$Q[i,j] = \max\Big(\quad Q[i-1,j],\quad c_i + Q[i-1, j-w_i],\quad 2c_i + P[i-1, j-2w_i]\quad\Big)$$

Súčasťou úlohy bolo aj vypísať číslo modelu, ktorý sme v optimálnom riešení zduplikovali. Preto si pri výpočte $Q[i, j]$ pre každé i a j zapamätáme aj toto číslo.

Toto riešenie má časovú zložitosť $O(mn)$, teda rovnakú ako riešenie úlohy bez duplikátora. A opäť vieme dosiahnuť pamäťovú zložitosť $O(m)$: stačí postupne pre všetky i vhodne striedavo počítať hodnoty P a Q.

Listing programu (C++)

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

int main() {
    int n, m; cin >> m >> n;
    vector<int> P(m + 1, 0);
    // v Q mame dvojice (najvyssia cena, cislo zduplikovaneho modelu)
    vector<pair<int, int> > Q(m + 1, make_pair(0, 0));
    for (int i = 0; i < n; ++i) {
        int w, c;
        cin >> w >> c;
        for (int j = m; j >= w; --j) {
            if (P[j - w] + c > P[j])
                P[j] = P[j - w] + c;
            if (Q[j - w].first + c > Q[j].first)
                Q[j] = make_pair(Q[j - w].first + c, Q[j - w].second);
            if (j >= 2 * w && P[j - 2 * w] + 2 * c > Q[j].first)
                Q[j] = make_pair(P[j - 2 * w] + 2 * c, i + 1);
        }
    }
    if (Q[m].first > P[m]) cout << "zduplikuj model " << Q[m].second << endl
                                << "najlepsia cena " << Q[m].first << endl;
    else                   cout << "nezduplikuj nic" << endl
                                << "najlepsia cena " << P[m] << endl;
}
```

Iné vzorové riešenie:

Predchádzajúce riešenie by sa dalo efektívne rozšíriť, ak by sme mali povolené použiť duplikátor nie raz, ale s-krát. Namiesto polí P a Q by sme mali polia P_0, P_1, …, P_s, pričom $P_k[i, j]$ by sme si definovali ako najvyššiu cenu,

ktorú vieme získať ukradnutím niektorých modelov, ak máme na výber len z prvých i modelov na vstupe, súčet ich hmotností môže byť najviac j a duplikátor môžeme použiť najviac k-krát.

V našom prípade však existuje aj jednoduchší prístup. Vypočítame len hodnoty P z predchádzajúceho riešenia. Bez duplikovania by najlepším riešením bola hodnota $P[n, m]$. Ak by sme zduplikovali i-ty model a duplikát ukradli, budeme mať pred sebou pôvodných n vecí, pričom si z nich môžeme nabrať do hmotnosti $m - w_i$. Najvyššia dosiahnuteľná cena by teda bola $c_i + P[n, m - w_i]$. Celkovo najlepšie riešenie nájdeme vyskúšaním všetkých možností čo zduplikovať. Časová zložitosť bude znova $O(mn)$, pamäťová zložitosť $O(m + n)$.

A-II-3 Parazity

V celom riešení predpokladáme, že platí $d \geq n$. V opačnom prípade (keď je typov báz viac ako dĺžka DNA, ktorú spracúvame), je totiž zjavne odpoveď 0.

Prvé riešenie, ktoré nám napadne skoro hneď, je použiť hrubú silu: Pre každý možný súvislý úsek si zistíme, či sa parazitom páči alebo nie. Toto vieme ľahko urobiť s časovou zložitosťou $O(d^3)$ tak, že si zvolíme začiatok a koniec (ktorých je rádovo d^2) a zakaždým pomocou jedného prechodu práve skúšaného úseku zistíme, či je tam každá báza aspoň raz.

Toto riešenie sa však dá veľmi jednoducho vylepšiť. Označme si $p(i)$ prvú takú pozíciu, že úsek i až $p(i)$ je parazitmi obľúbený. Potom pre každé $j > p(i)$ platí, že úsek i až j je tiež parazitmi obľúbený. Toto je celkom ľahké ukázať. Keďže na pozíciách i až $p(i)$ sa objavili všetky možné typy báz, tak sa tým skôr všetky typy báz vyskytujú v úseku od i po j. Stačí teda, keď pre každý možný začiatok i nájdeme jemu zodpovedajúci najľavejší možný koniec $p(i)$.

Ako to spraviť efektívne? Zoberieme si pole veľkosti n, do ktorého si budeme pre každú bázu značiť, či sme ju už videli. A navyše budeme mať premennú `pocet`, v ktorej si budeme pamätať, koľko *typov* báz sme už videli. Pridanie novej bázy x do práve spracúvaného úseku teraz vyzerá nasledovne: Pozrieme do poľa, či sme už bázu x videli. Ak áno, tak sa nič nezmenilo oproti predchádzajúceme úseku. Ak sme ju ešte nevideli, tak si ju v poli zaznačíme ako už videnú a zvýšime premennú `pocet`. Akonáhle `pocet` dosiahne n, našli sme pre práve skúšaný začiatok najľavejší možný koniec. Práve popísané riešenie má časovú zložitosť $O(d^2)$.

Vzorové riešenie:

Vyššie popísané riešenie však stále nie je optimálne. K optimálnemu riešeniu nám chýba ešte jedno pozorovanie: Ak $i < j$ tak $p(i) \leq p(j)$. Toto sa dá ukázať sporom. Ak by platilo, že $p(j) < p(i)$, tak to znamená, že na úseku j až $p(j)$ sa objavili všetky typy báz. Lenže toto je len nejaká časť úseku i až $p(i)$, a teda aj na úseku i až $p(j)$ sú všetky typy báz, a to je spor s tým, že $p(i)$ je najmenší taký index, pre ktorý platí, že i až $p(i)$ je parazitmi obľúbený.

Ako toto pozorovanie využiť? Hovorí nám vlastne, že keď už poznáme hodnotu $p(i)$, tak pri spracúvaní úseku začínajúceho na pozícii $i + 1$ stačí jeho koniec hľadať od pozície $p(i)$ ďalej. Lenže na to by sme potrebovali vedieť, ktoré bázy sa nachádzajú v úseku od pozície $i + 1$ po pozíciu $p(i)$. A na to nám pole boolovských premenných, ktoré sme použili v predchádzajúcom riešení, nestačí – nevieme z neho povedať, či sa báza z pozície i nachádza ešte niekde medzi pozíciami $i + 1$ až $p(i)$.

Tento nedostatok ale ľahko odstránime: namiesto boolovských premenných použijeme celočíselné. V nich si budeme pre každú bázu pamätať, *koľkokrát* sa nachádza v práve spracúvanom úseku. V nižšie uvedenom programe ide o pole `vyskyty`. Naďalej budeme používať premennú `pocet`, v ktorej budeme mať uložený počet rôznych typov báz v práve spracúvanom úseku.

Začneme tým, že si vyrátame hodnotu $p(1)$, teda najľavejší možný koniec, ak je začiatkom úseku prvý prvok na vstupe. V okamihu, kedy nájdeme hodnotu $p(1)$, máme poli `vyskyty` pre každý typ bázy jeho počet výskytov na pozíciách 1 až $p(1)$. Teraz posunieme začiatok na pozíciu 2. Tým nám z aktuálneho úseku vypadla báza z pozície 1. Zmenšíme teda hodnotu na príslušnom políčku poľa `vyskyty`. A ak sme ju zmenšili na 0, tak o jedno zmenšíme aj hodnotu v premennej `pocet`. A pokračujeme rovnako ako predtým: kým je `pocet` menej ako n, posúvame koniec aktuálneho úseku. Takto pokračujeme postupne pre všetky možné začiatky, až do chvíle, kým pre niektorý začiatok nezistíme, že už žiaden možný koniec nevyhovuje.

Nakoľko sme vlastne zlepšili časovú zložitosť? Stačí si uvedomiť, že v priebehu riešenia len meníme dva indexy: jeden, čo ukazuje na aktuálne skúšaný začiatok úseku, a jeden ukazujúci na jeho koniec. Posunutie každého indexu vieme spraviť v konštantnom čase. No a v priebehu celého riešenia každý z našich indexov prejde (nanajvýš) raz celú postupnosť. Preto je časová zložitosť tohto riešenia $O(d)$.

Listing programu (C++)

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

int main() {
    int n, d; cin >> n >> d;
    vector<int> vstup(d);
    for (int i=0; i<d; ++i) { cin >> vstup[i]; --vstup[i]; }

    vector<int> vyskyty(n,0); // pre kazdu bazu pocet vyskytov
    int pocet = 0;
    long long vysledok = 0;
    for (int zaciatok=0, p=0; zaciatok<d ; ++zaciatok) {
        while (p<d && pocet<n) {
            if (vyskyty[ vstup[p] ]==0) ++pocet;
            ++vyskyty[ vstup[p] ];
            ++p;
        }
        if (pocet==n) vysledok += d-p+1;
        --vyskyty[ vstup[zaciatok] ];
        if (vyskyty[ vstup[zaciatok] ]==0) --pocet;
    }
    cout << vysledok << endl;
}
```

Iné dobré riešenia:

Namiesto poľa `vyskyty`, ktoré budeme priebežne udržiavať, si môžeme nejaké údaje predpočítať. Napríklad si môžeme pre každú pozíciu x (od 1 po d) a každý typ bázy y (od 1 po n) predpočítať, kedy najbližšie za pozíciou x sa vyskytne báza y. Tieto informácie vieme priamočiaro predpočítať v čase $O(nd)$ a pomocou nich už ľahko implementujeme riešenie podobné vzorovému – vždy, keď posunieme začiatok úseku, nájdeme si, kde sa najbližšie vyskytuje báza, ktorá z neho práve vypadla.

To isté vieme spraviť ešte trochu šikovnejšie v čase $O(d)$, čím dostaneme iné optimálne riešenie. Stačí si napríklad pre každý typ bázy predpočítať zoznam pozícií, na ktorých sa nachádza. (A sú dva rôzne spôsoby, ako toto implementovať: buď môžeme mať n disjunktných zoznamov – jeden pre každú bázu – alebo jednoducho pomocné pole dĺžky d, kde je pre každú pozíciu i zaznačené, na ktorej nasledujúcej pozícii leží báza rovnaká ako na pozícii i.)

Iná technika návrhu efektívneho riešenia je použitie binárneho vyhľadávania. Pri takýchto riešeniach si najskôr predpočítame nejaké informácie, z ktorých vieme rýchlo povedať, či je konkrétny úsek vyhovujúci. (Napr. vyššie uvedené predpočítanie v $O(nd)$ má takúto vlastnosť, sú však aj lepšie spôsoby.) Následne pre každý z d možných začiatkov binárnym vyhľadávaním nájdeme jemu zodpovedajúci koniec. Najbežnejšie časové zložitosti takýchto riešení sú $O(d \log d)$, $O(d \log^2 d)$, prípadne $O(dn + d \log d)$.

A-II-4 Zlomkové programy

V oboch podúlohách budú riešenia založené na podobnom princípe ako v domácom kole: Na každé prvočíslo v rozklade aktuálnej hodnoty sa môžeme dívať ako na premennú. Teda ak je napríklad aktuálna hodnota rovná $2^4 3^5 5^2$, môžeme to čítať nasledovne: „v premennej #2 je uložená hodnota 4, v premennej #3 je uložená hodnota 5 a v premennej #5 je uložená hodnota 2“.

A ako prebieha krok výpočtu? Hľadáme vhodný zlomok – teda pozeráme sa na menovatele a pre každý z nich *otestujeme*, či sú v nejakých premenných dostatočne vysoké hodnoty. Napríklad zlomok s menovateľom $2^3 7$ môžeme použiť len vtedy, keď aktuálne máme v premennej #2 hodnotu aspoň 3 a v premennej #7 hodnotu aspoň 1. No a keď nájdeme vhodný zlomok, najskôr *znížime* hodnoty premenných zodpovedajúcich jeho menovateľu a potom *zvýšime* hodnoty iných premenných, zodpovedajúcich jeho čitateľu.

Podúloha a):

Zadanie tejto podúlohy si teraz môžeme preformulovať nasledovne: na začiatku sú v premenných #2 a #3 uložené vstupné hodnoty x a y. Naším cieľom je v premennej #5 vyrobiť výstupnú hodnotu $\max(x, y)$.

Slovne si riešenie tejto úlohy môžeme popísať nasledovne:

1. kým sú premenné #2 a #3 obe kladné, obe zníž a zvýš premennú #5
2. kým je premenná #2 kladná, zníž ju a zvýš premennú #5
3. kým je premenná #3 kladná, zníž ju a zvýš premennú #5

(Spomedzi krokov 2 a 3 sa vždy najviac jeden naozaj vykoná, keďže po kroku 1 ostala v aspoň jednej z premenných #2 a #3 nula.)

A tomuto slovnému popisu zodpovedá nasledujúci jednoduchý program:

$$\left(\frac{5}{6} \;,\; \frac{5}{2} \;,\; \frac{5}{3} \right)$$

Príklad výpočtu pre $n = 144 = 2^4 3^2$:

$$2^4 3^2 \quad \xrightarrow{1} \quad 2^3 3^1 5 \quad \xrightarrow{1} \quad 2^2 5^2 \quad \xrightarrow{2} \quad 2^1 5^3 \quad \xrightarrow{2} \quad 5^4$$

Príklad výpočtu pre $n = 729 = 2^0 3^6$:

$$3^6 \quad \xrightarrow{3} \quad 3^5 5 \quad \xrightarrow{3} \quad 3^4 5^2 \quad \xrightarrow{3} \quad 3^3 5^3 \quad \xrightarrow{3} \quad 3^2 5^4 \quad \xrightarrow{3} \quad 3^1 5^5 \quad \xrightarrow{3} \quad 5^6$$

Podúloha b):

Túto úlohu si môžeme preformulovať nasledovne: treba sčítať obsah premenných #2 a #3, ale zároveň zachovať obsah týchto premenných. Ak by nebolo treba zachovať vstupné premenné, bolo by riešenie tejto úlohy jednoduché:

1. kým je premenná #2 kladná, zníž ju a zvýš premennú #5
2. kým je premenná #3 kladná, zníž ju a zvýš premennú #5

Aby sme nestratili pôvodný obsah premenných, budeme vždy naraz zvyšovať dve premenné: aj premennú #5, aj novú pomocnú premennú. Začiatok nášho algoritmu bude teda vyzerať takto:

1. kým je premenná #2 kladná, zníž ju, zvýš premennú #5 a zvýš #43
2. kým je premenná #3 kladná, zníž ju, zvýš premennú #5 a zvýš #47

A následne už len „upraceme“ – obsah premenných #43 a #47 vrátime späť:

3. kým je premenná #43 kladná, zníž ju a zvýš premennú #2
4. kým je premenná #47 kladná, zníž ju a zvýš premennú #3

Ak by sme ale priamo podľa tohoto algoritmu napísali zlomkový program, nastali by podobné problémy ako v riešení druhej podúlohy domáceho kola: program by nikdy neskončil. Akonáhle by sa vykonal krok 3, bola by premenná #2 opäť kladná, opäť by sa vykonal krok 1, a tak dokola.

My potrebujeme zabezpečiť, aby sa pri vykonávaní nášho zlomkového programu kroky 1 až 4 vykonali len raz, a to presne v tomto poradí.

A na to využijeme hodnotu 7, ktorou je zaručene deliteľné vstupné číslo. Zlomkový program navrhneme tak, aby počas výpočtu bola v každom okamihu práve jedna z premenných #7, #11, #13 a #17 nenulová. A podľa toho, ktorá z nich to je, budeme vykonávať ďalší krok výpočtu.

V spresnenom algoritme už teda nemusíme číslovať jednotlivé kroky, ich poradie bude jednoznačne určené. Nový, presnejší popis nášho algoritmu vyzerá nasledovne:

- kým je premenná #7 kladná:
  - ak je premenná #2 kladná, zníž ju, zvýš #5 a zvýš #43
  - ak je premenná #3 kladná, zníž ju, zvýš #5 a zvýš #47
  - ak sú premenné #2 aj #3 nulové, vynuluj #7, nastav #13 na 1

- kým je premenná #13 kladná:
  - ak je premenná #43 kladná, zníž ju a zvýš #2
  - ak je premenná #47 kladná, zníž ju a zvýš #3
  - ak sú premenné #43 aj #47 nulové, vynuluj #13 a tým končíme

Premenné #11 a #17 budú nášmu programu slúžiť ako pomocné pri kontrole, či je premenná #7, resp. #13, kladná. Napr. vždy, keď aktuálnu hodnotu (v krokoch 1 a 2) vydelíme 7, zároveň ju vynásobíme 11. A následne použijeme zlomok 7/11, aby sme opäť dostali hodnotu deliteľnú 7.

Výsledný program:

$$\left(\frac{5\cdot 43\cdot 11}{2\cdot 7}\,,\ \frac{5\cdot 47\cdot 11}{3\cdot 7}\,,\ \frac{7}{11}\,,\ \frac{13}{7}\,,\ \frac{2\cdot 17}{43\cdot 13}\,,\ \frac{3\cdot 17}{47\cdot 13}\,,\ \frac{13}{17}\,,\ \frac{1}{13} \right)$$

Príklad výpočtu pre $n = 2^1 \cdot 3^2 \cdot \mathbf{7}$:

$$\begin{aligned} 2^1\cdot 3^2\cdot \mathbf{7} &\xrightarrow{1} 3^2\cdot 5^1\cdot \mathbf{11}\cdot 43^1 \xrightarrow{3} 3^2\cdot 5^1\cdot \mathbf{7}\cdot 43^1 \xrightarrow{2} 3^1\cdot 5^2\cdot \mathbf{11}\cdot 43^1\cdot 47^1 \\ &\xrightarrow{3} 3^1\cdot 5^2\cdot \mathbf{7}\cdot 43^1\cdot 47^1 \xrightarrow{2} 5^3\cdot \mathbf{11}\cdot 43^1\cdot 47^2 \xrightarrow{3} 5^3\cdot \mathbf{7}\cdot 43^1\cdot 47^2 \\ &\xrightarrow{4} 5^3\cdot \mathbf{13}\cdot 43^1\cdot 47^2 \xrightarrow{5} 2^1\cdot 5^3\cdot \mathbf{17}\cdot 47^2 \xrightarrow{7} 2^1\cdot 5^3\cdot \mathbf{13}\cdot 47^2 \\ &\xrightarrow{6} 2^1\cdot 3^1\cdot 5^3\cdot \mathbf{17}\cdot 47^1 \xrightarrow{7} 2^1\cdot 3^1\cdot 5^3\cdot \mathbf{13}\cdot 47^1 \\ &\xrightarrow{6} 2^1\cdot 3^2\cdot 5^3\cdot \mathbf{17} \xrightarrow{7} 2^1\cdot 3^2\cdot 5^3\cdot \mathbf{13} \xrightarrow{8} 2^1\cdot 3^2\cdot 5^3 \end{aligned}$$

(Tučným písmom je všade vysádzané prvočíslo reprezentujúce aktuálny „stav výpočtu“. Ostatné prvočísla majú vždy uvedený exponent, aj ak je rovný jednej – tieto exponenty sú hodnoty dotyčných premenných.)

Trochu iné riešenie: V poslednej fáze programu už nepotrebujeme mať špeciálne prvočíslo, ktoré nám hovorí, aké zlomky môžeme používať – poslednú fázu totiž môžeme spoznať jednoducho tak, že sa nedá použiť žiadny zlomok z predchádzajúcich fáz. Toto pozorovanie vedie k o niečo jednoduchšiemu a stručnejšiemu programu:

$$\big(\ (5.43.11)/(2.7),\ (5.47.11)/(3.7),\ 7/11,\ 1/7,\ 2/43,\ 3/47\ \big)$$

Riešenia krajského kola kategórie B

B-II-1 Letenka

Pomalšie ale funkčné riešenia:

Jednou možnosťou, ako úlohu vyriešiť, je vyskúšať všetky lety, potom všetky dvojice letov a nakoniec všetky trojice letov. Z nich vyberieme len tie, ktoré zodpovedajú našim požiadavkam. (Lety musia na seba nadväzovať a musí sedieť začiatočné aj koncové letisko.) Spomedzi tých, ktoré vyhovujú, si vyberieme najlacnejšiu možnosť. Takto postupne nájdeme najskôr najlacnejší priamy let, potom najlacnejší let na jeden prestup a potom najlacnejší let na dva prestupy.

Takéto riešenie má časovú zložitosť až kubickú od počtu letov, teda $O(\ell^3)$. A zjavne nie je optimálne – robíme pri ňom veľa zbytočnej práce. Napríklad keď sme si vybrali prvý let z Viedne do Mníchova, nemá zmysel pre druhý let skúšať všetky možné lety po celom svete. Omnoho šikovnejšie je skúsiť len lety z Mníchova. A keď sme si už odtiaľ vybrali druhý let do Frankfurtu a našim cieľom je Zurich, pre tretí let máme ešte lepšiu situáciu. Nemusíme predsa prezerať všetky lety z Frankfurtu, keď jediné, čo potrebujeme, je najlacnejší priamy let do Zurichu.

Tieto pozorovania nás privádzajú k záveru, že si potrebujeme lety uložiť v pamäti tak, aby sme vedeli efektívne robiť dva typy operácii: O1: „nájsť všetky lety z daného letiska“ a O2: „pre danú dvojicu letísk povedať, či sú spojené priamym letom, a ak áno, akým najlacnejším“. Takýto spôsob uloženia údajov existuje, ale je zbytočne komplikovaný, preto nebudeme zachádzať do detailov.

Iný možný prístup vedúci k trocha horšiemu riešeniu: Stačí nám uložiť informácie o letoch tak, aby sme vedeli robiť operáciu O2. Potom najlacnejší let z A do B na dva prestupy nájdeme tak, že vyskúšame všetky možné prestupové letiská C a D a zakaždým si zistíme najlacnejšiu cenu cesty $A \to C \to D \to B$. Ak si počet rôznych letísk na vstupe označíme n, vieme o tomto riešení povedať, že rádovo n^2-krát budeme potrebovať pre nejakú dvojicu letísk spraviť operáciu O2. Čím lepšie ju budeme vedieť robiť, tým rýchlejšie toto riešenie bude.

Pri všetkých týchto postupoch ešte stále skúšame aj niektoré možnosti, ktoré by sme skúšať nemuseli. Napríklad v poslednom uvedenom riešení nemá zmysel skúšať všetky dvojice prestupných letísk (C, D): Akonáhle vieme, že medzi A

a C nevedie priamy let, nemusíme skúšať žiadne D. A pre konkrétne C stačí skúšať tie D, do ktorých sa z C vieme dostať priamym letom.

Riešenie v čase priamo úmernom počtu letov:

V prvom rade si môžeme uvedomiť, že keďže každé letisko má trojpísmenný identifikátor, je len $26^3 = 17576$ možností pre názov letiska. Teda nie je problém spraviť si pole veľkosti 26^3 a v ňom si niečo uložiť o každom z letísk, ktoré sa na vstupe vyskytnú.

Začneme tým, že načítame začiatočné letisko A a cieľové letisko B. Následne načítame celý zoznam letov do jedného poľa. A už počas načítavania letov (alebo hneď po ňom) si viem vyplniť dve ďalšie polia: pre každé letisko C cenu najlacnejšieho letu $A \to C$, a pre každé letisko D cenu najlacnejšieho letu $D \to B$.

Akonáhle teda skončilo načítanie, viem už cenu najlacnejšieho priameho letu $A \to B$ (mám ju uloženú hneď na dvoch miestach).

Následne spočítam cenu najlacnejšieho letu s jedným prestupom: pre každé letisko C, ktoré sa na vstupe aspoň raz vyskytlo, sa pozriem na súčet cien najlacnejšieho letu $A \to C$ a najlacnejšieho letu $C \to B$.

Na záver spočítam cenu najlacnejšieho letu s dvomi prestupmi: Prejdem celý zoznam letov. Pre každý let vyskúšam možnosť: „čo ak je toto optimálny let z C do D?“ Zistím teda cenu práve skúšaného letu, plus cenu letu z A do miesta, kde práve skúšaný let začína, plus cenu letu z miesta, kde práve skúšaný let končí, do B.

Listing programu (Pascal)

```
const NEKONECNO = 987654321;
      POCET_LETISK = 26*26*26;

type let = record odkial, kam, cena : longint; end;
var zaciatok, koniec : longint;
    lety : array of let;
    L : longint;

{ pomocna funkcia: skonvertuje trojpismenovy retazec na cislo z [0,26*26*26) }
function retazec_na_cislo(s : string) : longint;
begin retazec_na_cislo := 26*26*(ord(s[1])-65)+26*(ord(s[2])-65)+ord(s[3])-65; end;

{ pomocna procedura: zmensi hodnotu v premennej, ak vies }
procedure vylepsi(var premenna : longint; nova_hodnota : longint);
begin if nova_hodnota < premenna then premenna := nova_hodnota; end;

{ procedura na nacitanie kodu letiska a vyroby cisla letiska z neho }
procedure nacitaj_letisko(var kod : longint);
var c : char;
    letisko : string;
begin
    letisko := '';
```

```
    read(c); letisko += c; read(c); letisko += c; read(c); letisko += c;
    read(c); { aj whitespace za nazvom nacitame }
    kod := retazec_na_cislo(letisko);
end;

procedure nacitaj_vstup;
var i : longint;
begin
    nacitaj_letisko(zaciatok); nacitaj_letisko(koniec);
    readln(L);
    setlength(lety,L);
    for i:=0 to L-1 do begin
        nacitaj_letisko(lety[i].odkial); nacitaj_letisko(lety[i].kam);
        readln(lety[i].cena);
    end;
end;

procedure vypocitaj_vystup;
var najlepsi_prilet, najlepsi_odlet : array[0..POCET_LETISK-1] of longint;
    i, odpoved1, odpoved2, odpoved3 : longint;
begin
    for i:=0 to POCET_LETISK-1 do najlepsi_prilet[i] := NEKONECNO;
    for i:=0 to POCET_LETISK-1 do najlepsi_odlet[i]  := NEKONECNO;

    { vypocitame ceny priamych letov zo zaciatku a do konca trasy }
    for i:=0 to L-1 do begin
        if lety[i].odkial = zaciatok then
           vylepsi( najlepsi_prilet[ lety[i].kam ],   lety[i].cena );
        if lety[i].kam    = koniec   then
           vylepsi( najlepsi_odlet[ lety[i].odkial ], lety[i].cena );
    end;
    odpoved1 := najlepsi_odlet[zaciatok];

    { pre kazde letisko vyskusame letiet zo zaciatku na koniec cez neho }
    odpoved2 := odpoved1;
    for i:=0 to POCET_LETISK-1 do
        vylepsi( odpoved2, najlepsi_prilet[i]+najlepsi_odlet[i] );

    { a pre kazdy let vyskusame letiet zo zaciatku na koniec cez neho }
    odpoved3 := odpoved2;
    for i:=0 to L-1 do
        vylepsi( odpoved3,
           najlepsi_prilet[lety[i].odkial]+lety[i].cena+najlepsi_odlet[lety[i].kam]);

    { vypiseme odpovede }
    if odpoved1 >= NEKONECNO then writeln('neexistuje') else writeln(odpoved1);
    if odpoved2 >= NEKONECNO then writeln('neexistuje') else writeln(odpoved2);
    if odpoved3 >= NEKONECNO then writeln('neexistuje') else writeln(odpoved3);
end;

begin
    nacitaj_vstup;
    vypocitaj_vystup;
end.
```

Iné vzorové riešenie:

Ukážeme si ešte jedno riešenie s optimálnou časovou zložitosťou $O(\ell)$. Toto riešenie je myšlienkovo o trochu ťažšie, ale má dve nesporné výhody: veľmi ľahko sa implementuje a navyše ostane efektívne aj vtedy, keď by sme sa zaujímali o lety s tromi, štyrmi, či piatimi prestupmi.

Hlavná myšlienka: Ako vyzerá najlacnejší let z A do B s dvomi prestupmi? Tak, že najskôr s jedným prestupom prídeme z A na nejaké letisko C, a odtiaľ najlacnejším letom na letisko B. A ktorú možnosť použijeme pri ceste z A na C? No predsa najlacnejšiu.

Celé riešenie bude teraz úplne priamočiare: Trikrát prejdeme cez zoznam letov. Pri prvom prechode si pre *každé letisko* zistíme cenu najlacnejšieho priameho letu naň. Pri druhom prechode použijeme tieto údaje na to, aby sme pre každé letisko zistili cenu najlacnejšieho letu naň pomocou dvoch letov. A v treťom prechode spravíme ešte raz to isté: z cien pre dva lety spočítame ceny pre tri lety.

(Ide vlastne o jednoduchú aplikáciu postupu nazývaného *dynamické programovanie*. Navyše si pozorný čitateľ určite všimol, že v treťom prechode si už stačilo pamätať optimálnu cenu len pre naše cieľové letisko. Keďže nám to však nepokazí zložitosť, budeme si to pamätať pre všetky, bude sa to ľahšie implementovať. Časová zložitosť bude naďalej lineárna od počtu letov.)

Nasleduje mierne lenivá implementácia tohto postupu v C++: aby sme nemuseli prečíslovať letiská, použijeme namiesto toho hešovacie tabuľky.

Listing programu (C++)

```
#include <iostream>
#include <vector>
#include <unordered_set>
#include <unordered_map>
using namespace std;

const int NEKONECNO = 987654321;

struct let { string odkial, kam; int cena; };

string zaciatok, koniec;       // z ktoreho letiska a na ktore letime
vector<let> lety;              // zoznam letov zo vstupu
unordered_set<string> letiska; // zoznam letisk zo vstupu

// najlepsia_cena[x][YYY]:
// najlepsia cena, za ktoru vieme prist na letisko YYY pomocou x letov
unordered_map<string,int> najlepsia_cena[4];

void nacitaj_vstup() {
    cin >> zaciatok >> koniec;
    int L; cin >> L;
    while (L--) {
        let T; cin >> T.odkial >> T.kam >> T.cena;
        lety.push_back(T);
        letiska.insert(T.odkial); letiska.insert(T.kam);
    }
}

void inicializuj() {
    for (int let=0; let<=3; ++let)
        for (auto it : letiska)
```

```
            najlepsia_cena[let][it] = NEKONECNO;
    najlepsia_cena[0][zaciatok] = 0;
}

void vypocitaj() {
    for (int let=1; let<=3; ++let)
        // pre kazde letisko vieme najlepsi sposob na (let-1) letov,
        // chceme najst najlepsi na (let) letov
        for (unsigned i=0; i<lety.size(); ++i) {
            int takto = najlepsia_cena[let-1][lety[i].odkial] + lety[i].cena;
            if (takto < najlepsia_cena[let][lety[i].kam])
                najlepsia_cena[let][lety[i].kam] = takto;
        }
}

void vypis_vystup() {
    int odpoved1 = najlepsia_cena[1][koniec];
    if (odpoved1 == NEKONECNO) cout << "neexistuje" << endl;
    else cout << odpoved1 << endl;
    int odpoved2 = min( odpoved1, najlepsia_cena[2][koniec] );
    if (odpoved2 == NEKONECNO) cout << "neexistuje" << endl;
    else cout << odpoved2 << endl;
    int odpoved3 = min( odpoved2, najlepsia_cena[3][koniec] );
    if (odpoved3 == NEKONECNO) cout << "neexistuje" << endl;
    else cout << odpoved3 << endl;
}

int main() {
    nacitaj_vstup();
    inicializuj();
    vypocitaj();
    vypis_vystup();
}
```

B-II-2 Benátky

Pomalšie riešenie:

Jedno možné jednoduché riešenie vyzerá nasledovne: Samostatne pre každý počet zatopených poschodí p prejdeme celú ulicu a spočítame počet ostrovov. Presnejšie, keďže každý ostrov práve na jednom mieste začína, stačí spočítať všetky začiatky ostrovov. Začiatok ostrovu je taký dom, ktorý má na ľavej strane vodu (teda naľavo od neho je dom, ktorý už je celý pod vodou). Výnimkou je prvý dom: ak ešte nie je celý pod vodou, tak aj on je začiatkom ostrova.

V programe to ľahko implementujeme napr. tak, že ku výškam domov, ktoré si uložíme ako $H[1]$ až $H[n]$, doplníme „dom nulovej výšky“: $H[0] = 0$. Teraz pre každé $i = 1 \ldots n$ platí: Ak je zatopených presne p poschodí, tak dom i je začiatkom ostrova vtedy a len vtedy, ak $H[i] \geq p$ a zároveň $H[i-1] < p$.

Listing programu (Pascal)

```
var vysky: array[0..10000] of longint;
    pocet,voda,i,n:longint;
begin
    readln(n);
    for i:=1 to n do read(vysky[i]);
    vysky[0]:=0;
    for voda:=1 to n do begin
        pocet:=0;
        for i:=1 to n do if (vysky[i]>=voda) and (vysky[i-1]<voda) then inc(pocet);
        write(pocet, ' ');
    end;
    writeln;
end.
```

Takéto počítanie však nie je príliš efektívne – spravíme pri ňom spolu až rádovo n^2 krokov. Existujú však aj lepšie postupy. Jeden z nich si ukážeme.

Vzorové riešenie:

Budeme postupne simulovať dvíhanie sa hladiny. Dá sa všimnúť, že keď sa domy ponoria o jedno poschodie, nezmení sa toho príliš veľa. Presnejšie, zmena nastane len vtedy, keď sa strecha nejakého domu zaplaví. Vtedy sa možno nejak (najviac o 1) zmení počet ostrovov.

Vzorové riešenie teda bude nasledovné. Rozdelíme si domy do n chlievikov podľa ich výšky. (V prvom chlieviku budú jednoposchodové, v druhom dvojposchodové...) V cykle budeme postupne zvyšovať počet zatopených poschodí p od 1 po n. Vždy, keď zatopíme nové poschodie p, tak postupne po jednom „ponoríme“ domy v $(p-1)$-tom chlieviku, pričom si neustále pamätáme a upravujeme aktuálny počet ostrovov.

Ponoriť dom znamená, že ho akoby odstránime z ulice. T.j., do poľa boolovských premenných si o ňom zaznačíme, že už je celý pod vodou. Následne sa pozrieme, či sú už domy, ktoré s ním bezprostredne susedia, celé ponorené alebo nie. Mohli nastať tri prípady:

- Ak pred ním aj za ním už je voda, počet ostrovov sa zníži o 1.
  (Tento dom bol samostatný ostrov a ten práve zanikol.)
- Ak pred ním aj za ním je nepotopený dom, počet sa zvýši o 1.
  (Potopením tohto domu sa jeden ostrov rozpojil na dva.)
- Ak na jednej strane je voda a na druhej nepotopený dom, počet ostrovov sa nezmení.
  (Potopením tohto domu sa len zmenšila šírka jedného z ostrovov.)

Časová zložitosť algoritmu bude lineárna od počtu domov, teda $O(n)$. To preto, že rozdelenie domov do chlievikov vieme spraviť v lineárnom čase a následne už len každý dom raz spracujeme.

Listing programu (Pascal)

```
var vyska: array [1..1000007] of longint;
    vynoreny: array [0..1000008] of boolean;
    chlievik: array [0..1000007] of array of longint;
    pocet:array [0..1000007] of longint;
    n,i,voda,pocetOstrovov,cisloDomu:longint;
begin
    readln(n);
    { zistim velkosti chlievikov, nacitam vstup a nastavim pole vynoreny }
    for i:=0 to n do pocet[i] := 0;
    for i:=1 to n do begin
        read(vyska[i]);
        vynoreny[i] := true;
        inc(pocet[vyska[i]]);
    end;
    vynoreny[0] := false;
    vynoreny[n+1] := false;
    { podla zistenych velkosti nastavim velkosti chlievikov }
    for i:=0 to n do begin
        setlength(chlievik[i], pocet[i]);
        pocet[i] := 0;
    end;
    { naplnim chlieviky cislami budov}
    for i:=1 to n do begin
        chlievik[vyska[i],pocet[vyska[i]]]:=i;
        inc(pocet[vyska[i]]);
    end;
    { na zaciatku je jeden ostrov }
    pocetOstrovov := 1;
    for voda:=1 to n do begin
        { postupne zaplavujeme poschodia: pre domy so spravnou vyskou potopime }
        for i:=1 to pocet[voda-1] do begin
            cisloDomu := chlievik[voda-1,i-1];
            { vsimnite si, ze sa pocet ostrovov zmeni podla podmienok}
            dec(pocetOstrovov);
            if (vynoreny[cisloDomu-1]) then inc(pocetOstrovov);
            if (vynoreny[cisloDomu+1]) then inc(pocetOstrovov);
            vynoreny[cisloDomu] := false;
        end;
        write(pocetOstrovov,' ');
    end;
    writeln;
end.
```

B-II-3 Aničkine darčeky

Podúloha a):

Algoritmus, ktorý ste programovali pre Aničku, sa volá binárne vyhľadávanie (v angličtine binary search).

Stručne si pripomeňme hlavnú myšlienku, ktorá bola podrobnejšie popísaná v domácom kole: Máme usporiadanú postupnosť čísel, medzi ktorými chceme nájsť niektoré konkrétne. Tie si pamätáme v poli, prípadne ich vieme zistiť aj bez poľa (ako dátumy v decembri v domácom kole). Na začiatku hľadáme v intervale celého rozsahu poľa a postupne ho vhodne zmenšujeme na polovičnú veľkosť, až kým nehľadáme na intervale dĺžky 1 – teda kým nám neostal už len jediný kandidát.

Myšlienka binárneho vyhľadávania je síce ľahká, ale pri implementácii je veľmi ľahké urobiť kopu chýb „o plus mínus jedna“ a tak stvoriť program, ktorý dá občas nesprávnu odpoveď, či sa dokonca zacyklí. Týmto typickým chybám sa dá vyhnúť vhodným myšlienkovým prístupom, ktorý si teraz ukážeme.

Interval, v ktorom hľadáme, si budeme pamätať ako polootvorený interval, konkrétne $\langle z, k)$. Tento zápis znamená, že na pozícii z (začiatok) je prvá hodnota, ktorá už do intervalu patrí, a k (koniec) je prvá hodnota, ktorá už do intervalu nepatrí. Teda napr. polootvorený interval $\langle 5, 8)$ obsahuje celé čísla 5, 6 a 7, ale už nie 8.

Táto reprezentácia má oproti iným reprezentáciám (napr. uzavretý interval) veľa výhod. Skúste si rozmyslieť ako by ste vyjadrili dĺžku intervalu jedným a druhým spôsobom alebo ako by ste zapísali prázdny interval.

Ako tieto intervaly využiť pri binárnom vyhľadávaní? Jednoducho – rozdelíme si prvky v poli na dva typy: *zlé a dobré*. V našom prípade zlé prvky sú tie darčeky, ktoré sú prilacné, a dobré prvky sú tie darčeky, ktoré sú už dostatočne drahé. To, čo hľadáme, je prvý dobrý prvok v poli. Urobíme to tak, že nájdeme súčasne posledný zlý aj prvý dobrý prvok. Počas hľadania si budeme udržiavať vyššie spomenuté dve premenné z a k. Budeme si pritom dávať pozor, aby vždy platilo: prvok na pozícii z je zlý a prvok na pozícii k je dobrý.

Keďže pole je usporiadané, vieme, že všetky prvky naľavo od pozície z sú nutne tiež zlé. (Ak je siedmy darček príliš lacný, je aj každý z prvých šiestich darčekov príliš lacný.) A podobne, všetky prvky napravo od pozície k sú tiež dobré. Jedine prvky medzi z a k sú teda neznáme.

Ktorý prvok môže teda byť posledným zlým prvkom v poli? Do úvahy pripadajú práve pozície z polootvoreného intervalu $\langle z, k)$. V tejto chvíli je jasné, kedy binárneho vyhľadávanie skončí: vtedy, keď už ostane len jeden kandidát. Teda vtedy, keď $k - z = 1$.

Aj krok binárneho vyhľadávania je teraz priamočiary: pozrieme sa do stredu intervalu, na pozíciu[4] $m = \lfloor (z + k)/2 \rfloor$. Ak je na tejto pozícii dobrý prvok

[4]Symbolmi $\lfloor c \rfloor$ a $\lceil c \rceil$ značíme dolnú a hornú celú časť čísla c: $\lfloor c \rfloor$ je najväčšie celé číslo

(dostatočne drahý darček), môžeme zmeniť k na m. V opačnom prípade zmeníme z na m. V oboch prípadoch sme takto skrátili interval, v ktorom hľadáme, približne na polovicu.[5]

Posledné, čo potrebujeme vyriešiť, je inicializácia: na začiatku potrebujeme nastaviť hodnoty z a k tak, aby platila vyššie uvedená podmienka (odborne nazývaná *invariant*).

Majme teda v poli A na pozíciách 0 až $n-1$ uložené ceny darčekov od najmenšej po najväčšiu. Jedna možnosť je začať vyhľadávanie tým, že sa pozrieme na pozície 0 a $n-1$, či nenastal nejaký zo špeciálnych prípadov: všetky prvky zlé alebo naopak všetky dobré – v našom prípade teda všetky darčeky prilacné alebo všetky dostatočne drahé. Ak nastal špeciálny prípad, ošetríme ho, a ak nie, môžeme nastaviť $z = 0$, $k = n-1$ a spustiť binárne vyhľadávanie.

Ešte šikovnejšie je však pomôcť si drobným trikom: predstavme si, že pred poľom (teda na pozícii -1) je darček, ktorý je určite príliš lacný, a za poľom (na pozícii n) darček, ktorý je dostatočne drahý. Nastavíme teda $z = -1$, $k = n$ a rovno spustíme binárne vyhľadávanie.

A kedy teda spoznáme, či nenastal špeciálny prípad? To je jednoduché: úplne na konci. Ak náhodou skončíme s tým, že k ostalo rovné n, sú všetky skutočné darčeky príliš lacné, a teda vypíšeme -1. V opačnom prípade (a to vrátane situácie, kedy ostalo z rovné -1) ukazuje číslo k na pozíciu v poli, kde je prvý dobrý darček. A to je už úplne všetko.

Listing programu (Pascal)

```
{ ========= toto je funkcia, ktoru ste mali implementovat ======== }

function najdi_darcek(var A: array of longint; n : longint; p : longint) : longint;
var z, k, m : longint;
begin
    { inicializujeme z a k tak, aby ukazovali na urcite zly a urcite dobry darcek }
    z := -1;
    k := n;

    { kym mame viac ako jednu moznost, zmensujeme interval na polovicu }
    while k-z > 1 do begin
        m := (z+k) div 2;
        if A[m] < p then z := m else k := m;
    end;

    { zistime, ci mame riesenie, a ak ano, vratime ho }
    if k=n then najdi_darcek := -1 else najdi_darcek := A[k];
end;
```

ktoré je najviac c, a $\lceil c \rceil$ je najmenšie celé číslo, ktoré je aspoň c.

[5] Povšimnite si tiež, že krok robíme len vtedy, ak $k - z > 1$. V takomto prípade platí $z < m < k$, a teda určite spravíme aspoň nejaký pokrok aj pre malé intervaly – vždy pozeráme na nové, neznáme políčko a nikdy sa nezacyklíme.

```
{ ======== a takto by sa dala tato funkcia pouzit v programe ======== }

var pocet_darcekov : longint = 12;
    ceny_darcekov : array[0..11] of longint
       = (2, 3, 3, 3, 5, 8, 9, 10, 23, 32, 47, 99);
begin
    { vyskusame vo vyssie definovanom poli vyhladavat pre rozne ceny }
    writeln( najdi_darcek( ceny_darcekov, pocet_darcekov,   4 ) );   { vypise 5 }
    writeln( najdi_darcek( ceny_darcekov, pocet_darcekov,   5 ) );   { vypise 5 }
    writeln( najdi_darcek( ceny_darcekov, pocet_darcekov,  17 ) );   { vypise 23 }
    writeln( najdi_darcek( ceny_darcekov, pocet_darcekov, 100 ) );   { vypise -1 }
end.
```

Podúloha b):

Na koľko políčok poľa A sa naše riešenie pozrie? Vždy, keď nejaké políčko poľa A porovnáme s hodnotou p, zmenší sa počet možností, ktoré nám ostávajú, približne na polovicu. Mohli by sme sa teda pýtať otázku: Koľkokrát musíme interval danej dĺžky „rozpoliť“, kým nám ostane len jeden prvok? Pre lepšiu predstavu sa môžeme tú istú otázku opýtať aj opačným smerom: Koľkokrát musíme dĺžku intervalu zdvojnásobiť, aby z 1 narástla až na pôvodnú dĺžku?

Presne na túto otázku nám odpovedajú logaritmy. Logaritmus so základom 2 z čísla a zapíšeme ako $\log_2 a$. Hodnota $\log_2 a$ je také číslo b, že platí $2^b = a$. Napríklad $\log_2 8 = 3$, pretože $2^3 = 8$.

Naše riešenie teda spraví rádovo $\log_2 n$ krokov a v každom z nich sa pozrie na jedno políčko poľa A.

Príklad: Ak by pole A malo milión prvkov, pozrieme sa len na 20 z nich (lebo $\log_2 1\,000\,000 \approx 20$).

Podúloha c):

Dokážeme, že ľubovoľný *deterministický*[6] algoritmus riešiaci zadanú úlohu sa v najhoršom možnom prípade musí pozrieť na aspoň $\lceil \log_2(n+1) \rceil$ políčok poľa A.

Základná myšlienka dôkazu je jednoduchá: Existuje $n+1$ možných výstupov a každým prístupom do poľa A vieme zaručene vylúčiť najviac polovicu z nich, preto potrebný počet prístupov do poľa musí byť v najhoršom možnom prípade aspoň rovný dvojkovému logaritmu čísla $n+1$.

Vo zvyšku tohto vzorového riešenia tento dôkaz rozpíšeme poriadnejšie.

[6] Slovo „deterministický“ znamená, že ďalší krok algoritmu je vždy jednoznačne určený tým, čo sa udialo dovtedy. Inými slovami, je to algoritmus, ktorý sa nikdy nerozhoduje náhodne. Ona by mu tu tá náhoda aj tak nepomohla – keďže sa pozeráme na najhorší možný prípad, tak či tak by museli všetky možné postupy hľadania byť efektívne. A načo si potom náhodne vyberať jeden z nich?

V prvom rade si uvedomme, že keď sa algoritmus pozrie na nejaké políčko x poľa A, jediné rozumné, čo môže spraviť, je porovnať ho s hľadanou hodnotou p. Toto porovnanie nám dá jeden z dvoch možných výsledkov: buď sa dozvieme, že $A[x] < p$, teda dotyčný darček je prilacný, alebo sa dozvieme, že $A[x] \geq p$, teda dotyčný darček je dostatočne drahý.

Na začiatku ešte o poli A nič nevieme. To, čo chceme zistiť, je vlastne počet prvkov v poli A, ktoré sú menšie ako p. (Inými slovami, chceme zistiť polohu z posledného prilacného darčeka.) Je $n+1$ možností: prilacných darčekov môže byť 0, 1, 2, ..., až všetkých n. A ku každej z týchto $n+1$ možností patrí iný výstup algoritmu: v prvých n prípadoch vypíšeme cenu 1., 2., ..., n. darčeka, v poslednom prípade vypíšeme hodnotu -1.

Na začiatku sme teda v situácii, že do úvahy pripadá $n+1$ rôznych možností pre hodnotu z. Tieto možnosti budeme volať *kandidátmi*. Počas behu programu sa tento občas pozrie na niektoré políčko poľa A. V tomto okamihu sa nám množina kandidátov môže zmenšiť.

Príklad 1: Kandidátmi pre z boli čísla 4 až 11. Program sa pozrel na políčko 7 a bolo $A[7] < p$. Od tohto okamihu sú už kandidátmi len čísla od 7 po 11.

Príklad 2: Kandidátmi pre z boli čísla 4 až 11. Program sa pozrel na políčko 13 a samozrejme sa dozvedel, že $A[13] \geq p$. Jeho chyba, že sa tam pozeral, predsa to už mal vedieť. Množina kandidátov sa nezmenila.

No a už sme skoro hotoví. Stačí si len uvedomiť, že pri každom prístupe do poľa A sa množina kandidátov *rozdelí z jedného intervalu na dva*. Ak boli kandidátmi čísla od a po $b-1$ vrátane a opýtali sme sa na c, tak sa stane nasledovné: Ak $A[c] < p$, kandidátmi ostanú čísla c až $b-1$, inak kandidátmi ostanú čísla a až $c-1$. Ak bolo teda pred prístupom do poľa A kandidátov q, po prístupe je ich *v tej pre nás horšej vetve* aspoň $\lceil q/2 \rceil$.

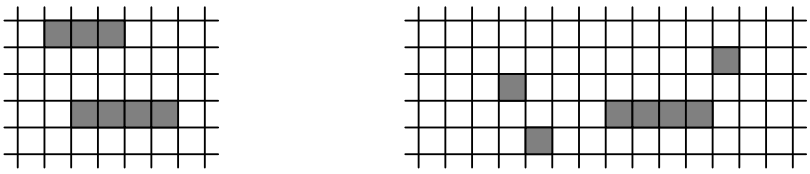
Nech teda 2^k je najväčšia mocnina dvoch ostro menšia ako $n+1$. (Čiže platí $2^k < n+1 \leq 2^{k+1}$.) Tvrdíme, že ak má pole A veľkosť n, neexistuje algoritmus, ktorému vždy stačí pozrieť sa na k políčok poľa A. Totiž na začiatku má pred sebou tento algoritmus $n+1$ kandidátov, a keď mu budeme na otázky odpovedať tak, aby zakaždým nastala tá pre neho horšia alternatíva, aj po k otázkach mu ešte ostanú aspoň dvaja kandidáti, a teda si nebude istý odpoveďou.

B-II-4 Čierne štvorce sú tu opäť

Podúloha a):

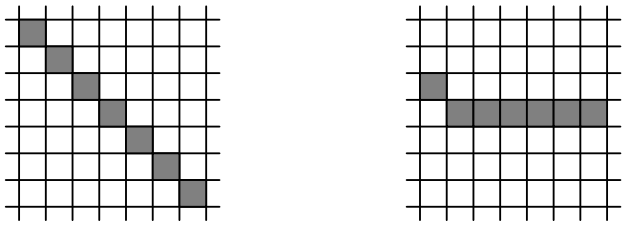
Prvé rozmiestnenie, ktoré sme mali nájsť, malo tvoriť 7 čiernych štvorcov, pre ktoré platí: O 3 minúty ešte bude aspoň jeden štvorec čierny, ale o 4 minúty už budú úplne všetky štvorce v celej rovine biele.

My už vieme vyrobiť rozmiestnenie 4 čiernych štvorcov, ktoré presne 3 minúty vydrží: stačí ich dať do radu vedľa seba. A keď ich máme mať na začiatku namiesto štyroch sedem, stačí ešte pridať tri iné niekde „dostatočne ďaleko“. Na nasledujúcom obrázku vidíme dve rôzne vyhovujúce rozmiestnenia. Treťou možnosťou bolo použiť riešenie podúlohy d) domáceho kola pre $n = 7$ a $k = 4$.



Druhé hľadané rozmiestnenie malo tvoriť 7 čiernych štvorcov, pre ktoré platí: O 6 minút bude práve jeden štvorec v celej rovine čierny. Tento štvorec nesmie ležať na žiadnej z pozícií, kde boli čierne štvorce na začiatku.

Najjednoduchšie je všimnúť si, čo sa stane, ak čierne štvorce tvoria rad idúci „šikmo doprava dodola“. Ten bude postupne postupovať doprava dohora a skracovať sa. Toto riešenie je na nasledujúcom obrázku vľavo. Na obrázku vpravo je iné riešenie tejto podúlohy. Skúste si odsimulovať jeho prvý krok a uvidíte, ako funguje.

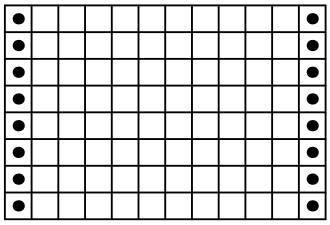


Podúloha b):

Štvorce na obvode Marikinho papiera ostanú navždy biele.

Uvedomte si, že na to, aby sme to dokázali, stačí dokázať, že ak niekedy je celý obvod Marikinho papiera biely, tak aj o minútu neskôr bude celý biely.

Všimnime si najskôr tie štvorce na zvislých okrajoch (teda tie, ktoré sú zvýraznené na nasledujúcom obrázku):



Pre každý z nich platí, že aj on je biely, aj štvorec bezprostredne pod ním je biely. A teda všetky tieto štvorce budú biele aj v nasledujúcom kroku. A rovnakú úvahu môžeme spraviť aj pre štvorce na vodorovných hranách – každý z nich je biely a má naľavo od seba biely štvorec, preto aj v nasledujúcej minúte bude biely.

Podúloha c):

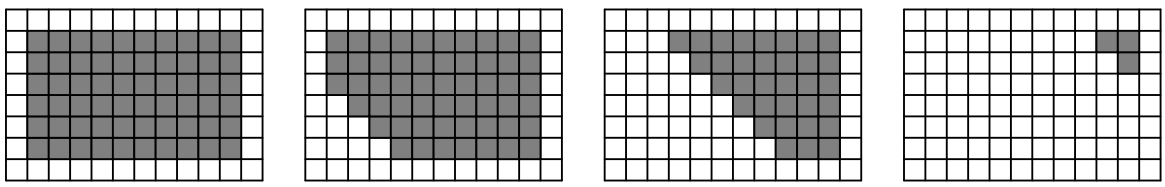
Žiadne začiatočné zafarbenie nevydrží ani len 20 minút, nie to ešte 100. Ukážeme dva rôzne (aj keď podobné) spôsoby, ako toto tvrdenie dokázať.

Prvý dôkaz bude založený na nasledujúcom pozorovaní: Predstavme si dve rôzne štvorcové siete. V prvej označme množinu čiernych štvorcov $\mathcal{A}$, v druhej $\mathcal{B}$. Nech $\mathcal{A} \subseteq \mathcal{B}$, teda každý štvorec, ktorý je čierny v prvej sieti, je čierny aj v druhej.

Zamyslime sa teraz, ako bude situácia v našich dvoch sieťach vyzerať o minútu. V prvej sieti sa množina čiernych štvorcov zmení na $\mathcal{A}'$, v druhej na $\mathcal{B}'$. A bude naďalej platiť vzťah $\mathcal{A}' \subseteq \mathcal{B}'$? Ľahko sa presvedčíme, že áno. Ak je nejaký štvorec s čierny v $\mathcal{A}'$, tým skôr je čierny aj v $\mathcal{B}'$ – všetky čierne štvorce z $\mathcal{A}$, ktoré „hlasovali“ za to, aby s bol čierny v $\mathcal{A}'$, máme aj v $\mathcal{B}$.

Preto vôbec nemusíme skúšať všetkých $2^{6.10}$ možných začiatočných rozložení čiernych štvorcov v Marikinej štvorcovej sieti. Stačí sa nám sústrediť na jediné z nich – to, v ktorom sú na začiatku čierne úplne všetky štvorce. Ak totiž toto rozloženie po niekoľkých krokoch „vyhynie“, vieme, že aj hocijaké iné začiatočné rozloženie by „vyhynulo“ po nanajvýš toľko isto krokoch.

A pre čierny obdĺžnik už ľahko odsimulujeme zmeny počas nasledujúcich minút. Na nasledujúcom obrázku je znázornené, ako vyzerá po 0, 3, 7 a 13 minútach od začiatku.



Iný možným riešením je priamo si všimnúť (a dokázať), že každé rozmiestnenie čiernych štvorcov sa bude postupne meniť na biele podobne ako náš obdĺžnik

– po „uhlopriečkach“, začínajúc v ľavom dolnom rohu.

Aby sme mohli toto tvrdenie presnejšie sformulovať, očíslujme si políčka vo vnútri Marikinho plánu tak ako na nasledujúcom obrázku.

	6	7	8	9	10	11	12	13	14	15	
	5	6	7	8	9	10	11	12	13	14	
	4	5	6	7	8	9	10	11	12	13	
	3	4	5	6	7	8	9	10	11	12	
	2	3	4	5	6	7	8	9	10	11	
	1	2	3	4	5	6	7	8	9	10	

A teraz už ľahko postupne zdôvodňujeme:

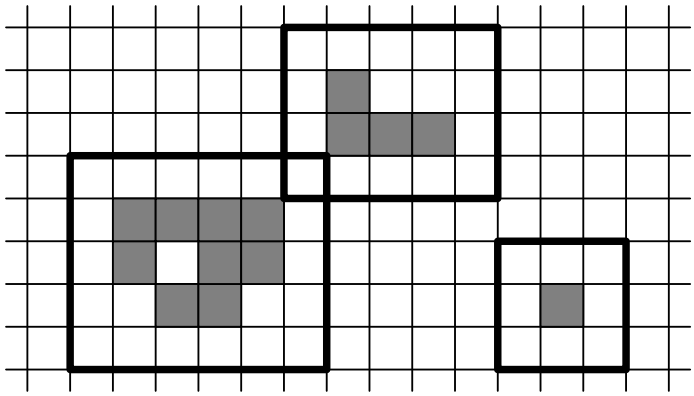
- Po prvej minúte bude štvorec s číslom 1 už navždy biely – aj pod ním, aj naľavo od neho je biely štvorec.
- Po druhej minúte budú aj štvorce s číslom 2 už navždy biele – lebo všetky štvorce, ktoré sú bezprostredne dodola a naľavo od nich sú po prvej minúte navždy biele.
- Po tretej minúte sa k navždy bielym štvorcom pridajú aj štvorce s číslom 3, a tak ďalej.

Teda bez ohľadu na to, ako Marika rozmiestni čierne štvorce, bude o 15 minút celý jej obdĺžnik biely. Vo všeobecnosti, ak máme obdĺžnik rozmerov $r \times c$ (kde $r, c \geq 3$), ktorého okraj je celý biely, tak vieme, že po $r + c - 5$ minútach už bude celé biele aj vnútro.

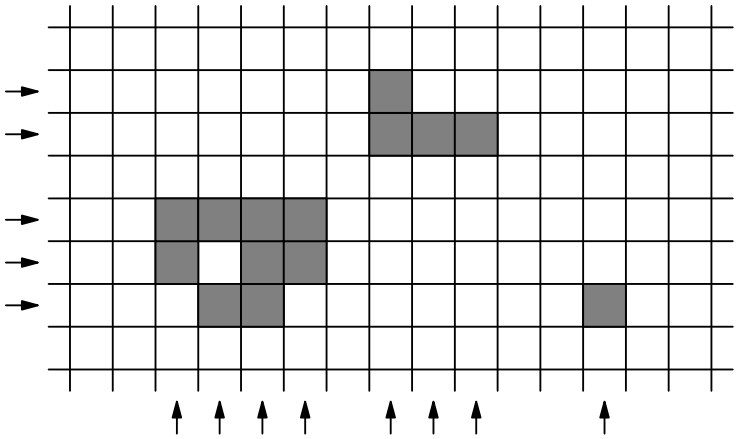
Podúloha d):

Základná myšlienka bude jednoduchá: Pozrieme sa na čierne štvorce a zvolíme nejaký obdĺžnik, ktorý ich obsahuje a má celý okraj biely. Z jeho rozmerov potom vieme, rovnako ako v predchádzajúcej podúlohe, vypočítať, po koľkých krokoch už bude zaručene celý biely.

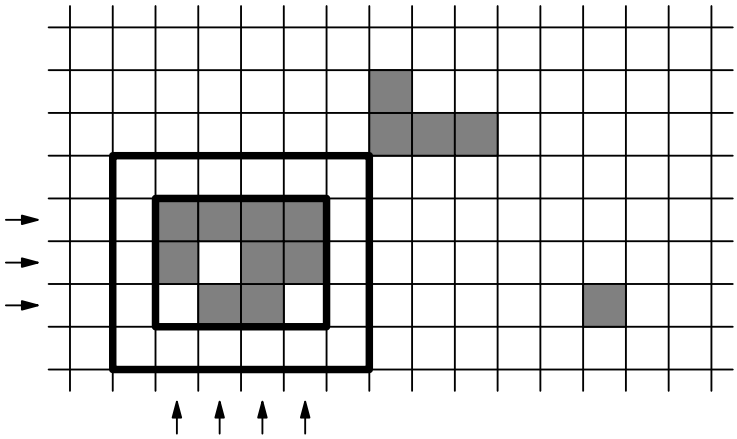
Poriadny dôkaz ale treba predsa len robiť trochu poriadnejšie. To, na čo si potrebujeme dať pozor, je situácia, kedy sú čierne štvorce príliš „roztrúsené“ – vtedy totiž musíme namiesto jedného veľkého obdĺžnika použiť viacero malých. Na nasledujúcom obrázku je príklad toho, ako sme takto „uzavreli“ 14 čiernych štvorcov do troch obdĺžnikov. Pre situáciu na obrázku vieme (podľa rozmerov najväčšieho obdĺžnika) zaručiť, že o nanajvýš 6 sekúnd už budú všetky štvorce biele.



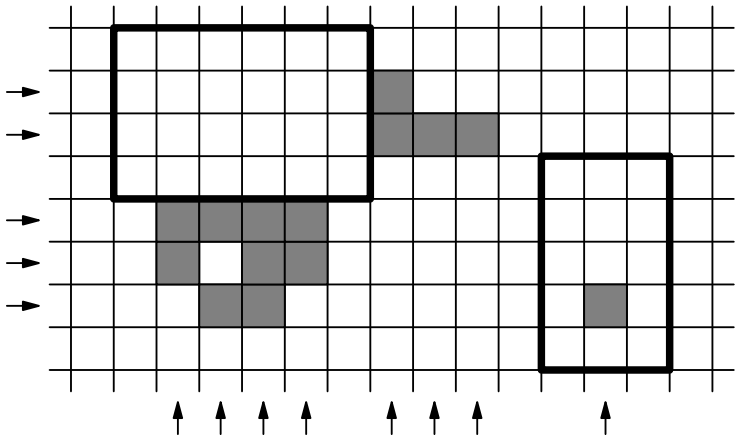
Ako takéto uzavretie do obdĺžnikov vyrobiť systematicky a zaručiť, že budú všetky z nich malé? Môžeme napríklad začať tým, že si vyznačíme riadky a stĺpce, v ktorých leží aspoň jeden čierny štvorec. (Ak teda máme 47 čiernych štvorcov, tak dokopy vyznačíme najviac 47 riadkov a najviac 47 stĺpcov.)



Vyznačené riadky aj stĺpce sa nám rozpadnú na niekoľko súvislých skupín. Napr. na obrázku nad týmto textom sú dve skupiny vyznačených riadkov a tri skupiny stĺpcov. No a keď si vyberieme jednu skupinu vyznačených riadkov a jednu skupinu vyznačených stĺpcov, dostaneme jeden z obdĺžnikov:



Tu sú ďalšie dva zo šiestich obdĺžnikov, ktoré našim postupom vznikli. Všimnite si, že môžeme dostať aj prázdne alebo zbytočne veľké obdĺžniky, to nám ale vôbec neprekáža.



A kedy že to prefarbovanie vlastne skončí? Každý obdĺžnik sa bude prefarbovať nezávisle od zvyšku roviny, preto môžeme uvažovať každý zvlášť. Najhorší možný prípad je, že budeme mať presne jeden obdĺžnik a ten bude mať rozmery 49×49. No a podľa vzťahu, ktorý sme si odvodili v podúlohe c), o 93 minút bude už aj takto veľký obdĺžnik celý biely. Jedným možným riešením podúlohy d) je teda číslo $k = 93$.

Pár slov na záver:

Viacerí riešitelia uviedli pokusy o dôkaz, založené na tvrdení, že v každom kroku sa celkový počet čiernych štvorcov zmenší. Toto ale nie je pravda – existujú dokonca konfigurácie, pre ktoré sa v niektorom kroku celkový počet čiernych štvorcov dokonca zväčší. Viete takúto konfiguráciu nájsť?

Tiež si môžeme všimnúť, že odhad, ktorý sme dokázali v riešení podúlohy d), je síce správny, ale nie je tesný. V skutočnosti sa dá dokázať, že ak máme na začiatku n čiernych štvorcov, tak už po n minútach budú úplne všetky štvorce biele. Detaily dôkazu ponechávame ako cvičenie. Poradíme len, že namiesto vhodných obdĺžnikov môžeme radšej zvoliť vhodné rovnoramenné pravouhlé trojuholníky.

Riešenia celoštátneho kola kategórie A

A-III-1 Odpoveď

Keďže číslo 42 sa v celej postupnosti vyskytuje práve raz, začneme tým, že nájdeme jeho pozíciu p. Zaujímajú nás teraz len tie súvislé podpostupnosti, ktoré túto pozíciu obsahujú.

Riešenia hrubou silou (aspoň kubická časová zložitosť):

Prvým a najjednoduchším riešením je zobrať si každý možný začiatok z a koniec k také, že $z \leq p \leq k$ a $k-z$ je párne, a pre každú dvojicu overiť, či medián tohoto úseku je naozaj 42. Priamočiara forma overovania: overovaný úsek si skopírujeme do pomocného poľa, to usporiadame a pozrieme sa na prostredný prvok. Takéto riešenie má časovú zložitosť $\Theta(n^3 \log n)$, lebo máme $\Theta(n^2)$ úsekov a pre väčšinu z nich na triedenie potrebujeme $\Theta(n \log n)$ krokov.

Existujú aj rôzne (pomerne komplikované) algoritmy, pomocou ktorých vieme medián n-prvkovej postupnosti nájsť v čase $\Theta(n)$. Použitie takéhoto algoritmu namiesto triedenia vedie k časovej zložitosti $\Theta(n^3)$.

Tú istú časovú zložitosť však vieme dosiahnuť aj oveľa ľahšie. Stačí si uvedomiť, že nepotrebujeme *nájsť medián*, len *overiť*, či je 42 mediánom. Pri overovaní nás nezaujíma rozmiestnenie prvkov. Jediné, čo potrebujeme overiť, sú ich počty: počet prvkov menších ako 42 musí byť rovný počtu prvkov väčších ako 42. A toto vieme veľmi ľahko overiť v čase lineárnom od dĺžky testovaného úseku. Takto teda dostávame ľahšie riešenie s časovou zložitosťou $\Theta(n^3)$.

Lepšie riešenie (kvadratická časová zložitosť):

Ako sme si všimli, nikdy nás nezaujímala konkrétne hodnota nejakého prvku, vždy sme sa len pýtali, či je menší alebo väčší ako 42. Upravme si teda naše vstupné pole tak, že namiesto čísla 42 dáme 0, čísla väčšie ako 42 zmeníme na $+1$ a čísla menšie na -1. Podmienku „úsek obsahuje rovnako veľa čísel väčších ako 42 a menších ako 42“ teraz vieme vyjadriť jednoducho ako „úsek má súčet 0“.

Teda platí, že konkrétna súvislá podpostupnosť má medián 42 práve vtedy, ak po úprave poľa obsahuje 0 a zároveň má súčet rovný 0. (Všimnite si, že nepotrebujeme uvažovať podmienku o nepárnej dĺžke. Tá totiž vyplýva zo zvyšných dvoch – každá postupnosť, ktorá obsahuje jednu 0 a rovnako veľa $+1$ a -1 musí mať nepárnu dĺžku.)

Kubické riešenie bolo zbytočne pomalé preto, že pre každý začiatok a koniec začínalo vždy rátať odznova. Skúsme to teda šikovnejšie. Postupne vyskúšame všetky z od 1 po p. Pre konkrétne z najskôr položíme $k = z$. V tejto chvíli je súčet úseku rovný hodnote na políčku z. Následne zväčšujeme k až po n a zakaždým v konštantnom čase prepočítame súčet zodpovedajúceho úseku. (Iba zoberieme doterajší súčet a pripočítame k nemu nasledujúci člen postupnosti.) No a vždy, keď $k \geq p$ a aktuálny súčet je rovný 0, sme našli jeden z vyhovujúcich úsekov.

Toto riešenie teda každú dvojicu $z \leq k$ spracuje v konštantnom čase, a teda má časovú zložitosť $\Theta(n^2)$.

(Poznámka: Na myšlienke „pre každý začiatok postupne zvyšujem koniec“ sa tiež dá navrhnúť riešenie s o trochu horšou časovou zložitosťou $\Theta(n^2 \log n)$. Pri ňom pracujeme priamo s pôvodnými hodnotami zo vstupu. Použijeme usporiadanú množinu (`multiset` v C++), do ktorej postupne vkladáme prvky postupnosti od z-teho ďalej. Pritom si udržiavame ukazovateľ (iterátor) na medián.)

Iné kvadratické riešenie:

Aj vyššie uvedené kvadratické riešenie ešte robí veľakrát to isté: pre každé z prechádzame v cykle pre k aspoň od p po n a znova a znova sčitujeme tie isté prvky. Pokúsime sa túto neefektívnosť odstrániť. Najskôr to síce opäť povedie ku riešeniu s kvadratickou časovou zložitosťou, toto však nižšie vylepšíme.

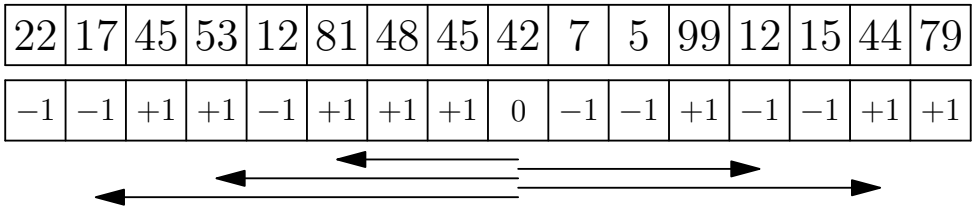
Začneme rovnakým predspracovaním ako vyššie uvedené kvadratické riešenie: zmeníme prvky na 0, +1ky a −1ky. Potom ale budeme prvky sčitovať len raz. Najskôr začneme od p a pre každé k si spočítame súčet β_k úseku od p po k. Toto celé vieme spraviť v lineárnom čase. No a následne spravíme to isté v opačnom smere: pre každé z od p po 1 spočítame súčet α_z úseku od z po p.

Pomocou takto predpočítaných hodnôt vieme spraviť ďalšie riešenie v čase $\Theta(n^2)$: vyskúšame všetky dvojice z, k pre ktoré platí $z \leq p \leq k$ a vyberieme tie z nich, kde $\alpha_z + \beta_k = 0$. (Alebo inými slovami, $\alpha_z = -\beta_k$.)

Vzorové riešenie:

Ak chceme lepšiu ako kvadratickú časovú zložitosť, potrebujeme nájsť spôsob, ako zarátať viac dobrých úsekov naraz. Tu nám pomôže, keď si uvedomíme, že súčty α_z a β_k nadobúdajú hodnoty len od $-n$ po n.

Hlavnú myšlienku si najskôr ukážeme na obrázku:



- Horný riadok ukazuje vstupné pole, v dolnom sú hodnoty po úvodnej úprave.
- Šípky smerujúce doľava ukazujú tri rôzne úseky, pre ktoré platí $\alpha_z = 3$.
- Šípky smerujúce doprava ukazujú dva rôzne úseky, pre ktoré platí $\beta_k = -3$.
- Spojením hociktorého z týchto ľavých a hociktorého z týchto pravých úsekov dostaneme úsek so súčtom 0.

Formálne, pre každé s, nech $A[s]$ je počet z takých, že $\alpha_z = s$ a $B[s]$ nech je počet k takých, že $\beta_k = s$. Zvoľme si konkrétne s (medzi $-n$ a n, vrátane). Koľko existuje postupností takých, že $\alpha_z = s$ a $\beta_k = -s$? Presne $A[s] \times B[-s]$: máme totiž $A[s]$ možností, kde takáto postupnosť môže začínať, a nezávisle od toho $B[-s]$ možností, kde končí.

Polia A a B vieme vypočítať v lineárnom čase od n (takmer rovnakým algoritmom ako sme počítali hodnoty α_z a β_k). Aj následné vyskúšanie všetkých možných s a spočítanie dobrých podpostupností prebehne v lineárnom čase. Práve sme teda popísali riešenie s časovou zložitosťou $\Theta(n)$.

Listing programu (C++)

```
#include <iostream>
#define MAXN 1234567

// drobny trik: deklarujeme A a B tak, aby do nich slo indexovat zapornymi cislami
int N, P[MAXN], Ashift[2*MAXN+1], Bshift[2*MAXN+1], *A=Ashift+MAXN, *B=Bshift+MAXN;

int main() {
    // nacitame a upravime vstup
    std::cin >> N;
    for (int n=0; n<N; ++n) std::cin >> P[n];
    for (int n=0; n<N; ++n) P[n] = P[n]>42 ? 1 : P[n]<42 ? -1 : 0;

    // najdeme nulu
    int p=0; while (P[p]!=0) ++p;

    // zistime sucty usekov zacinajucich a konciacich nulou
    int alpha=0, beta=0;
    for (int i=p; i>=0; --i) { alpha += P[i]; ++A[alpha]; }
    for (int i=p;  i<N; ++i) { beta  += P[i]; ++B[beta];  }

    // spocitame odpoved
    long long odpoved = 0;
```

```
    for (int i=-N; i<=N; ++i) odpoved += 1LL * A[i] * B[-i];
    std::cout << odpoved << "\n";
}
```

A-III-2 U obchodníka s rozličným kradnutým tovarom

V tomto vzoráku budeme predpokladať, že suma s, ktorú treba zaplatiť, je rádovo väčšia ako hodnota najväčšej mince c_n. (To úplne zodpovedá dnešným trhovým cenám modelov vesmírnych lodí a tiež ohraničeniam v zadaní úlohy.) Tento predpoklad využijeme len pri porovnávaní efektivity rôznych riešení – nebude na ňom závisieť ich korektnosť.

Stručný prehľad riešení:

Základným korektným (ale veľmi pomalým) riešením je skúšanie všetkých možností platenia. Napríklad postupne skúšať všetky možné spôsoby použitia 1, 2, 3, ... mincí, až kým nenájdeme spôsob, ako zaplatiť práve požadovanú sumu. Takéto riešenie má časovú zložitosť rádovo úmernú n^m, kde m je počet mincí, ktoré treba v optimálnom riešení.

Existujú aspoň tri rôzne prístupy, ktoré vedú k riešeniu s časovou zložitosťou lineárnou od s:

1. Pre dostatočne veľa súm vypočítame optimálne zaplatenie bez vydávania a následne vyskúšame všetky potrebné možnosti pre sumu, ktorú Roberto obchodníkovi vydá. To ťažké na tomto riešení je určiť, ktoré možnosti už netreba skúšať.

2. Iné riešenie je založené na grafovom pohľade: Vrcholy grafu sú celé čísla predstavujúce sumu, ktorú ešte treba zaplatiť. Každá hrana predstavuje použitie jednej mince. V takomto grafe hľadáme najkratšiu cestu z 0 do s. Počas tohoto hľadania pre každú sumu „po ceste“ vypočítame najkratšiu vzdialenosť do nej – teda najmenší počet mincí potrebný na jej zaplatenie.

3. Časovú zložitosť zhruba kvadratickú od s (a ešte závisiacu aj od n a c_n) vieme dosiahnuť tak, že postupne pre každý počet mincí zostrojíme množinu všetkých súm, ktoré sa dajú daným počtom mincí zaplatiť.

   Vhodnými optimalizáciami (generovaním len niektorých vhodných súm, nie všetkých) sa dá toto riešenie vylepšiť tak, aby od s závisela jeho časová zložitosť len lineárne.

Vzorové riešenie navyše pridáva pozorovania, ktoré nám umožnia sumy s väčšie od istej hranice platiť pažravo. Tým dostávame riešenie, ktorého časová zložitosť závisí len od n a c_n.

Platba bez vydávania:

Začnime riešením zjednodušenej úlohy, v ktorej musí obchodník zaplatiť presne sumu s bez toho, aby mu Roberto vydal. Na prvý pohľad by sa mohlo zdať, že pri platení je vždy optimálne použiť čo najväčšiu možnú mincu, no už vstup v zadaní nás presvedčí o opaku: Keď máme mince $(1, 5, 6)$ a chceme zaplatiť sumu 10, stačia nám na to dva kusy mincí: $5 + 5$. Ak by sme však použili mincu 6, zvyšok sumy by sme museli zaplatiť pomocou mince 1, teda dokopy by sme potrebovali päť kusov mincí.

Takéto *greedy* (pažravé) riešenie je síce nekorektné, no pomôže nám získať horný odhad správneho výsledku. Na zaplatenie sumy s vždy stačí menej ako $\lfloor s/c_n \rfloor + c_n$ mincí. Totiž jeden spôsob platenia je, že platíme mincou c_n, kým sa dá, a zvyšok (ktorý je určite menší ako c_n) zaplatíme mincou 1. Ak by sme počítali so všetkými druhmi mincí (nielen s 1 a c_n), mohli by sme dostať ešte lepší odhad, nám však postačí aj tento.

Označme najmenší počet kusov mincí potrebných na zaplatenie sumy i bez možnosti vydávania ako $P[i]$. Použijeme princíp *dynamického programovania*: Hodnoty $P[i]$ budeme počítať postupne od menších i k väčším a pri tom budeme používať už vyrátané hodnoty.

Na zaplatenie sumy 0 nám stačí 0 mincí, preto $P[0] = 0$. Keď máme zaplatiť sumu $i > 0$, môžeme začať ľubovoľnou mincou c_j, ktorej hodnota nepresahuje i. Použijeme tak jednu mincu a zostane nám zaplatiť $i - c_j$, na čo potrebujeme minimálne $P[i - c_j]$ mincí. Zo všetkých možných mincí c_j si samozrejme vyberieme tú najvýhodnejšiu. Hodnotu $P[i]$ teda vypočítame podľa vzťahu:

$$P[i] = \min \left\{ P[i - c_j] + 1 \;\middle|\; 1 \leq j \leq n \;\wedge\; c_j \leq i \right\}$$

Všimnite si, že pri výpočte $P[i]$ naozaj využívame len známe hodnoty. Aby sme nakoniec vedeli vypísať nejaký optimálny spôsob platenia, pre každú sumu i si zapamätáme aj to, ktorou mincou ju máme začať platiť.

V poli P si potrebujeme pamätať $O(s)$ hodnôt; každú z nich vypočítame v čase $O(n)$. Časová zložitosť je preto $O(sn)$ a pamäťová $O(s)$. Zdôrazňujeme, že ešte nejde o korektné riešenie úlohy zo zadania, len o pomocnú úvahu, ktorú ďalej využijeme.

Riešenie systémom „najskôr zaplať, potom vydaj“:

Na optimálny spôsob platenia sa môžeme dívať tak, že najskôr obchodník Robertovi zaplatí nejakú sumu $s + x$ a následne mu Roberto vydá sumu x. Obchodník na to samozrejme v optimálnom riešení použije $P[s + x]$ mincí a Roberto $P[x]$. Stačí teda nájsť to optimálne $x \geq 0$, pre ktoré bude súčet $P[s + x] + P[x]$ najmenší možný.

Samozrejme, zatiaľ máme pre x nekonečne veľa možností a rozhodne by sme ich nechceli skúšať všetky. Otázka preto znie: aké najväčšie môže byť x?

Tu treba byť opatrný, pretože je veľmi ľahké oklamať sám seba napríklad tvrdením „Roberto nikdy nepoužije najväčšiu mincu“ alebo tvrdením „Roberto vždy bude vydávať nanajvýš s“.

Protipríklad na obe tieto tvrdenia: mince $(1, 90, 100)$ a suma $s = 70$. Jediným optimálnym riešením je, že obchodník zaplatí $90 + 90 + 90$ a Roberto mu vydá $100 + 100$ (teda dokopy 5 kusov mincí zmení majiteľa).

Jeden možný (aj keď nie najlepší) korektný horný odhad, po aké x treba skúšať:

Každý typ mince zjavne použije len jeden z nich – nemá zmysel, aby obchodník nejakými mincami platil a Roberto mu také isté vydal.

Ak Roberto v optimálnom riešení nepoužije mincu s cenou c_n: Každej inej mince použije najviac $c_n - 1$ kusov. (Ak by totiž použil c_n kusov mince s cenou c_i, bolo by výhodnejšie použiť c_i kusov mince s cenou c_n.) Dokopy teda Roberto použije určite menej ako nc_n mincí, každá má cenu menej ako c_n, a teda $x < nc_n^2$.

Ak Roberto použije mincu s cenou c_n: Znamená to, že túto mincu nechcel použiť obchodník. Rovnakou argumentáciou ako v predchádzajúcom prípade dostávame, že suma $s + x$, ktorú platil obchodník, je nutne menšia ako nc_n^2. A to isté teda tým skôr platí pre hodnotu x.

Výsledkom je teda nasledujúce riešenie: Vypočítame hodnoty v poli P od 0 po $s + nc_n^2$. Potom vyskúšame všetky x od 0 po nc_n^2 a nájdeme to, pre ktoré je súčet $P[s + x] + P[x]$ najmenší možný. Toto riešenie má časovú zložitosť $O(sn + n^2c_n^2)$.

Riešenie pomocou prehľadávania grafu:

Pri návrhu riešenia zadanej úlohy nás môže pokúšať prístup, pri ktorom priamo počas jedného dynamického programovania budeme spracúvať aj možnosti, kedy platí obchodník, aj možnosti, kedy vydáva Roberto. Označme $Q[i]$ najmenší počet kusov mincí potrebných na zaplatenie sumy i, ak je povolené aj

vydávanie. Tou istou úvahou ako pri poli P dostávame rekurentný vzťah:

$$Q[i] = \min\left\{Q[i-c_j]+1 \;\middle|\; 1 \le j \le n \;\wedge\; c_j \le i\right\} \cup \left\{Q[i+c_j]+1 \;\middle|\; 1 \le j \le n\right\}$$

Tu je ale problém: Tým, že sme povolili aj odčítanie, nám vo vzťahoch vznikli cykly: napr. hodnota $Q[7]$ závisí na hodnote $Q[7+c_n]$ a zároveň aj $Q[7+c_n]$ závisí na $Q[7]$. Tadiaľto teda cesta nevedie.

Tu je dobré si uvedomiť, že tá situácia, do ktorej sme sa dostali pri návrhu rekurencií, je totožná s tou, ktorú stretávame pri hľadaní najkratších ciest: vzdialenosť z Bratislavy do Popradu a z Bratislavy do Štrby spolu súvisia, lebo aj z Popradu môžeme ísť ďalej na Štrbu, aj zo Štrby do Popradu.

Na celý problém sa teda môžeme dívať ako na hľadanie najkratšej cesty v grafe. Vrcholmi grafu sú celé čísla, predstavujúce aktuálne zaplatenú sumu. Každá hrana má dĺžku 1 a zodpovedá použitiu mince či už jedným alebo druhým aktérom. V takomto grafe hľadáme najkratšiu cestu z vrcholu 0 do vrcholu s. Keďže všetky hrany majú jednotkovú dĺžku, môžeme použiť prehľadávanie do šírky.

Každý navštívený vrchol spracujeme v čase lineárnom od n (počtu mincí, a teda počtu vychádzajúcich hrán). Aby sme odhadli časovú aj pamäťovú zložitosť, stačí teda zhora odhadnúť počet navštívených vrcholov. Na to si pripomeňme, že máme odhad $Q[s] < \lfloor s/c_n \rfloor + c_n$. Prehľadávanie navštívi len tie sumy, ktoré sú od 0 vo vzdialenosti nanajvýš rovnej $Q[s]$.

Všetky sumy, ktoré sa dajú zaplatiť najviac k mincami, zjavne ležia v rozsahu od $-kc_n$ po kc_n. Po dosadení nášho odhadu pre $Q[s]$ dostávame, že naše prehľadávanie navštívi $O(s + c_n^2)$ vrcholov. Taká je teda aj jeho pamäťová zložitosť; časová zložitosť je potom $O(ns + nc_n^2)$.

Riešenie pomocou zostrojenia všetkých zaplatiteľných súm:

Označme M_i množinu súm, ktoré sa dajú zaplatiť použitím presne i kusov mincí; zrejme $M_0 = \{0\}$. Ak $j \in M_i$, potom pre každú mincu c_k platí $j + c_k \in M_{i+1}$ (lebo stačí i mincami zaplatiť j a potom obchodník pridá mincu c_k) a $j - c_k \in M_{i+1}$ (Roberto vydá mincu c_k). Naopak, každá suma v M_{i+1} musela vzniknúť z nejakej sumy v M_i týmto spôsobom.

Pre sadu mincí $(1, 5)$ takto dostaneme:

$M_0 = \{0\}$,
$M_1 = \{-5, -1, 1, 5\}$,
$M_2 = \{-10, -6, -4, -2, 0, 2, 4, 6, 10\}$,
$M_3 = \{-15, -11, -9, -7, -5, -3, -1, 1, 3, 5, 7, 9, 11, 15\}$, ...

Optimálnym počtom mincí na zaplatenie s je najmenšie také i, pre ktoré $s \in M_i$. Stačí nám teda postupne generovať $M_0, M_1, M_2, \ldots$, kým nezískame s.

Každú z množín M_i si môžeme reprezentovať poľom booleovských hodnôt, v ktorom si budeme pre jednotlivé sumy pamätať, či ich množina obsahuje alebo nie.[7] Iným spôsobom je pamätať si v poli pre každú sumu j najmenšie také i, pre ktoré $j \in M_i$. Z hľadiska optimálneho platenia nás totiž zaujíma len prvý výskyt sumy j. Pole s rovnakým významom sme si už skôr označili Q.

Vďaka odhadu $Q[s] < \lfloor s/c_n \rfloor + c_n$ vieme, že počet vygenerovaných množín bude nanajvýš $k = \lfloor s/c_n \rfloor + c_n - 1$ a ako sme ukázali už v predchádzajúcom riešení, rôznych hodnôt v záverečnej množine M_k (a teda aj v každej M_i) je $O(kc_n)$. Vygenerovať M_{i+1} z M_i vieme teda v čase $O(nkc_n)$. A celkovú časovú zložitosť dostávame ako súčet časov potrebných na vygenerovanie všetkých množín M_i. Celkovo teda vieme časovú zložitosť zhora odhadnúť ako $O(nk^2c_n)$. Po dosadení nášho odhadu za k dostávame teda horný odhad časovej zložitosti $O(ns^2/c_n + nc_n^3)$. Veľmi zjednodušene môžeme povedať, že čas behu tohto algoritmu rastie kvadraticky v závislosti od sumy s, ktorú platíme.

Optimalizácia predchádzajúceho riešenia:

Jednu optimalizáciu práve popísaného riešenia sme už videli: je ňou grafové riešenie. Zatiaľ, čo riešenie s množinami M_i pre každú sumu postupne zostrojuje *všetky* počty mincí, na ktoré sa ju dá zaplatiť, grafové riešenie nájde len ten *najlepší* z nich. Teraz si ešte ukážeme jeden trochu iný spôsob, ako riešenie s množinami M_i vylepšiť. Aby sme vedeli generovať množiny M_i efektívnejšie, obmedzíme sa iba na určité poradia mincí pri platení. Zavedieme dve pravidlá:

1. Aktuálne zaplatená suma nesmie nikdy presiahnuť s.
2. Roberto smie vydávať iba vtedy, keď je aktuálna suma väčšia ako $s - c_n$. (Teda nesmie vydávať, ak sa ešte dá bez porušenia prvého pravidla platiť každou mincou.)

Napríklad pre sadu mincí $(1, 5)$ a $s = 9$ budú množiny M_i vyzerať takto: $M_0 = \{0\}$, $M_1 = \{1, 5\}$, $M_2 = \{0, 2, 4, 6\}$, $M_3 = \{1, 3, 5, 7, 9\}$. Napriek tomu, že nám niektoré sumy ubudli, suma 9 sa dá stále zaplatiť tromi mincami. Aj vo všeobecnosti platí, že sa počet mincí potrebných na zaplatenie s nezväčší. Poďme si ukázať, že vždy existuje také poradie platenia a vydávania, ktoré spĺňa obe pravidlá.

[7] Polia zvyčajne nemôžeme indexovať zápornými číslami. Toto obmedzenie ľahko obídeme napríklad tak, že sume j priradíme index $j+t$, kde t je dostatočne veľké číslo. Alebo si môžeme pomôcť trikom, viď program k 1. úlohe.

Nech v optimálnom riešení obchodník zaplatí mincami $a_1, a_2, \ldots, a_u$ a Roberto vydá mince $b_1, b_2, \ldots, b_v$. Platí teda $a_1+a_2+\cdots+a_u-b_1-b_2-\cdots-b_v = s$. Korektné poradie dostaneme jednoducho tak, že vždy, keď to nezakazuje pravidlo č. 1, zaplatíme ďalšou z mincí a_i. V opačnom prípade je aktuálna suma väčšia ako $s - c_n$ (inak by sme nemohli zaplatením ďalšou mincou porušiť pravidlo č. 1), preto vydáme ďalšou z mincí b_j. Uvedomte si, že skôr ako mince na platenie sa nám minú mince na vydávanie, a že tento postup sa zasekne, až keď použijeme všetky mince a dosiahneme sumu s.

Dodržiavaním týchto pravidiel teda nič nestratíme a navyše získame nasledujúce záruky: Pre každé optimálne riešenie existuje taká suma $r \in (s - c_n, s\rangle$ a poradie mincí, že najprv bez vydávania zaplatíme r a potom už s vydávaním doplatíme do sumy s. Počas tejto druhej fázy sa bude aktuálna suma pohybovať len v intervale $(s - 2c_n, s\rangle$.

Využime tieto záruky na urýchlenie nášho riešenia. Najprv spustíme algoritmus bez vydávania, čím si v čase $O(sn)$ nájdeme optimálny spôsob platenia bez vydávania pre všetky sumy od 0 po s. Z tých si necháme len interval $(s - c_n, s\rangle$. Získali sme tak „základ“ pre množiny M_i, platí totiž $j \in M_{P[j]}$. Vezmime najmenšiu z hodnôt $P[s - c_n + 1], \ldots, P[s-1], P[s]$ a označme ju w. Ďalej budeme generovať množiny $M_{w+1}, M_{w+2}, \ldots$ naším pôvodným algoritmom, pričom sa obmedzíme len na sumy z intervalu $(s - 2c_n, s\rangle$. Takto časom vypočítame optimálny spôsob zaplatenia sumy s.

Pretože nás teraz zaujímajú len sumy z intervalu dĺžky $2c_n$, stačí nám $O(c_n)$ pamäte. Časová zložitosť sa zlepší na $O(nc_n^2)$, teda celé riešenie bude bežať v čase $O(ns + nc_n^2)$.

Ako riešiť vstupy pre $s \approx 10^{18}$:

Intuícia nám radí, že ak je s príliš veľké, optimálne bude zaplatiť väčšinu mincami c_n. Poďme si podobné tvrdenie sformalizovať a dokázať.

Majme nejakú postupnosť mincí použitých obchodníkom na platenie, označme ich $d_1, d_2, \ldots, d_\ell$. Nech sa v tejto postupnosti ani raz nenachádza minca c_n a nech $\ell \geq c_n$.

Vezmime prefixové súčty $p_0 = 0$, $p_1 = d_1$, $p_2 = d_1 + d_2$, $\ldots$, $p_\ell = d_1 + d_2 + \cdots + d_\ell$. Pretože týchto súčtov je $\ell + 1$, čo je ostro viac ako c_n, existujú také dva súčty p_i a p_j $(i < j)$, ktoré dávajú rovnaký zvyšok po delení číslom c_n. To znamená, že ich rozdiel $p_j - p_i = d_{i+1} + \cdots + d_j$ je deliteľný c_n. Časť postupnosti od $(i+1)$-vej po j-tu mincu teda môžeme nahradiť niekoľkými mincami c_n, pričom celkový počet použitých mincí sa zmenší.

Najväčšia hodnota, akú môžeme získať použitím najviac $c_n - 1$ kusov mincí,

ak nemáme k dispozícii žiadnu mincu c_n, je $(c_n - 1) \cdot c_{n-1}$. Teda ak $s > (c_n - 1) \cdot c_{n-1}$, potom pri optimálnom platení sumy s obchodník určite použije aspoň jednu mincu c_n – v opačnom prípade by potreboval aspoň c_n kusov mincí, čo by sa dalo zlepšiť podľa predchádzajúceho odseku.

Túto úvahu môžeme opakovať dovtedy, kým $s > (c_n - 1) \cdot c_{n-1}$; zakaždým znížime s o c_n a obchodníkovi zarátame použitie ďalšej mince. Efektívnejším spôsobom je pomocou delenia zistiť počet opakovaní tohto postupu a zmeny potom vykonať naraz v konštantnom čase.

Keďže suma, po ktorú musíme počítať hodnoty P, sa zmenší na $O(c_n^2)$, celková časová zložitosť nášho riešenia klesne na $O(nc_n^2)$.

Listing programu (C++)

```
#include <cstdio>
#include <vector>
#define INF 1023456789
using namespace std;

int main() {
    long long s;
    int n;
    scanf("%lld%d", &s, &n);
    vector<int> C(2 * n);  // za n minci zo vstupu pridame ich opacne hodnoty
    for (int i = 0; i < n; ++i) {
        scanf("%d", &C[i]);
        C[i + n] = -C[i];
    }
    int cn = C[n - 1];  // najvaecsia minca
    vector<long long> R(2 * n, 0);  // pocty minci v optimalnom plateni

    // zabezpecime, ze s <= (c_n - 1) * c_{n - 1}
    int q = C[n - 2] * (cn - 1);
    if (s > q) {
        R[n - 1] = (s - q + cn - 1) / cn;
        s -= R[n - 1] * cn;
    }

    // optimalne platenie bez vydavania
    vector<int> P(s + 1, INF);
    P[0] = 0;
    vector<int> tah_P(s + 1, -1);
    for (int i = 1; i <= s; ++i)
        for (int j = 0; j < n; ++j)
            if (C[j] <= i && P[i] > P[i - C[j]] + 1) {
                P[i] = P[i - C[j]] + 1;
                tah_P[i] = j;
            }

    // Q inicializujeme hodnotami z P ...
    vector<int> Q(2 * cn, INF);
    vector<int> tah_Q(2 * cn, -1);
    int t = s - 2 * cn + 1;  // sumy v Q maju indexy posunute o t
    int w = INF;
    for (int i = max(0ll, s - cn + 1); i <= s; ++i) {
        Q[i - t] = P[i];
        w = min(w, P[i]);
```

```
    }

    // ... a dalej zlepsujeme, tentoraz aj s vydavanim
    for (int i = w; i != Q[s - t]; ++i)
        for (int j = 0; j < 2 * cn; ++j)
            if (Q[j] == i)  // z prvku mnoziny M_i vyrabame prvky M_{i+1}
                for (int k = 0; k < 2 * n; ++k)
                    if (0 <= j+C[k] && j+C[k] < 2*cn && Q[j+C[k]] > i+1) {
                        Q[j + C[k]] = i + 1;
                        tah_Q[j + C[k]] = k;
                    }

    // zrekonstruujeme optimalne platenie
    while (tah_Q[s - t] != -1) {
        ++R[tah_Q[s - t]];
        s -= C[tah_Q[s - t]];
    }
    while (tah_P[s] != -1) {
        ++R[tah_P[s]];
        s -= C[tah_P[s]];
    }

    for (int i = 0; i < 2 * n; ++i)
        printf("%lld%c", R[i], (i % n == n - 1) ? '\n' : ' ');
}
```

A-III-3 Zlomkové programy sa lúčia

Pri riešení tejto úlohy ste pravdepodobne prišli na to, že podúlohy b1 a b2 spolu súvisia. Pri hľadaní odmocniny z čísla x totiž hľadáme najmenšie y také, že $y^2 \geq x$. Riešenie podúlohy b1 sa teda pravdepodobne bude dať využiť pri riešení podúlohy b2.

Ale skôr, než sa na to pozrieme podrobnejšie, si vyriešime nesúvisiacu podúlohu a. Budeme používať terminológiu, ktorú sme zaviedli v riešeniach krajského kola: na jednotlivé prvočísla v rozklade n sa dívame ako na premenné; ich exponenty sú hodnoty, ktoré sú v tých premenných uložené. Teda napr. číslo $2^7 5^4$ prečítame ako „v premennej #2 je uložená hodnota 7 a v premennej #5 hodnota 4“.

Podúloha a):

Medián troch premenných je zároveň rovný druhej najmenšej z nich. Vieme ho zostrojiť napr. nasledovne:

1. Kým sú všetky tri premenné kladné, každú zníž o 1 a zvýš medián o 1.
2. Kým sú práve dve premenné kladné, obe zníž o 1 a zvýš medián o 1.
3. V tejto chvíli už máme nájdený medián, ale ešte treba upratať:
   Kým je posledná premenná kladná, zníž ju o 1.

Samozrejme, my vopred nevieme, ktoré dve premenné budú kladné v kroku 2 a ktorá z nich bude kladná ešte aj v kroku 3. Tu je ale ľahká pomoc: do programu dáme všetky tri možnosti. Tá z nich, ktorá nastala, sa použije a tie, ktoré nenastali, sa použiť nebudú dať.

Výsledný program teda vyzerá nasledovne:

$$\left(\frac{7}{2\cdot 3\cdot 5} \;,\; \frac{7}{2\cdot 3} \;,\; \frac{7}{2\cdot 5} \;,\; \frac{7}{3\cdot 5} \;,\; \frac{1}{2} \;,\; \frac{1}{3} \;,\; \frac{1}{5} \right)$$

Príklad výpočtu pre $n = 4320 = 2^5 3^3 5^1$:

$$2^5 3^3 5^1 \quad \overset{1}{\to} \quad 2^4 3^2 7 \quad \overset{2}{\to} \quad 2^3 3^1 7^2 \quad \overset{2}{\to} \quad 2^2 7^3 \quad \overset{5}{\to} \quad 2^1 7^3 \quad \overset{5}{\to} \quad 7^3$$

Časová zložitosť nášho programu je priamo úmerná hodnote $\max(x, y, z)$.

Iné riešenie s rovnakou časovou zložitosťou sa dá založiť na myšlienke: pre tri čísla x, y a z je ich mediánom to, ktoré nie je ani najväčšie, ani najmenšie. Preto platí $\text{median}(x, y, z) = x + y + z - \max(x, y, z) - \min(x, y, z)$. No a sčítanie, odčítanie, maximum aj minimum už vieme naprogramovať – viď riešenia krajského kola.

Podúloha b1), prvé riešenie:

Namiesto výpočtu druhej mocniny čísla x môžeme vyriešiť o trochu všeobecnejšiu úlohu: násobenie. Napíšeme teda zlomkový program, ktorý bude vedieť ľubovoľné číslo tvaru $5^y 7^z 47$ prerobiť na číslo 3^{yz}. A ak sa nám toto podarí, už sme vyhrali: stačí na jeho začiatok pridať zlomky, ktoré z čísla $2^x 47$ vyrobia číslo $5^x 7^x 47$ a máme program počítajúci druhú mocninu.

Ako teda vyriešiť násobenie? Prevedieme ho na sčítanie, ktoré už robiť vieme (viď riešenia krajského kola). Začneme s 3^0 a y-krát k tej 0 pripočítame z.

Jedno kolo pripočítavania bude vyzerať nasledovne:

1. Kým je premenná #7 kladná: zníž #7, zvýš #3, zvýš #11.
2. Kým je premenná #11 kladná: zníž #11, zvýš #7.
3. O jednu zníž #5.

Ak teda začneme s číslom $3^0 5^y 7^z$, po jednom kole pripočítavania budeme mať $3^z 5^{y-1} 7^z$, po ďalšom kole $3^{2z} 5^{y-2} 7^z$, a tak ďalej. A keď sa nám minie y, dostaneme číslo $3^{yz} 7^z$. Potom už len vyprázdnime premennú #7 a máme násobenie hotové.

Presnejšie, rovnako ako v niektorých úlohách domáceho a krajského kola, aj tu budeme potrebovať jedno prvočíslo navyše. Pomocou toho si budeme pamätať, či sme ešte vo fáze 1 (znižujeme premennú #7), alebo už vo fázach 2 a 3 (naspäť zvyšujeme premennú #7). Prvočísla 47 a 43 budú predstavovať fázu 1, prvočísla 59 a 53 fázu 2 a 3.

Program pre násobenie teda bude vyzerať nasledovne:

$$\left(\frac{3 \cdot 11 \cdot 43}{7 \cdot 47} \;,\; \frac{47}{43} \;,\; \frac{59}{47} \;,\; \frac{7 \cdot 53}{11 \cdot 59} \;,\; \frac{59}{53} \;,\; \frac{61}{5^2 \cdot 59} \;,\; \frac{47 \cdot 5}{61} \right)$$

Povšimite si posledné dva zlomky. Vždy, keď sa dostaneme na koniec cyklu, chceme znížiť premennú #5 a potom otestovať, či už je na nule. To spravíme tak, že sa ju pokúsime znížiť o 2 (predposledný zlomok) a ak sa nám to podarilo, je všetko ok, zvýšime ju o 1 a pokračujeme novou iteráciou cyklu.

Keď teraz pridáme aj inicializáciu a upratanie po skončení násobenia, dostávame kompletný program:

$$\left(\frac{5 \cdot 7}{2} \;,\; \frac{3 \cdot 11 \cdot 43}{7 \cdot 47} \;,\; \frac{47}{43} \;,\; \frac{59}{47} \;,\; \frac{7 \cdot 53}{11 \cdot 59} \;,\; \frac{59}{53} \;,\; \frac{61}{5^2 \cdot 59} \;,\; \frac{47 \cdot 5}{61} \;,\; \frac{1}{59} \;,\; \frac{1}{7} \;,\; \frac{1}{5} \right)$$

Aká je časová zložitosť tohto programu? Začneme s číslom 2^x. V $\Theta(x)$ krokoch si vyrobíme číslo, z ktorého začneme počítať násobenie. Každá iterácia násobenia zahŕňa $\Theta(x)$ krokov výpočtu v prvej, $\Theta(x)$ krokov v druhej a $\Theta(1)$ krokov v tretej fáze. Iterácií bude presne x. No a záverečné upratovanie má tiež len $\Theta(x)$ krokov. Dokopy teda dostávame časovú zložitosť $\Theta(x^2)$. A keďže potrebujeme rádovo takú veľkú hodnotu vyrobiť v premennej #3, lepšie to ani nejde.

Podúloha b1), druhé riešenie:

Tentokrát nebudeme násobiť, ale rovno postupne počítať druhé mocniny od 1^2 až po x^2.

Presnejšie, v premennej #3, v ktorej má skončiť výstup, budeme mať hodnotu i^2, zatiaľ čo v premennej #5 budeme mať hodnotu $2i - 1$. Každá iterácia výpočtu bude vyzerať nasledovne:

1. Zväčši premennú #5 o dva. (Nová hodnota: $2i + 1$.)
2. Pridaj obsah premennej #5 do premennej #3. (Nová hodnota v #3: $i^2 + 2i + 1 = (i + 1)^2$.)

Zároveň pri každej iterácii znížime o jedna hodnotu vo vstupnej premennej #2. A keď tá klesne na z x na nulu, budeme mať v premennej #5 želanú hodnotu x^2. Výsledný program:

$$\left(\frac{3\cdot 5\cdot 59}{2\cdot 47}\,,\ \frac{5\cdot 5\cdot 43}{2\cdot 59}\,,\ \frac{3\cdot 7\cdot 41}{5\cdot 43}\,,\ \frac{43}{41}\,,\ \frac{37}{43}\,,\ \frac{5\cdot 31}{7\cdot 37}\,,\ \frac{37}{31}\,,\ \frac{59}{37}\,,\ \frac{1}{59}\,,\ \frac{1}{5}\,,\ \frac{1}{47} \right)$$

Trocha komentára k nemu:

- Prvý zlomok slúži na inicializáciu, použije sa len raz – v prvom kroku výpočtu.
- Druhý zlomok je začiatok cyklu: znížime premennú #2 a o dve zvýšime premennú #5.
- Tretí a štvrtý zlomok: znižujeme #5 a zvyšujeme #3. Zároveň zvyšujeme #7, aby sa nám pôvodná hodnota premennej #5 nestratila.
- Piaty až siedmy zlomok: keď sa #5 minie, presunieme obsah #7 späť do #5.
- Ôsmy zlomok: späť na začiatok cyklu.
- Deviaty a desiaty zlomok: Upratovanie – po poslednej iterácii cyklu zmažeme čo už nechceme.
- Jedenásty zlomok sa použije len pri $x = 0$.

Tentokrát je ešte očividnejšie, že časová zložitosť tohto programu je tiež $\Theta(x^2)$.
(Zaujímavosť: Pre ľubovoľné $x \geq 1$ spraví tento program presne o šesť krokov menej ako náš prvý program.)

Podúloha b2), pomalšie riešenie:

Ako sme už uviedli na začiatku, úlohu „nájdi hornú celú časť odmocniny z x“ si môžeme preformulovať do podoby „nájdi najmenšie y také, že $y^2 \geq x$“.

Z toho vyplýva priamočiary algoritmus na riešenie podúlohy b2:

1. Nájdi riešenie podúlohy b1.
2. Postupne ho používaj pre $i = 1, 2, \ldots$, zakaždým vypočítaj i^2 a porovnaj vypočítanú hodnotu s x.

Akú bude mať tento algoritmus časovú zložitosť? Na vyskúšanie konkrétnej hodnoty i potrebujeme rádovo i^2 krokov. Dokopy teda spravíme krokov rádovo $1^2 + 2^2 + \cdots + y^2$. Z toho dostávame, že časová zložitosť je $\Theta(y^3) = \Theta(x\sqrt{x})$.

Toto riešenie by dostalo 4 body.

Podúloha b2), lepšie riešenie:

Skúsenejší riešiteľ si isto uvedomí, kde sa dá ušetriť: Nebudeme odpovede skúšať sekvenčne všetky, ale použijeme upravené binárne vyhľadávanie.

V prvej fáze budeme teda skúšať hodnoty $i = 1, 2, 4, 8, \ldots$, až kým nenájdeme prvú, pre ktorú $i^2 \geq x$. V tejto chvíli vieme, že hľadané y leží niekde od $2^j + 1$ do 2^{j+1} pre nejaké j. V druhej fáze v tomto intervale správnu hodnotu y nájdeme binárnym vyhľadávaním.

Takto otestujeme dokopy $\Theta(\log y) = \Theta(\log x)$ rôznych hodnôt. Otestovanie jednej hodnoty vieme spraviť v čase $O(x)$, čím dostávame odhad časovej zložitosti $O(x \log x)$.

(Tento odhad je tesný. Počas druhej fázy riešenia vyskúšame rádovo $\log y$ hodnôt a každá hodnota, ktorú skúšame, je väčšia ako $y/2$. Časová zložitosť je teda $\Theta(x \log x)$.)

Toto riešenie by dostalo 6 bodov. Jeho implementácia je dosť nepríjemná, obzvlášť realizácia binárneho vyhľadávania vyžaduje doriešiť viacero technických detailov.

Podúloha b2), optimálne riešenie:

Existuje však aj riešenie, ktoré je ešte efektívnejšie a navyše sa výrazne ľahšie implementuje: Upravíme naše druhé riešenie podúlohy b1. Teda budeme postupne skúšať $i = 1, 2, 3, \ldots$, lenže nebudeme vždy odznovu počítať hodnotu i^2. Namiesto toho budeme zároveň so zvyšovaním i zmenšovať x tak, aby po vyskúšaní i bola vo vstupnej premennej hodnota $x - i^2$. Akonáhle klesne vstupná premenná na nulu, vieme, že aktuálna hodnota i je hľadanou odpoveďou.

Jeden možný zlomkový program podľa tejto myšlienky:

$$\left(\frac{5 \cdot 67}{2 \cdot 61}, \frac{61}{67}, \frac{3 \cdot 7^2 \cdot 61}{2}, \frac{5 \cdot 47}{61}, \frac{11 \cdot 43}{5 \cdot 7 \cdot 47}, \frac{47}{43}, \frac{1}{7 \cdot 47}, \right.$$
$$\left. \frac{3 \cdot 5 \cdot 11^2 \cdot 31}{47}, \frac{7 \cdot 37}{11 \cdot 31}, \frac{31}{37}, \frac{47}{31}, \frac{1}{11}, \frac{1}{7} \right)$$

Priebeh výpočtu, rozdelený na fázy:

1. Začíname s číslom 2^x.
   Pre $x = 0$ výpočet rovno skončil. Inak vieme použiť tretí zlomok, dostaneme $2^{x-1} \cdot 3 \cdot 7^2 \cdot 61$.
2. Dokola používame prvé dva zlomky, čím vyrobíme $3 \cdot 5^{x-1} \cdot 7^2 \cdot 61$.

3. Keď sa minú dvojky, použijeme štvrtý zlomok. Dostávame hodnotu $3 \cdot 5^x \cdot 7^2 \cdot 47$.
4. Kedykoľvek, keď sa dostaneme na toto miesto výpočtu, budeme mať číslo tvaru $3^i \cdot 5^{x-(i-1)^2} \cdot 7^{2i} \cdot 47$. (Keď sme tu prvýkrát, je $i = 1$.)
5. Používame piaty a šiesty zlomok, kým sa to dá. Ak všetko ide dobre, zmenšíme obsah premennej #5 o obsah premennej #7, teda o $2i$. Hodnota v premennej #5 sa teda zmení z $x-(i-1)^2$ na $x-(i-1)^2-2i = x-i^2-1$. Ak sa to celé podarilo, vidíme, že pôvodné x bolo ostro väčšie ako i^2, hodnota i (stále uložená v premennej #3) teda ešte nie je hľadanou odpoveďou.
6. V takomto prípade pokračujeme ďalej. Použije sa ôsmy zlomok, čím si zvýšime premennú #3 o jedno a premennú #11 (kde máme dočasne presunutú hodnotu premennej #7) o dve. Zároveň zvýšime premennú #5 o jedno.
   V tejto chvíli máme teda aktuálnu hodnotu $3^{i+1} \cdot 5^{x-i^2} \cdot 11^{2i+2} \cdot 31$.
7. Teraz sa dokola použije deviaty a desiaty zlomok na presun hodnoty premennej #11 späť do premennej #7. Keď to skončí, použije sa jedenásty zlomok.
   Tým dostávame hodnotu $3^{i+1} \cdot 5^{x-i^2} \cdot 7^{2i+2} \cdot 47$ a výpočet opäť pokračuje od vyššie popísanej fázy 4.
8. Ak sa vo fáze 5 vyprázdni premenná #5 skôr ako premenná #7, výpočet končí a v premennej #3 máme hľadanú odpoveď. Ešte potrebujeme vyprázdniť ostatné premenné.
   Toto začneme použitím siedmeho zlomku. Následne už jediné zlomky, ktoré sa ešte dajú použiť, sú posledné dva. Ich opakovaným použitím vyčistíme ostatné premenné a ostane nám nenulová len premenná #3.

Odhad časovej zložitosti: Fázy 1-3 majú zjavne časovú zložitosť $\Theta(x)$. Potom nasleduje niekoľko iterácií. V rámci každej iterácie sa v 5. fáze vykoná rádovo rovnako veľa krokov výpočtu ako v 7. fáze; ostatné fázy vykonáme v konštantnom čase. Celková časová zložitosť iterovania je teda priamo úmerná celkovému počtu krokov výpočtu v 5. fáze. No ale tých je $\Theta(x)$, lebo v 5. fáze zmenšujeme x až kým neklesne na nulu.

Tento algoritmus má teda celkovú časovú zložitosť $\Theta(x)$.

A-III-4 Tučný Santa

Hľadanie najkratšej cesty v grafe:

Vždy, keď sa vás zadanie úlohy pýta na najmenší počet krokov, na ktorý sa dá nejaký cieľ dosiahnuť, mali by sa vaše myšlienky okamžite vydať nasledujúcim smerom: Podľa situácie v úlohe si zostrojíme vhodný graf a v tomto grafe nájdeme najkratšiu cestu. Graf vznikne zakaždým rovnako: jeho jednotlivé vrcholy zodpovedajú stavom, v ktorých sa môžeme nachádzať, a hrany vychádzajúce z vrcholu zodpovedajú ťahom, ktoré môžeme v danej situácii spraviť.

Ako to bude vyzerať v našom prípade? Stavy, teda vrcholy grafu, budú pochopiteľne jednotlivé pozície, na ktorých sa Santa môže nachádzať. Presnejšie, budeme uvažovať pozície, na ktorých sa môže nachádzať ľavý horný roh Santu – tým je jeho poloha jednoznačne určená. Každý vrchol vieme jednoznačne popísať dvomi číslami: jeho súradnicami. No a keďže $r_1, s_1 \leq 2000$, bude náš graf mať nanajvýš 4 milióny vrcholov.

Z každého vrcholu povedú hrany zodpovedajúce krokom, ktoré v tej chvíli vie Santa spraviť. Keďže máme na výber len 4 smery pohybu, budú z ľubovoľného vrcholu vychádzať najviac 4 hrany.

Na to, aby sme vedeli, aké hrany v našom grafe máme, potrebujeme zistiť, na ktorých pozíciách môže Santa stáť a na ktorých (kvôli prekážkam) stáť nemôže. Týmto sa budeme zapodievať neskôr, nateraz sa tvárme, že to už vieme zistiť, a vráťme sa späť k nášmu grafu.

Keď už sme zostrojili graf, zostáva v ňom nájsť najkratšiu cestu. Na to použijeme algoritmus nazvaný prehľadávanie do šírky (breadth-first search, BFS, niekedy tiež nazývané flood fill). Toto prehľadávanie predstavuje spôsob, akým by daný graf preskúmala voda, šíriaca sa zo začiatočného vrcholu rovnomerne do všetkých smerov: najskôr spracujeme začiatočný vrchol (vzdialenosť doň je 0), potom postupne všetkých jeho susedov (do každého z nich je vzdialenosť 1), potom susedov jeho susedov (vzdialenosť 2), a tak ďalej.

Prehľadávanie do šírky ľahko implementujeme pomocou dátovej štruktúry fronta:

```
prehľadaj(vrchol v):
    označ v ako navštívený
    vzdialenosť do v = 0
    vyrob frontu Q obsahujúcu jediný prvok v
    kým Q nie je prázdna:
        vyber zo začiatku fronty Q prvok x
```

```
pre každú hranu vedúcu z x:
    ak jej koncový vrchol w nie je navštívený:
        označ w ako navštívený
        vzdialenosť do w = 1 + vzdialenosť do v
        zapamätaj si, že do w som sa dostal z x
        vlož w na koniec fronty Q
```

Časová aj pamäťová zložitosť tohto riešenia je lineárna od veľkosti plochy obývačky, teda $O(r_1 s_1)$. Prečo to tak je? V prvom rade si uvedomme, že máme $r_1 s_1$ vrcholov a najviac $4r_1 s_1$ hrán, teda dokopy máme $O(r_1 s_1)$ vrcholov a hrán. Každý vrchol najviac raz vložíme do fronty (v okamihu, keď sa nám doň prvýkrát podarí dostať) a najviac raz ho z nej potom vyberieme. Každú hranu tiež spracujeme najviac raz: vtedy, keď spracúvame vrchol, z ktorého vedie. Celkový počet krokov, ktoré náš algoritmus spraví, je preto priamo úmerný počtu vrcholov a hrán v našom grafe.

V našej implementácii si nepotrebujeme samostatne značiť, ktoré vrcholy už sú navštívené: poznáme to jednoducho tak, že akonáhle má vrchol vzdialenosť inú ako nekonečno, už sme ho niekedy navštívili.

Rekonštrukcia cesty:

Vtejto úlohe pribudla ešte jedna zákernosť a tou je vypisovanie samotnej cesty. Táto zákernosť sa dá vyriešiť napríklad tým, že si pre každé políčko zapamätáme, odkiaľ sme sa naň počas prehľadávania prvýkrát dostali. Keďže vieme, kde Santa skončil, môžeme následne pomocou týchto údajov od konca zrekonštruovať jeho cestu.

Predpočítanie:

Skôr než začne BFS, potrebujeme zistiť, ktoré pozície Santu sú navštíviteľné. Zo zadania vieme, že sú to také pozície, na ktoré keď sa Santa postaví, prekryje najviac M prekážok.

Do tabuľky T si chceme na políčko $T[y, x]$ zapísať, koľko prekážok je v odĺžniku $[y, x]$ až $[y + r_2 - 1, x + s_2 + 1]$. Ak $T[y, x] \leq M$ Santa môže stáť na pozícii y, x.

Jednoduché riešenie je, pre každé políčko prezrieť $r_2 \times s_2$ políčok vľavo dole od neho a spočítať si prekážky. Takéto riešenie dokopy spraví $\Theta(r_1 s_1 r_2 s_2)$ krokov, čo je pre veľké vstupy priveľa.

Pri vymýšľaní rýchlejšieho riešenia sa skúsme najskôr pozrieť na jednorozmerný prípad. (Samo zadanie nám ponúka radu – 6 bodov môžeme získať, ak

predpokladáme, že Santa má rozmery $1 \times s_2$.) V jednorozmernom prípade nám stačí spočítať, koľko prekážok je na políčkach $[y, x]$ až $[y, x + s_2 - 1]$. To si vypočítame postupne pre každé x:

```
T[y,0] = počet prekážok od [y,0] po [y,s2-1]
T[y,1] = T[y,0] + (1 ak [y,s2] je 'X') - (1 ak [y,0] je 'X')
...
T[y,i] = T[y,i-1] + (1 ak [y,i+s2-1] je 'X') - (1 ak [y,i-1] je 'X')
..
T[y,s1-s2] = T[y,s1-s2-1] + (1 ak [y,s1-1] je 'X') - (1 ak [y,s1-s2-1] je 'X')
```

Výborná vec na celom tomto je rýchlosť. Výpočet $T[y, 0]$ nám síce trvá s_2 krokov, ale potom už každé zo zvyšných $s_1 - s_2$ políčok vypočítame v konštantnom čase, celkovo teda spravíme pri spracovaní jedného riadku len $O(s_1)$ krokov.

No a ďalej to už je veľmi jednoduché: Stačí si uvedomiť, že dvojrozmerný prípad vyriešime tak, že spravíme ešte raz to isté, ale tentokrát po stĺpcoch (pričom použijeme hodnoty, ktoré sme práve vypočítali prechodom po riadkoch). Toto predpočítanie dokážeme spraviť v čase $O(r_1 s_1)$. To isté platí aj pre BFS. Takže dostávame riešenie so zjavne optimálnou časovou zložitosťou $O(r_1 s_1)$. Pamäťová zložitosť je tiež $O(r_1 s_1)$.

Listing programu (C++)

```
#include<cstdio>
#include<algorithm>
#include<vector>
#include<queue>
using namespace std;
#define INF 1023456789

int R,S,r,s,M;
char v;
int vstup[3][2047][2047];
int dist[2047][2047];
int prev[2047][2047];

int dy[] = {1,-1,0,0};
int dx[] = {0,0,1,-1};
char dc[] = "JSVZ";

int main(){
    scanf("%d %d %d %d %d", &R, &S, &r, &s, &M);
    //nacitanie vstupu a vyroba tabulky
    for(int i = 0; i<R; ++i)
        for(int j = 0; j<S; ++j){
            scanf(" %c", &v);
            vstup[0][i][j] = (v=='X')?1:0;
            dist[i][j] = INF;
        }
    int sum;
    for(int i = R-1; i>=0; --i){
        for(int j = S-1; j>=0; --j){
```

```
            vstup[1][i][j] = vstup[1][i][j+1]
                           + vstup[0][i][j] - ((j+s<S)?vstup[0][i][j+s]:0);
            vstup[2][i][j] = vstup[2][i+1][j]
                           + vstup[1][i][j] - ((i+r<R)?vstup[1][i+r][j]:0);
        }
    }
    // BFS
    queue<int> F;
    dist[0][0] = 0;
    F.push(0);
    F.push(0);
    int x,y, xx, yy;
    while(!F.empty()){
        y = F.front(); F.pop();
        x = F.front(); F.pop();
        if (y==R-r && x==S-s) {
            // zrekonstruovanie cesty
            vector<char> vc;
            while( y!=0 || x!=0 ){
                vc.push_back(dc[prev[y][x]]);
                yy = y-dy[prev[y][x]];
                x = x-dx[prev[y][x]];
                y = yy;
            }
            for(int i = vc.size()-2; i>=0; --i)
                printf("%c",vc[i]);
            printf("\n");
            return 0;
        }
        for(int d = 0; d<4; d++){
            yy = y+dy[d];
            xx = x+dx[d];
            // som mimo obyvacky, je tam vela prekazok alebo som tam uz bol
            if (xx<0 || yy<0 || xx>S-s || yy>R-r ||
                vstup[2][yy][xx]>M || dist[yy][xx]<=dist[y][x]+1) continue;

            dist[yy][xx] = dist[y][x]+1;
            prev[yy][xx] = d;
            F.push(yy);
            F.push(xx);
        }
    }
    printf("Santa je chudak.\n");

}
```

Alternatívne riešenie:

Namiesto predpočítania vyššie uvedeným spôsobom sa dajú priamo použiť dvojrozmerné prefixové súčty: pre každé y, x predpočítame počet prekážok $P[y, x]$ v obdĺžniku, ktorý má v protiľahlých rohoch políčka $[0, 0]$ a $[y-1, x-1]$. Toto vieme vhodným trikom spraviť v $O(r_1 s_1)$. No a z týchto hodnôt už vieme v konštantnom čase o ľubovoľnom obdĺžniku povedať, koľko prekážok obsahuje.

```
predpočítanie:
    P[y,x] = P[y-1,x] + P[y,x-1] - P[y-1,x-1] + (1 ak [y-1,x-1] je 'X')
počet prekážok v obdĺžniku od [y1,x1] po [y2-1,x2-1]:
    počet = P[y2,x2] - P[y2,x1] - P[y1,x2] + P[y1,x1]
```

A-III-5 Ministerstvo

Cieľom je samozrejme nájsť riešenie, ktoré bude efektívnejšie ako priamočiara simulácia každého príkazu. Štruktúra ministerstva, popísaná na vstupe, predstavuje strom. A keďže na všeobecnom strome sa predpísané operácie robia ťažko, prvým krokom riešenia bude prevedienie úlohy na trochu inú – na operácie nad intervalmi a prvkami v poli. Každému vrcholu stromu teda priradíme jeden prvok poľa a spravíme to tak, aby každému oddeleniu zodpovedal súvislý úsek poľa. Každý príkaz sa potom bude týkať buď jedného prvku poľa, alebo nejakého intervalu.

Vhodné priradenie pozícií v poli vrcholom strimu vieme nájsť ľahko: stačí napr. ľubovoľným spôsobom prehľadať strom do hĺbky a každému vrcholu priradiť index rovný poradiu, v ktorom je objavený. (Vyhovujúcich očíslovaní je veľa, toto je len jedno z nich, ktoré je ľahko a systematicky zostrojiteľné.)

Počas prehľadávania si zároveň vieme pre každý vrchol zapamätať, kde začína a končí interval, ktorý obsahuje jeho podriadených. (Pri prehľadávaní do hĺbky interval zodpovedajúci vrcholu v začína ním samotným. Koniec intervalu zistíme vtedy, keď prehľadávanie opúšťa vrchol v.)

Túto časť riešenia vieme teda celú ľahko spraviť v čase $O(n)$.

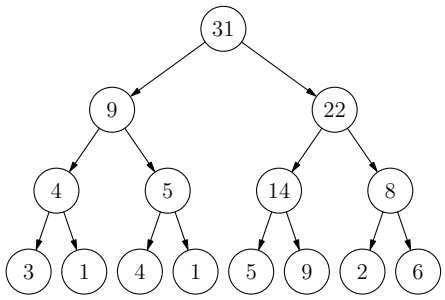
V tejto chvíli teda už môžeme na strom zabudnúť, máme už len obyčajné pole. Aby sme vedeli vykonávať príkazy zo zadania, potrebujeme s ním efektívne robiť nasledovné operácie:

- Zvýš hodnotu prvku a o k
- Zisti súčet v intervale $[a, b]$.
- Zvýš hodnotu všetkých prvkov v intervale $[a, b]$ o k.
- Zisti minimum v intervale $[a, b]$.
- Zmeň hodnotu všetkých prvkov v intervale $[a, b]$ na k.

Intervalové stromy:

Začnime efektívnou implementáciou prvých dvoch operácii. (Za to vieme získať 9 bodov. Stačí navyše doplniť, že vždy, keď budeme potrebovať zvýšiť plat celému oddeleniu, spravíme to hrubou silou – prejdeme všetkých jeho zamestnancov a každému zvýšime plat.)

Nad poľom si postavíme binárny strom. Hodnota každého vrchola bude súčet hodnôt jeho synov, a teda zároveň súčet v celom intervale poľa, ktorý je pokrytý týmto vrcholom.



Zmenu hodnoty implementujeme jednoducho: zmeníme hodnotu v poli a potom postupne ideme dohora po strome a upravujeme hodnotu všetkých predkov daného vrchola. Časová zložitosť je úmerná hĺbke tohoto stromu, čiže $O(\lg n)$. Zistenie súčtu v úseku bude trochu komplikovanejšie, ale rovnako rýchle. Potrebujeme sčítať vrcholy stromu, ktoré inteval prekrýva celé a neprekrýva ich otcov. Takýchto vrcholov bude maximálne $O(\lg n)$ – uvedomte si, že z každej úrovne sčítame maximálne dva vrcholy. Nájdeme ich jednoduchou rekurzívnou procedúrou, ktorá prezerá strom zhora nadol:

```
zisti_sucet(vrchol x, sčitovaný interval I):
  Ak je interval I disjuktný s intervalom, ktorý pokrýva podstrom vo vrchole x:
    vráť 0
  Ak je interval I nadmnožinou intervalu, ktorý pokrýva podstrom vo vrchole x:
    vráť súčet vo vrchole x
  vráť zisti_sucet(ľavý syn x, I) + zisti_sucet(pravý syn x, I)
```

Tento prechod má tiež časovú zložitosť $O(\lg n)$. Analogicky, pomocou rekurzie, vieme implementovať aj prvú operáciu: začneme v koreni stromu, zídeme dole na správne políčko poľa, to upravíme a pri vynáraní sa z rekurzie prepočítavame hodnoty v jeho predkoch. Pri tomto riešení to ešte je jedno, ale práve takáto rekurzívna implementácia bude potrebná pri modifikáciách popísaných v nasledujúcej časti riešenia.

Lenivé vykonávanie operácii:

Teraz si ukážeme ako efektívne vykonať zvyšné operácie. Začnime treťou operáciou. Na tej si ukážeme správny princíp. Zvyšné dve operácie sa potom dajú doriešiť analogicky.

Hlavnou myšlienkou bude „nerob nič predčasne“. Predstavme si, že chceme zvýšiť o k všetky hodnoty v intervale, ktorý zodpovedá nejakému vrcholu v nášho binárneho stromu. Namiesto toho, aby sme prechádzali celý podstrom a zvyšovali každý jeden prvok v ňom, jednoducho si vo vrchole v upravíme jeho súčet (to vieme spraviť hneď) a poznačíme si: „všetky prvky pod týmto vrcholom treba o k zvýšiť“. Jeho potomkov zatiaľ necháme tak, budú mať dočasne nesprávne hodnoty. To nám ale nevadí – totiž ak niekedy v budúcnosti budeme chcieť pristupovať k potomkom vrcholu v, musíme pri tom prejsť cez v. No a

práve až v tomto okamihu príde k vykonaniu zapamätaného príkazu. Presnejšie, vždy, keď chceme z nejakého vrcholu u ísť v strome hlbšie, najskôr sa pozrieme, či v u nemáme „odloženú“ nejakú operáciu. Ak áno, najskôr dotyčnú operáciu presunieme z u do oboch jeho synov, a až potom sa pohneme o krok dodola (a tam celý postup opakujeme, ak treba).

Vo všeobecnosti bude zvyšovanie intervalu o x vyzerať podobne ako zisťovanie súčtu. Len do všetkých troch operácií (zvyšovanie prvku, zvyšovanie intervalu, aj zisťovanie súčtu intervalu) teda musíme doplniť časť, pri ktorej skontrolujeme a prípadne vyriešime odložené operácie. Taktiež potrebujeme vedieť „skladať“ zvýšenia: keď už máme v nejakom vrchole zapamätané zvýšenie o a a príde zvýšenie o b, jednoducho zmeníme pamätanú hodnotu na $a + b$.

Ako vykonať posledné dve operácie – zisťovanie minima a úpravu intervalu na konkrétnu hodnotu? Zisťovanie minima bude jednoduché: v každom vrchole si budeme udržiavať aj minimum príslušného intervalu. Úprava intervalu na konkrétnu hodnotu prebehne podobne ako zvýšenie celého intervalu o k. Vykonáme ju lenivo len vo vrcholoch, ktoré sú celé pokryté intervalom, ktorý meníme. Keď robíme túto operáciu a náhodou sa vo vrchole predtým vyskytla operácia pridania, tak túto staršiu operáciu zahodíme (lebo nová operácia všetky hodnoty aj tak prepíše). A naopak, ak sme mali odloženú zmenu na hodnotu a a príde nám zvýšenie o b, môžeme si napr. zmeniť zapamätanú operáciu na „zmeň hodnotu na $a + b$“.

Vďaka lenivému spôsobu vykonávania dokážeme každú operáciu vykonať v čase $O(\lg n)$.

Listing programu (C++)

```
#include <cstdio>
#include <vector>
#include <algorithm>

using namespace std;

#define INF 2000000000000000000ll

int n;
vector<vector<int> > sons;
vector<int> salary;
vector<long long> sizes;

// Interval <a,b>
vector<pair<int, int> > postorder;

int postord(int x, int num) {
  int last = num;
  int size = 1;
  for (int i = 0; i < sons[x].size(); i++) {
    last = postord(sons[x][i], last);
```

```
    size += sizes[sons[x][i]];
  }
  sizes[x] = size;
  postorder[x] = make_pair(num, last);
  return last+1;
}

int is = 131072;

typedef long long ll;

struct Inter {
  ll sum;
  ll min;

  ll pending_change;
  ll pending_add;

  Inter() : sum(0), min(0), pending_change(-1), pending_add(0) {}
};

Inter strom[1234567];

pair<ll, ll> add(ll a, int my_begin, int my_end, int a_begin, int a_end, int v);
pair<ll, ll> change(ll ch, int my_begin, int my_end, int a_begin, int a_end, int v);

void push_down(int my_begin, int my_end, int v) {
  if (strom[v].pending_change != -1) {
    change(strom[v].pending_change, my_begin, (my_begin+my_end)/2,
           my_begin, my_end, 2*v);
    change(strom[v].pending_change, (my_begin+my_end)/2, my_end,
           my_begin, my_end, 2*v+1);
    strom[v].pending_change = -1;
  }
  if (strom[v].pending_add != 0) {
    add(strom[v].pending_add, my_begin, (my_begin+my_end)/2,
           my_begin, my_end, 2*v);
    add(strom[v].pending_add, (my_begin+my_end)/2, my_end,
           my_begin, my_end, 2*v+1);
    strom[v].pending_add = 0;
  }
}

// Vracia <sucet, minimum>
pair<ll, ll> add(ll a, int my_begin, int my_end, int a_begin, int a_end, int v) {
  if (my_begin >= a_end || my_end <= a_begin)
    return make_pair(strom[v].sum, strom[v].min);

  if (my_begin >= a_begin && my_end <= a_end) {
    strom[v].pending_add += a;
    strom[v].min += a;
    strom[v].sum += (my_end - my_begin) * a;
    return make_pair(strom[v].sum, strom[v].min);
  }

  push_down(my_begin, my_end, v);

  pair<ll, ll> r1 = add(a, my_begin, (my_end+my_begin)/2, a_begin, a_end, 2*v);
  pair<ll, ll> r2 = add(a, (my_end+my_begin)/2, my_end, a_begin, a_end, 2*v+1);
  strom[v].sum = r1.first+r2.first;
  strom[v].min = min(r1.second, r2.second);
  return make_pair(strom[v].sum, strom[v].min);
}
```

```
void add(ll a, int a_begin, int a_end) {
  add(a, 1, is, a_begin, a_end, 1);
}

pair<ll, ll> change(ll ch, int my_begin, int my_end, int a_begin, int a_end, int v) {
  if (my_begin >= a_end || my_end <= a_begin)
    return make_pair(strom[v].sum, strom[v].min);

  if (my_begin >= a_begin && my_end <= a_end) {
    strom[v].pending_add = 0;
    strom[v].pending_change = ch;
    strom[v].min = ch;
    strom[v].sum = ch*(my_end - my_begin);
    return make_pair(strom[v].sum, strom[v].min);
  }

  push_down(my_begin, my_end, v);

  pair<ll, ll> r1 = change(ch, my_begin, (my_end+my_begin)/2, a_begin, a_end, 2*v);
  pair<ll, ll> r2 = change(ch, (my_end+my_begin)/2, my_end, a_begin, a_end, 2*v+1);
  strom[v].sum = r1.first+r2.first;
  strom[v].min = min(r1.second, r2.second);
  return make_pair(strom[v].sum, strom[v].min);
}

void change(ll ch, int a_begin, int a_end) {
  change(ch, 1, is, a_begin, a_end, 1);
}

pair<ll, ll> get(int my_begin, int my_end, int a_begin, int a_end, int v) {
  if (my_begin >= a_end || my_end <= a_begin)
    return make_pair(0, INF);
  if (my_begin >= a_begin && my_end <= a_end) {
    return make_pair(strom[v].sum, strom[v].min);
  }

  push_down(my_begin, my_end, v);

  pair<ll, ll> r1 = get(my_begin, (my_end+my_begin)/2, a_begin, a_end, 2*v);
  pair<ll, ll> r2 = get((my_end+my_begin)/2, my_end, a_begin, a_end, 2*v+1);
  return make_pair(r1.first + r2.first, min(r1.second, r2.second));
}

ll get_min(int a_begin, int a_end) {
  return get(1, is, a_begin, a_end, 1).second;
}

ll get_sum(int a_begin, int a_end) {
  return get(1, is, a_begin, a_end, 1).first;
}

int main() {
  scanf("%d", &n);
  salary.resize(n);
  sons.resize(n);
  sizes.resize(n);
  postorder.resize(n);
  for (int i = 0; i < n; i++) {
    int p;
    scanf("%d %d", &salary[i], &p);
    sons[i].resize(p);
    for (int j = 0; j < p; j++) {
      scanf("%d", &sons[i][j]);
      sons[i][j]--;
```

```
    }
  }
  postord(0, 1);
  for (int i = 0; i < n; i++)
    change(salary[i], postorder[i].second, postorder[i].second+1);

  int q; scanf("%d", &q);

  for (int o = 0; o < q; o++) {
    int a, u, x, y;
    scanf("%d %d", &a, &u);
    if (a != 3) scanf("%d %d", &x, &y);
    u--;
    if (a == 1) {
      if (get_sum(postorder[u].second, postorder[u].second+1) < x) {
        add(y, postorder[u].second, postorder[u].second+1);
      }
    } else if (a == 2) {
      if (get_sum(postorder[u].first, postorder[u].second+1) < x * sizes[u]) {
        add(y, postorder[u].first, postorder[u].second+1);
      }
    } else {
      ll mi = get_min(postorder[u].first, postorder[u].second+1);
      change(mi, postorder[u].first, postorder[u].second+1);
    }
  }
  for (int i = 0; i < n; i++) {
    printf("%Ld\n", get_min(postorder[i].second, postorder[i].second+1));
  }
}
```

Výsledky krajských kôl kategórie B

V kategórii B súťaž končí krajským kolom. Na nasledujúcich stranách uvádzame výsledky tohto kola v jednotlivých krajoch. Výsledky neboli koordinované, je teda možné, že v rôznych krajoch boli použité mierne odlišné bodovacie škály. Úspešnými riešiteľmi tohto krajského kola sú tí riešitelia, ktorí získali aspoň 10 bodov.

Banskobystrický kraj:

Meno	Ročník, škola	1.	2.	3.	4.	Σ
1. Juraj Pančík	2 Gym. Tajovského B. Bystrica	4	4	9	9	26
2. Norbert Slivka	2 Gym. Tajovského B. Bystrica	4	4	5	7	20
3. Martin Žilák	2 Gym. Tajovského B. Bystrica	1	–	4	8	13

Bratislavský kraj:

Meno	Ročník, škola	1.	2.	3.	4.	Σ
1. Barbora Kováčová	2 Škola pre mim. nadané deti	7	9	7	8	31
2. Martin Kotuliak	2 Gym. Grösslingová	7	4	5	6	22
3. Zhen Ning Dávid Liu	2 Gym. Grösslingová	2	4	8	7	21
4. Bui Truc Lam	1 Gym. Grösslingová	0	4	10	4	18
5. Jakub Havelka	2 Gym. Jura Hronca	8	2	1	6	17
6. Ján Ondráš	2 Gym. Grösslingová	3	4	3	6	16
7. Marián Longa	1 Škola pre mim. nadané deti	3	3	1	6	13
8. Alena Poláchová	2 Gym. Grösslingová	0	4	2	6	12
9. Martin Rusinko	2 Gym. Jura Hronca	2	3	2	4	11
Šimon Seman	2 Gym. Jura Hronca	1	4	–	6	11
11. Jakub Veselý	1 Gym. Jura Hronca	2	2	2	4	10

Košický kraj:

Meno	Ročník, škola	1.	2.	3.	4.	Σ
1. Matúš Porázik	2 Gym. Alejová Košice	5	4	5	6	20
Vladislav Vancák	2 Gym. Alejová Košice	0	4	8	8	20
3. Jana Bátoryová	2 Gym. Alejová Košice	4	4	5	6	19
Marek Dobranský	2 Gym. P. Horova Michalovce	4	4	5	6	19
5. Miroslav Stankovič	2 Gym. Poštová Košice	0	3	5	9	17
6. Jakub Kupčík	2 Gym. Šrobárova Košice	5	0	6	5	16
7. Jozef Karlik	2 Gym. P. Horova Michalovce	4	3	4	4	15
8. Ladislav Rauch	2 Gym. P. Horova Michalovce	3	2	4	5	14

Nitriansky kraj:

Meno	Ročník, škola	1. 2. 3. 4.	Σ
1. Michal Bali	1 Gym. Párovská Nitra	5 – 5 6	16
2. Michal Porubský	0 SKŠ Nitra, Gym. sv. Cyrila a Metoda	3 3 4 5	15
3. Balázs Nagy	2 SPŠ Komárno	1 3 2 2	8

Prešovský kraj:

Meno	Ročník, škola	1. 2. 3. 4.	Σ
1. Róbert Eckhaus	2 Gym. Konštantínova Prešov	4 2 8 7	21
2. Jozef Lukáč	2 Gym. L. Stöckela Bardejov	4 1 7 6	18
3. Peter Gábor	1 Gym. Konštantínova Prešov	5 3 4 5	17
4. Lukáš Richtarik	1 Gym. sv. Moniky Prešov	4 3 – 5	12
5. Michal Bujňák	2 Gym. Konštantínova Prešov	2 – 1 5	8
6. Matej Veselovský	1 Gym. Konštantínova Prešov	1 0 0 6	7
7. Kamil Janeček	1 Gym. Kežmarok	1 3 2 –	6

Trenčiansky kraj:

Meno	Ročník, škola	1. 2. 3. 4.	Σ
1. Filip Pokrývka	2 Gym. Bánovce nad Bebravou	2 4 6 10	22
Pavel Madaj	0 Gym. Nedožerského Prievidza	– 8 5 9	22
3. Michal Korbela	2 Gym. Bánovce nad Bebravou	9 0 5 6	20
4. Július Flimmel	2 Gym. Púchov	4 1 6 6	17
5. Filip Ayazi	1 Gym. Ľudovíta Štúra Trenčín	– 3 6 7	16
6. Andrej Zbín	2 Gym. Púchov	4 4 0 7	15
7. Jakub Arbet	-3 ZŠ Kubranská Trenčín	4 – 2 5	11
8. Dušan Javorek	2 Gym. Púchov	4 2 0 4	10
9. Andrej Tkadlec	0 Gym. Nedožerského Prievidza	– 0 2 5	7

V Trnavskom kraji sa krajské kolo kategórie B neuskutočnilo, keďže doň z domáceho kola nik nepostúpil.

Žilinský kraj:

Meno	Ročník, škola	1. 2. 3. 4.	Σ
1. Filip Matušák	2 Gym. Námestovo	4 4 5 9	22
2. Pavol Olexa	2 Gym. Námestovo	5 4 1 8	18

Výsledky celoštátneho kola kategórie A

Celoštátne kolo kategórie A sa v 27. ročníku Olympiády v informatike uskutočnilo v dňoch 28. až 31. marca v Rakoviciach, okres Piešťany, na tamojšej strednej odbornej škole. Najlepších štrnásť riešiteľov bolo vyhlásených za úspešných a z nich najlepších osem za víťazov celoštátneho kola. Všetci úspešní riešitelia boli pozvaní na výberové sústredenie.

	Meno	Ročník, škola	1.	2.	3.	4.	5.	Σ
1.	Andrej Mariš	4. Gym. Piaristická Nitra	10	9	5	15	12	51
2.	Jakub Šafin	3. Gym. P. Horova Michalovce	10	8	8	15	9	50
3.	Jozef Brandys	4. Gym. Námestovo	10	9	5	15	5	44
4.	Eduard Batmendijn	1. Cirk. gym. Stará Ľubovňa	10	10	5	14	3	42
5.	Marián Horňák	4. Gym. Párovská Nitra	10	5	10	15	0	40
6.	Mário Lipovský	2. Gym. Jura Hronca	10	8	5	15	0	38
7.	Jaroslav Petrucha	3. Gym. Metodova	10	9	10	5	3	37
	Viktória Vozárová	2. Gym. Jura Hronca	10	4	5	15	3	37
9.	Kamila Součková	3. Škola pre mim. nadané deti	6	6	3	15	4	34
10.	Askar Gafurov	3. Gym. Grösslingová	10	7	6	6	3	32
11.	Vladimír Macko	3. Gym. Ľ. Štúra Zvolen	10	9	4	5	3	31
12.	Jerguš Greššák	3. Škola pre mim. nadané deti	10	7	6	—	6	29
	Roman Hudec	4. Gym. Nové Zámky	6	8	6	5	4	29
14.	Jozef Marko	3. Gym. Lettricha Martin	10	3	4	5	5	27
15.	Martin Kalužník	3. GLN Tomášikova	5	2	3	13	—	23
	Rastislav Rabatin	3. Gym. Jura Hronca	3	3	2	15	—	23
17.	Filip Rydzi	4. GLN Tomášikova BA	10	4	3	2	3	22
18.	Barbora Klembarová	4. Gym. Kukučínova Poprad	10	8	3	0	0	21
19.	Richard Kakaš	3. Gym. Jura Hronca	10	1	3	2	3	19
	Richard Molnár	3. Ev. gym. Lipt. Mikuláš	10	0	6	0	3	19
21.	Michal Bock	3. Gym. Grösslingová	10	1	5	0	1	17
	Gabriela Sklenčárová	3. Gym. Jura Hronca	7	3	5	—	2	17
23.	Mária Vajdová	4. Gym. de Coubertina Piešťany	6	4	4	—	1	15
24.	Michal Fikar	3. Gym. Jura Hronca	6	0	5	0	3	14
	Miroslav Stankovič	2. Gym. Poštová Košice	4	6	4	0	—	14
26.	Márton Bartal	3. Súkr. gym. s vjm. Dun. Streda	1	0	3	2	—	6
27.	Milan Mikuš	4. Gym. Ul. 1. mája Trenčín	1	0	3	—	0	4

Výsledky výberového sústredenia

V dňoch 25. apríla až 1. mája 2012 sa v Bratislave konalo výberové sústredenie. Na toto sústredenie boli pozvaní všetci štrnásti úspešní 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, za „domáce úlohy“ a za sedem súťažných dní výberového sústredenia.

	Meno	$\sum$	CK	d.ú.	st	št	pi	so	ne	po	ut
1.	Jakub Šafin	715.96	50	57.10	45.5	105.0	150.9	49.0	130.5	79.96	48.0
2.	Marián Horňák	708.84	40	33.00	35.0	94.0	122.5	70.0	149.5	127.34	37.5
3.	Eduard Batmendijn	672.21	42	24.75	68.0	89.5	138.0	24.0	114.0	128.46	43.5
4.	Andrej Mariš	600.80	51	53.25	54.5	79.0	97.0	31.0	130.0	75.05	30.0
5.	Jozef Brandys	585.62	44	46.50	26.5	71.0	132.0	42.5	115.0	90.12	18.0
6.	Askar Gafurov	420.61	32	35.25	30.5	30.5	72.5	21.0	121.0	48.36	29.5
7.	Vladimír Macko	396.95	31	15.00	18.5	50.5	101.5	26.0	55.0	67.95	31.5
8.	Jerguš Greššák	395.23	29	13.00	34.0	35.0	99.0	17.5	79.5	69.23	19.0
9.	Jozef Marko	392.76	27	39.50	24.5	26.5	102.0	45.0	49.5	64.26	14.5
10.	Mário Lipovský	342.91	38	14.50	34.5	47.5	50.5	13.0	94.5	41.91	8.5
11.	Roman Hudec	310.25	29	12.00	6.0	27.0	99.0	18.0	62.5	47.25	9.5
12.	Kamila Součková	308.30	34	35.80	4.0	27.0	88.0	12.0	83.5	12.50	11.5
13.	Jaroslav Petrucha	302.27	37	6.25	32.0	43.0	57.0	7.5	55.5	52.02	12.0
14.	Viktória Vozárová	258.61	37	16.50	7.0	37.0	72.0	21.0	29.5	27.11	11.5

Medzinárodné prípravné sústredenie v Davose

Vďaka blízkej spolupráci medzi národnými olympiádami v informatike na Slovensku a vo Švajčiarsku mali aj tento rok vybraní riešitelia KSP a OI možnosť zúčastniť sa prípravného sústredení vo švajčiarskom Davose v dňoch 20. až 25. februára 2012. Sústredenía sa tiež zúčastnili delegácie z Rumunska, Ruska a Hongkongu. V prvej tabuľke uvádzame výsledky dlhodobej súťaže, ktorá prebiehala počas celého sústredenia.

Meno	kraj.	day1	day2	day3	day4	$\sum$
1. Andrei Purice	ROM	360	380	330	225	1295
2. Adrian Budau	ROM	320	400	340	185	1245
3. Mihai Gheorghe	ROM	290	380	330	225	1225
4. Gavrila Alexandru	ROM	360	380	340	140	1220
5. Radu Voroneanu	ROM	390	330	320	175	1215
6. Marián Horňák	SVK	330	300	240	195	1065
7. Jakub Šafin	SVK	250	270	270	135	925
8. Johannes Kapfhammer	SUI	240	300	220	150	910
9. Wai Pan	HKG	130	330	270	175	905
10. Eduard Batmendijn	SVK	290	125	210	220	845
11. Daniil Liferenko	RUS	195	210	230	160	795
12. Dmitriy Kravchenko	RUS	105	220	150	140	615
13. Igor Labutin	RUS	165	190	100	140	595
14. Peter Mueller	SUI	45	120	230	100	495
15. Kamila Součková	SVK	190	150	0	140	480
16. Szeto Ethan	SUI	95	110	110	150	465
17. Janis Peyer	SUI	170	75	120	85	450
18. Andre Ryser	SUI	280	25	10	110	425
19. Chun Hin	HKG	150	70	0	165	385
20. Benjamin Schmid	SUI	25	115	100	100	340
21. Marco Keller	SUI	150	75	60	25	310
22. Yik Hei	HKG	85	70	10	140	305
23. Livio Ciorciaro	SUI	75	85	0	120	280
24. Michael Baumann	SUI	25	75	60	115	275
25. Cedric Muenger	SUI	95	75	50	50	270
26. Lukas Roth	SUI	100	55	50	25	230
27. Cyrill Kuenzi	SUI	75	25	0	30	130
28. Cedric Neukom	SUI	45	55	0	20	120
29. Dominik Buehler	SUI	0	65	0	30	95

V druhej tabuľke uvádzame výsledky jednokolovej súťaže „Davos Cup“. Rovnaké úlohy riešilo aj 148 stredoškolákov v Rusku, poradie v zátvorke je poradím v spoločnej výsledkovej listine.

	Meno	kraj.	úl.1	úl.2	úl.3	úl.4	$\sum$
1.	(1.) Adrian Budau	ROM	100	100	100	100	400
	Andrei Purice	ROM	100	100	100	100	400
	Gavrila Alexandru	ROM	100	100	100	100	400
4.	(8.) Eduard Batmendijn	SVK	100	100	66	100	366
5.	(9.) Radu Voroneanu	ROM	100	100	56	100	356
6.	(13.) Wai Pan	HKG	100	100	64	60	324
7.	(14.) Jakub Šafin	SVK	100	100	74	40	314
8.	(24.) Mihai Gheorghe	ROM	100	100	26	30	256
9.	(26.) Johannes Kapfhammer	SUI	40	80	100	30	250
10.	(35.) Marián Horňák	SVK	0	100	100	0	200
11.	(39.) Marco Keller	SUI	100	30	64	0	194
12.	(51.) Benjamin Schmid	SUI	40	50	56	20	166
	Janis Peyer	SUI	40	20	76	30	166
14.	(57.) Daniil Liferenko	RUS	40	30	56	30	156
15.	(59.) Michael Baumann	SUI	40	10	56	45	151
16.	(64.) Igor Labutin	RUS	40	35	70	0	145
17.	(77.) Andre Ryser	SUI	40	30	28	35	133
18.	(81.) Peter Mueller	SUI	40	25	64	0	129
19.	(86.) Dmitriy Kravchenko	RUS	40	20	60	0	120
20.	(94.) Chun Hin	HKG	40	35	10	30	115
	Yik Hei	HKG	40	15	60	0	115
22.	(97.) Kamila Součková	SVK	40	10	64	0	114
23.	(121.) Livio Ciorciaro	SUI	40	0	52	0	92
24.	(138.) Lukas Roth	SUI	40	15	0	0	55
25.	(141.) Cyrill Kuenzi	SUI	40	0	14	0	54
26.	(143.) Cedric Muenger	SUI	40	0	10	0	50
27.	(153.) Cedric Neukom	SUI	40	0	0	0	40
28.	(167.) Szeto Ethan	SUI	15	5	0	0	20
29.	(169.) Dominik Buehler	SUI	0	0	16	0	16

Česko-poľsko-slovenské prípravné sústredenie

V poradí už štrnáste súťažné stretnutie najlepších stredoškolákov z Čiech, Poľska a Slovenska sa uskutočnilo v Bílovci (neďaleko Ostravy) v Českej Republike v dňoch 10. až 16. júna 2012. Slovenskú výpravu sprevádzali Vladimír Boža a Marek Špano, obaja študenti FMFI UK. Podobne ako v mnohých predchádzajúcich ročníkoch, aj tentokrát vo výsledkoch dominujú naši severní susedia.

meno	kraj.	day1	day2	day3	day4	$\sum$
1. Bartosz Rafał Tarnawski	POL	285	270	300	300	1155
2. Bartłomiej Dudek	POL	260	290	300	300	1150
3. Karol Farbiś	POL	230	260	300	300	1090
4. Krzysztof Pszeniczny	POL	230	280	300	220	1030
5. Wojciech Krzysztof Nadara	POL	240	280	200	300	1020
6. Mateusz Gołębiewski	POL	160	260	250	300	970
7. Wiktor Kuropatwa	POL	195	260	240	115	810
8. Eduard Batmendijn	SVK	160	54	220	300	734
9. Štěpán Šimsa	CZE	135	134	250	130	649
10. Ondřej Hübsch	CZE	85	290	200	70	645
11. Andrej Mariš	SVK	80	188	200	90	558
12. Jiří Eichler	CZE	105	230	100	110	545
13. Mário Lipovský	SVK	70	89	210	50	419
14. Marián Horňák	SVK	0	44	100	270	414
15. Jozef Marko	SVK	0	92	250	40	382
16. Jan-Sebastian Fabík	CZE	40	147	30	130	347
17. Mark Karpilovsky	CZE	20	37	155	90	302
18. Askar Gafurov	SVK	60	60	10	140	270
19. Ondřej Cífka	CZE	50	60	70	0	180
20. Martin Hora	CZE	0	0	50	90	140

Stredoeurópska olympiáda v informatike

V roku 2012 sa Stredoeurópska olympiáda v informatike (CEOI) konala v maďarskom meste Tata, neďaleko nášho Komárna, v dňoch 7. až 13. júna 2012. Okrem tradičných siedmich krajín sa tohtoročnej CEOI zúčastnili aj tímy z Holandska, Izraela, Slovinska a Švajčiarska.

Slovensko na CEOI 2012 reprezentovali štyria stredoškoláci: Askar Gafurov (GAMČA Bratislava), Jerguš Greššák (ŠPMNDaG Bratislava), Vladimír Macko (Gym. Zvolen) a Jakub Šafin (Gym. Michalovce). Našu delegáciu na tejto súťaži viedli doc. RNDr. Gabriela Andrejková, CSc. a RNDr. Rastislav Krivoš-Belluš (obaja z Ústavu informatiky PF UPJŠ v Košiciach).

Súťaž nepríjemne poznačili chyby organizátorov počas druhého súťažného dňa. Kvôli spomínaným chybám neboli výsledky počas druhého dňa súťaže spravodlivé, a tak sa medaily udeľovali podľa kompromisu: každý súťažiaci dostal lepšiu z medailí, na ktoré mal nárok podľa celkového poradia a podľa poradia po prvom dni.

Traja z našich súťažiacich nás potešili ziskom bronzových medailí. Askarovi Gafurovovi do nej chýbalo prvý deň len päť bodov a druhý deň bohužiaľ patril medzi súťažiacich najviac zasiahnutých chybou organizátorov.

Podrobné výsledky našich súťažiacich uvádzame v tabuľke.

Meno	1. deň			2. deň			$\sum$	medaila
Vladimír Macko	100	0	60	60	0	90	310	bronzová
Jakub Šafin	95	40	35	40	20	43	273	bronzová
Jerguš Greššák	100	5	25	25	8	36	199	bronzová
Askar Gafurov	100	15	10	0	0	2	127	

Medzinárodné prípravné sústredenie vo Švédsku

V dňoch 6. až 11. septembra sa vo Švédsku (na ostrove Möja a následne v Stockholme) uskutočnilo historicky prvé spoločné prípravné sústredenie mladých slovenských a švédskych súťažných programátorov. Hlavným organizátorom akcie bol Lukáš Poláček – bývalý absolvent magisterského štúdia na FMFI UK v Bratislave, teraz doktorand na KTH v Stockholme. Zo Slovenska sa aj vďaka finančnej podpore *Slovenskej informatickej spoločnosti* (kúpe leteniek) mohla akcie zúčastniť štvorica súťažiacich: Eduard Batmendijn, Marián Horňák, Ján Hozza a Andrej Mariš. Traja z nich (Batmendijn, Horňák, Mariš) následne reprezentovali Slovensko na Medzinárodnej olympiáde v informatike.

Ako prednášajúci sa akcie zúčastnili viacerí švédski odborníci: Per Austrin, Ulf Lundström, Mikael Goldmann, Oskar Werkelin Ahlin. Okrem nich mal prednášku aj hlavný organizátor Lukáš Poláček a ako pozvaný externý prednášajúci sa akcie zúčastnil aj Michal Forišek z FMFI UK. Súčasťou akcie boli aj tri súťažné kolá: dve teoretické a jedno praktické. Celkové výsledky súťaží uvádzame v priloženej tabuľke.

	Meno	kraj.	teor. 1	bonus	teor. 2	prax	$\sum$
1.	Sannemo	SWE	34	3	35	48	120
2.	Lindholm	SWE	36	0	38	40	114
3.	Wiman	SWE	26	0	37	48	111
4.	Hozza	SVK	34	0	34	40	108
5.	Horňák	SVK	24	0	38	32	94
	Batmendijn	SVK	27	3	32	32	94
7.	Lenngren	SWE	20	0	25	32	77
8.	Odenman	SWE	20	0	23	32	75
9.	Mariš	SVK	22	0	27	24	73
10.	Granberg	SWE	19	0	20	32	71
	Grensjö	SWE	10	0	21	40	71
12.	Klein	SWE	15	0	20	24	59
13.	Wärnberg	SWE	11	0	20	24	55
14.	Aule	SWE	10	0	18	24	52
15.	Bäckström	SWE	10	0	19	16	45

Medzinárodná olympiáda v informatike

Zlatú a dve bronzové medaily získali slovenskí študenti na Medzinárodnej olympiáde v informatike 2012 v Taliansku.

Tohtoročná medzinárodná olympiáda v informatike (International Olympiad in Informatics – IOI 2012) sa konala v dňoch 23. – 30. septembra 2012 v mestách Sirmione a Montechiari v Taliansku. Zúčastnilo sa na nej 310 oficiálnych súťažiacich z 81 krajín celého sveta, ktorí prešli sitom národných súťaží. Odhad počtu všetkých súťažiacich počnúc domácimi kolami až po medzinárodné kolo je 250 tisíc študentov.

V roku 2012 Slovensko reprezentovali štyria súťažiaci:

- Eduard Batmendijn, Cirk. gymnázium sv. Mikuláša, Stará Ľubovňa
- Marián Horňák, Gymnázium Párovská, Nitra
- Andrej Mariš, Gymnázium Piaristická, Nitra
- Jakub Šafin, Gymnázium P. Horova, Michalovce

Zvyšok slovenskej výpravy tvoril člen medzinárodného odborného výboru súťaže RNDr. Michal Forišek, PhD. (FMFI UK), vedúca delegácie doc. RNDr. Gabriela Andrejková, CSc. (PF UPJŠ) a pedagogický vedúci Bc. Vladimír Boža (FMFI UK).

Súčasťou programu mimo súťažných dní bola prehliadka Benátok a pre súťažiacich tiež návšteva neďalekého zábavného parku. Lídri mali možnosť zúčastniť sa na medzinárodnej konferencii Youth Talents Conference. Ako na záverečnom ceremoniáli povedal prezident IOI dr. Richard Forster, organizátorom sa skutočne podarilo spraviť „pravú talianskú“ medzinárodnú olympiádu.

Z výsledkov našich súťažiacich musíme vyzdvihnúť zlatú medailu Eduarda Batmendijna. Edo, v kolektíve známy ako Baklažán, má pred sebou ešte potenciálne tri ďalšie medzinárodné olympiády. Želáme mu, aby sa mu na nich darilo aspoň tak dobre, ako na tejto. Kompletné výsledky uvádzame v tabuľke.

Meno	1. deň			2. deň			$\sum$	medaila
Eduard Batmendijn	72	100	60	55	100	49	436	12. miesto, zlato
Marián Horňák	47	0	100	0	0	49	196	104. miesto, bronz
Jakub Šafin	47	0	60	32	17	17	173	137. miesto, bronz
Andrej Mariš	0	0	100	32	8	0	140	172. miesto

IUVENTA – Slovenský inštitút mládeže je príspevková organizácia priamo riadená Ministerstvom školstva, vedy, výskumu a športu SR. Pripravuje a riadi množstvo zaujímavých programov a projektov pre mladých ľudí, pracovníkov s mládežou a ľudí zodpovedných za mládežnícku politiku. Snaží sa o to, aby mladí ľudia poznali svoje možnosti, boli aktívni a pracovali na sebe tak, aby boli raz úspešní a uplatnili sa na trhu práce. IUVENTA vychováva mládež k ľudským právam, podporuje rozvoj dobrovoľníctva, vzdelávacie programy a tiež mladé talenty.

IUVENTA je realizátorom národného projektu v oblasti práce s mládežou KomPrax – Kompetencie pre prax, ktorý je podporený z Európskeho sociálneho fondu. Zároveň administruje Programy finančnej podpory aktivít detí a mládeže MŠVVaŠ SR ADAM, je národnou agentúrou programu EÚ Mládež v akcii a národným partnerom európskej informačnej siete pre mládež Eurodesk.

Kontakt: IUVENTA – Slovenský inštitút mládeže
Búdková 2
SK-811 04 Bratislava 1
tel.: (421-2) 59 29 61 12
fax: (421-2) 59 29 61 23
e-mail: iuventa@iuventa.sk
web: www.iuventa.sk, www.facebook.com/iuventa

RNDr. Michal Forišek PhD., Slovenská komisia OI
Dvadsiaty siedmy ročník Olympiády v informatike
Vydavateľ: IUVENTA – Slovenský inštitút mládeže
Rok vydania: 2012
Tlač: DOLIS, s.r.o.
Rozsah: 128 strán
Náklad: 300 výtlačkov
ISBN: 978-80-8072-122-0
Neprešlo jazykovou úpravou