PRASK
III. ročník, 2016/17
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

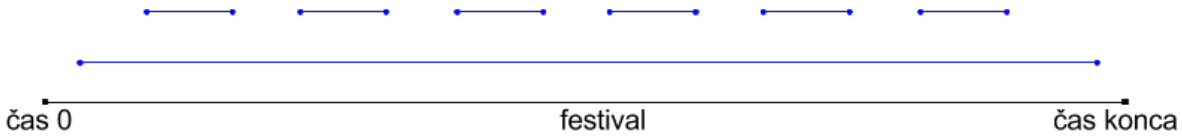
Vzorové riešenia 1. kola zimnej časti

1. Pohodový festival

vzorák napísal Žaba
(max. 15 b za riešenie)

Podúloha a)

V tomto prípade je situácia veľmi jednoduchá. Lebo bez ohľadu na to ako umiestnime naše koncerty, tak je jasné, na ktorý z nich Syseľ pôjde – na ten, čo začína najskôr. Chceli by sme preto Sysľa za tento výber potrestať. No a nie je nič ľahšie, ako ho už na žiadny ďalší koncert nepustiť, čo môžeme docieliť tak, že koncert, ktorý začína ako prvý, skončí ako posledný. Tým pádom vieme počas trvania tohto predstavenia usporiadať veľké množstvo kratších koncertov, ktoré sa navzájom neprekrývajú a teda keby sa ich Syseľ zúčastnil, bol by spokojnejší.



Podúloha b)

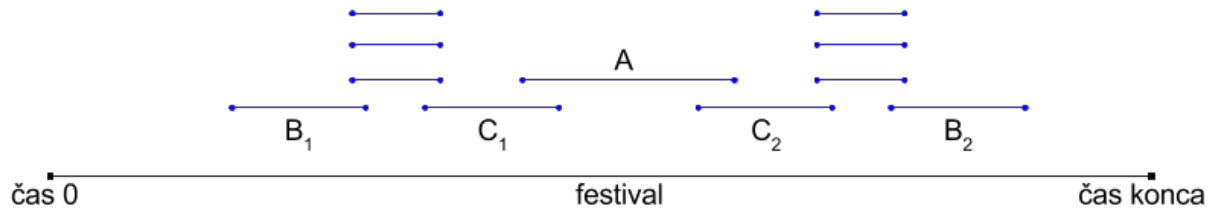
Syseľ si vyberá koncerty podľa toho, s koľkými inými koncertami sa prekrývajú. A snaží sa tvrdiť, že ak zakaždým pôjde na ten, ktorý sa prekrýva s najmenším množstvom iných koncertov, určite ich navštívi najväčší možný počet.

Ak sa snažíme vymyslieť protipríklad, je to o niečo zložitejšie ako v podúlohe a), pretože to ako rozmiestnime koncerty ovplyvňuje Sysľov výber. Najlepšie je naozaj si zobrať papier a pero a skúšať rôzne možnosti. Dobrý prístup môže byť zvoliť si jeden koncert (označme si ho A), ktorý bude mať najmenší počet prekryvov. Následne sa snažiť dokresliť ostatné tak, aby sa viac oplatilo **neísť** na koncert A.

Pritom nám môžu pomôcť ešte dve pozorovania. Ak sa nejaké koncerty odohrávajú napravo od koncertu A (neskôr), tak symetricky s nimi môžeme to isté robiť aj naľavo. Koncert A sa teda stane stredom nášho riešenia. Takýto postup nám zjednoduší hľadanie protipríkladu a intuitívne sa zdá, že nám to nič nepokazí.

No a nakoniec ešte musíme vyriešiť to, aby mal koncert A naozaj najmenej prekryvov. K tomu nám vie pomôcť to, že vieme vytvoriť veľké množstvo koncertov, ktoré sa odohrávajú v ten istý čas. Ich úlohou bude naozaj iba umelo nafúknuť počet prekryvov pre nejaký iný koncert.

Jedno možné riešenie môže vyzerať takto:



Všimnime si, že Syseľ si najskôr vyberie koncert A, ktorý má iba dva prekryvy. Potom ho prestane uvažovať, spolu s koncertami C_1 a C_2, ktoré sa s ním prekrývajú. No a potom postupne vyberie napríklad koncerty B_1 a B_2. Zúčastní sa teda iba troch koncertov – A, B_1 a B_2.

Ak by však nešiel na koncert A, mohol by sa zúčastniť až štyroch koncertov – B_1, C_1, C_2 a B_2.

Podúloha c)

Oba Sysľove pokusy boli neúspešné, musíme preto prísť s nejakým iným riešením. Prekvapivo, správne riešenie vôbec nie je ťažké a naviac je pomerne intuitívne. Pokúsme sa naň spoločne prísť.

Predstavme si nejaký zoznam koncertov, ktoré sa v daný deň majú konať na festivale. Na žiadny z nich zatiaľ nepoďme, skúsme čakať a sledovať čo sa stane. Postupne niektoré z týchto koncertov začnú. Nechceme sa však riadiť podľa času kedy začínajú, lebo ako sme videli v podúlohe a), tak to neviedlo k dobrému riešeniu. V nejakom momente však prvý z koncertov skončí, označme si ho písmenom A.

A toto je presne moment, v ktorom sa chceme zamyslieť nad tým, či sme sa koncertu A nechceli zúčastniť. Zoberme si ľubovoľné optimálne riešenie. Teda také, v ktorom Syseľ navštívy najväčší možný počet koncertov. V tomto riešení sa Syseľ zúčastní niektorého koncertu ako prvého – nech je to koncert B.

Ak by B začínal neskôr ako končí koncert A, naše riešenie by nebolo optimálne. Mohli by sme k nemu totiž pridať koncert A, ktorý by sa určite s ničím neprekrýval, keďže B bol v tomto riešení prvý koncert a A končí skôr ako začína B. Týmto pridaním by sme však dostali ešte lepšie riešenie, čo je nemožné, lebo naše vybrané riešenie bolo optimálne.

V každom správnom riešení teda platí, že prvý koncert, ktorého sa Syseľ zúčastni (B) začne skôr ako skončí koncert, ktorý končí ako prvý (A). To ale znamená, že koncert B nemôže skončiť skôr ako koncert A. No a všetky zvyšné koncerty v tomto optimálnom riešení začínajú až potom ako skončí koncert B (lebo sa s ním neprekrývajú) a teda všetky začínajú aj po skončení koncertu A. V tomto momente si môžeme uvedomiť, že v našom riešení môžeme nahradiť koncert B koncertom A a naše riešenie sa určite nepokazí a bude obsahovať rovnaký počet koncertov, teda bude rovnako dobré.

Táto úvaha vedie k veľmi jednoduchému riešeniu. Syseľ si vyberie koncert, ktorý končí ako prvý a tohto koncertu sa zúčastní. Následne vyškrtne tento koncert a aj všetky, ktoré sa s ním prekrývajú, lebo sa ich už nebude môcť zúčastniť. Na zvyšné koncerty potom použije vyššie uvedenú myšlienku a teda si opäť vyberie najskôr končiaci koncert, ktorého sa zúčastní. Takto bude pokračovať, až kým nevyškrtne všetky koncerty.

Toto riešenie je pomerne intuitívne preto, lebo čím skôr skončí koncert, na ktorom sa Syseľ nachádza, tým viac času mu ostane na návštevu zvyšných koncertov. Takéto riešenie voláme "pažravé" (po anglicky greedy). Ako ste však videli v podúlohách a) a b), pri vymýšľaní takýchto riešení si musíte dať pozor, aby ste sa nenechali okmalať zdanlivo dobrou úvahou a svoju myšlienku si vždy overte tak ako v podúlohe c) – porovnaním s ľubovoľným optimálnym riešením.

2. Pestovanie ovocia v Absurdistane

vzorák napísal Hodobox
(max. 15 b za riešenie)

V tomto vzorovom riešení postupne vyriešime jednotlivé podúlohy a pri každej skúsime načrtnúť aj postup, ktorým sme mohli dospieť k správnemu riešeniu.

a) $1, 2, 3, 1, 2, 3, 1, \ldots$

V prvom roku máme mať jednu rastlinu. Tu nemáme príliš nad čím rozmýšľať – pomenujeme ju `A` a zasadíme ako prvú. Takisto vieme, že v druhom roku máme mať dva plody. Tie nutne musí urodiť `A`, preto jej zatiaľ dáme predpis `A` → `BC`. Z týchto dvoch sa nám majú urodiť 3 plody. Tie sa kľudne môžu urodiť z jednej z nich, a druhú necháme bez plodov: `C` → `0`.

Vieme, že v treťom roku musí vyrásť rastlina, na ktorej sa opäť urodí `A`, aby sa celý tento cyklus mohol opakovať. Túto rastlinu nazveme `D` a bude mať predpis `D`→`A`. Rastlina `D` sa musí urodiť na rastline `B` a zvyšné dve rastliny, ktoré sa na `B` urodia by mali na ďalší rok zahynúť, aby nám ostala iba rastlina `A`. No a vymierajúcu rastlinu už máme v podobe `C`. Preto posledné pravidlo bude `B`→`DCC`.

Rastliny, ktoré budeme postupne dostávať, sú `A`, `BC`, `DCC`, `A` ... Toto riešenie je navyše správne, lebo prvé štyri roky tvoria postupnosť $(1, 2, 3, 1)$ a v štvrtom roku máme rovnakú rastlinu `A` ako v prvom roku. Nutne sa teda táto postupnosť bude opakovať.

b) $x, +2, -1, +2, -1, +2, \ldots$

Striedavo nám v jednom roku pribúdajú dve rastliny, a jedna ubúda. Ubúdanie rastliny vyriešime rastlinou `C`→`0` z minulej podúlohy. Chceme aby nám každý druhý rok vyrástla jedna takáto rastlina. Zároveň potrebujeme nejakú rastlinu `A`, z ktorej v predchádzajúci rok vyrastú tri rastliny (aby sme mali dokopy +2 rastlín). Jedna z týchto rastlín bude `C`, ktorá nám zabezpečí žiadaný pokles o jeden na ďalší rok. Máme teda `A`→`BCD` a `C`→`0`. Zvyšné dve rastliny `BC` musia dokopy urodiť dva plody – ak by urodili menej alebo viac, nemali by sme celkový zisk -1, ktorý nám už zabezpečuje `C`. Takisto vieme, že v nasledovnom roku budeme chcieť vypestovať rastlinu `A`, aby sme tento proces začali odznova. Dajme si teda `B`→`A`. `A` sme pritom navrhli tak, aby samo o sebe spustilo proces +2, -1 v ďalších rokoch. To znamená že zostávajúca rastlina `D` by už nikdy nemala mať dopad na počet plodov – mala by sa len zachovávať. Predpíšeme teda `D`→`D`.

Celé riešenie je teda A→BCD, B→A, C→0 a D→D. V každom druhom roku budeme v rovnakej situácií ako na začiatku, len budeme mať zakaždým o jednu trvanlivú rastlinu D naviac. Jediná rastlina A v takomto roku potom spôsobí žiadané zmeny +2, -1 v nasledujúcich rokoch, a tento proces sa bude opakovať.

c) $x, *4, /2, *4, /2, \ldots$

Čím ďalej, tým častejšie vytvárame rastliny na plnenie špecifických úloh. Odteraz si teda budeme rastliny aj pomenovávať v závislosti od funkcie, ktorú budú plniť.

Prvou takou rastlinou bude rastlina 'násobič' A. Presný predpis tejto rastliny dopredu nevieme, chceme však rastlinu ktorá sa do budúceho roku rozmnoží na štyri. Jednu takúto rastlinu zasadíme v prvom roku a tá sa nám rozmnoží na štyri, zatiaľ neurčené, rastliny. Aké rastliny by boli vhodné? No z týchto štyroch rastlín ktoré vyrastú v druhom roku, chceme mať len dva plody, aby sme do tretieho roku mali žiadanú zmenu počtu /2.

Zároveň, rovnako ako v minulej podúlohe, sa nám tento proces začne odznova – následne budeme chcieť počet rastlín zoštvornásobiť. Na to sme už vytvorili násobiča A, teda v tomto roku budeme chcieť aby obe vyrastené rastliny boli A. Zatiaľ teda vieme, že počas prvých rokov by mala situácia vyzerať A→????→AA. Aspoň z dvoch ? nám do ďalšieho roka nesmie nič narásť – potrebujeme akúsi rastlinu 'samovrah'. Takú však máme z predchádzajúcej podúlohy. Je ňou rastlina C→0.

Teda A→??CC→AA. Teraz už vidíme jednoduché riešenie – obe zostávajúce ? budú rastlina 'späť' B→A, ktorej úloha je jeden rok prespať a v ďalšom roku sa zmeniť na násobiča A.

Celkovo teda dostávame A→BBCC→AA. V treťom roku máme rovnaké rastliny ako v prvom, akurát ich je dvojnásobne viac. Ďalšie dve roky teda budú opäť vyzerať rovnako – dostaneme zmenu $*4$ a potom /2, pričom sa nám počet rastlín A opäť zdvojnásobí.

d) $x, *2, +1, *2, +1, *2, \ldots$

Táto podúloha sčasti kombinuje dve predošlé podúlohy – v jeden rok chceme počet rastlín násobiť, a hneď nato zmeniť o konštantú hodnotu.

Na začiatok sa pokúsme vyriešiť úlohy, kde by zadaná postupnosť vyzerala $(x, *2, +0, *2, +0 \ldots)$. Teda počet rastlín sa nám každý druhý rok zdvojnásobí. Zdvojnásobenie je ľahké, stačí použiť rastlinu C→CC. Avšak takéto rastliný sa budú násobiť každý rok. Tak ako v predchádzajúcej podúlohe si však môžeme pridať rastlinu 'späť', ktorá zaručí, že v nasledujúci rok sa nič nezmení, teda naozaj bude prírastok +0.

Rastliny teda môžu vyzerať nasledovne: C→DD a D→C. C teda vytvorí dvoch späťov D, ktorý sa nasledujúci rok iba prepíšu na 'násobiča' C, čím nespôsobia zmenu v párne roky, ale zaručia, že v nepárny rok sa počet rastlín opäť zdvojnásobí. Rastlina D teda slúži len na zdržanie nášho násobenia.

Do vzorového riešenia však musíme pridať aj to, aby sa každý párny rok pridal jeden nový plod. Keďže je len jeden, stačí si na to vyhradiť jednu rastlinu. Táto rastlina však musí vedieť, či je rok párny alebo nepárny. Ale tak ako predtým, toto vieme ľahko vyriešiť pomocou striedania dvoch rastlín. Presnejšia, keby sme mali A→B a B→A, pričom začíname s rastlinou A, tak vieme, že vždy keď sadíme A je rok nepárny a keď B tak rok párny.

Ostáva už len domyslieť, čo pridať na pravú stranu oboch týchto šipiek. Keď sadíme rastlinu B, potrebujeme aby nám vyrátlo o jednu rastlinu viac. To znamená, že okrem zadaného A musí vyrásť aj nejaká iná rastlina ?. Táto rastlina sa potom v ďalšom roku musí zdvojnásobiť. A na to predsa slúži rastlina C a teda B→AC.

No a keď máme rastlinu A, tak počet rastlín sa musí v daný rok zdvojnásobiť. To znamená, že A musí okrem B vytvoriť ešte jednu rastlinu ?. No a táto rastlina má v ďalšom roku iba nahradiť samú seba, keďže o +1 sa postará B, a potom sa začať každý druhý rok zdvojnásobovať. A takto predsa funguje rastlina D.

Finálne riešenie je teda A→BD, B→AC, C→DD a D→C, pričom nám stačí ako prvú zasadiť rastlinu A.

Aby sme ukázali, že toto riešenie je správne, môžeme ukázať, že v nepárne roky sadíme iba rastliny A a C, ktoré sa v daný rok naozaj zdvojnásobia a v nepárne roky sadíme iba rastliny B a D, ktoré nahrádzajú samé seba. S výnimkou, že B vytvorí ešte jednu rastlinu naviac, aby sme získali +1, ale to je dobre, lebo máme práve jednu rastlinu B.

e) n-té Fibonacciho číslo

Pripomeňme si najskôr, ako počítame Fibonacciho čísla. Označme si n-té Fibonacciho číslo ako F_n. Vieme, že $F_0 = 1$ a $F_1 = 1$. No a na základe týchto dvoch hodnôt už vieme vypočítať všetky väčšie, keďže n-té Fibonacciho číslo dostaneme ako súčet predchádzajúcich dvoch, teda $F_n = F_{n-1} + F_{n-2}$.

Pokúsme sa využiť túto vlastnosť. Predstavme si, že sa nám podarilo mať v n-tom roku vypestovaných F_n rastlín. Tieto rastliny si vieme rozdeliť na dve kôpky – jednu veľkosti F_{n-1} a druhú veľkosti F_{n-2}. Nech všetky rastliny v prvej kôpke sú rastliny A a všetky rastliny v druhej kôpke rastliny B. Tieto rastliny následne zasadíme.

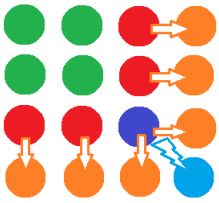
Čo musí platiť, aby sa nám z nich urodilo dokopy F_{n+1} rastlín? Vieme, že $F_{n+1} = F_n + F_{n-1}$, teda sa nám má urodiť F_n rastlín `A` a F_{n-1} rastlín `B`. Vidíme, že ak by sa z každej rastliny `A`, ktorých máme zatiaľ iba F_{n-1} urodila jedna rastlina `B`, tak v ďalšom roku budeme mať naozaj F_{n-1} rastlín `B`. No a aby sa nám urodilo F_n rastlín `A`, musí z každej zasadenej rastliny vyrásť rastlina `A`.

Táto myšlienka nás vedie k dvom veľmi jednoduchým pravidlám: `A`→`AB` a `B`→`A`, pričom začínať budeme s jednou rastlinou `A`. Postupnosť rastu teda bude `A`→`AB`→`AAB`→`AABAB`...

Zopakujme si teda ešte raz, čo presne robí naše riešenie. Po n-tom roku máme vypestovaných F_n rastlín. Z toho presne F_{n-1} je rastlín `A` a presne F_{n-2} je rastlín `B`, čo teda v súčte naozaj dáva hodnotu F_n. Keď tieto rastliny zasadíme, tak z každej rastliny vyrastie jedna rastlina `A`, teda budeme mať F_n rastlín `A` a z každej rastliny `A` vyrastie rastlina `B`, takže budeme mať F_{n-1} rastlín `B`. Dokopy budeme mať $F_n + F_{n-1} = F_{n+1}$ rastlín, čo je presne počet čo sme chceli. Naviac, opäť bude platiť, že počet `A`-čok je predchádzajúce Fibonacciho číslo a počet `B`-čok to ešte pred tým. Túto úvahu teda budeme môcť použiť znova a naše riešenie je korektné.

f) n^2

V tejto úlohe bola nápomocná vizualizácia – nie nadarmo sa číslo n^2 nazýva štvorec čísla n. Naše rastliny teda zasaďme do štvorca, ktorý má n riadkov a n stĺpcov. Začínať budeme s jednou rastlinou. No a aby sme zo štvorca $n \times n$ dostali štvorec $(n+1) \times (n+1)$, potrebujeme k tomu menšiemu štvorcu pridať jeden riadok na spodok a jeden stĺpec napravo.



Na obrázku je znázornený prechod záhrady z tretieho do štvrtého roku. Oranžové a svetlomodrá rastlina sú tie, ktoré v tomto roku museli vyrásť, všetky ostatné sa museli zachovať. Ostáva z obrázku vyčítať, ako vyzerajú pravidlá pre jednotlivé rastliny.

Zelené rastliny (`Z`) nemuseli spraviť nič, stačilo, že sa zachovali aj do ďalšieho roka. Preto pridáme pravidlo `Z`→`Z`. Zaujímavejšie je to však s rastlinami, ktoré boli na kraji štvorca 3×3. V ich blízkosti totiž museli pribudnúť nové rastliny. A dáva zmysel, aby sa práve červená rastlina postarala o to, že veľa nej niečo vyrastie.

Pozrime sa teda bližšie na červenú rastlinu (`C`). Nestačí, že sa táto rastlina zachová, musí spraviť aj o jednu rastlinu naviac, rastlinu, ktorá je na obrázku označená oranžovou. Teda potrebujeme pravidlo `C`→`??`. Ostáva zistiť, čo dať na miesto otáznikov. Miesto, kde leží červená rastlina nebude v ďalšom roku na spodnom ani pravom kraji. To znamená, že od toho momentu sa toto miesto už len má nahrádzať samo sebou, čo úspešne robí rastlina `Z`. A čo má robiť novovzniknutá oranžová rastlina? Keďže bude v ďalšom roku na kraji, označili by sme ju červenou farbou, lebo presne tak by sa mala správať. Preto dostaneme pravidlo `C`→`ZC`.

Trochu špeciálne sa bude správať modrá rastlina `M`. Tá má totiž až dvoch oranžových susedov a naviac sa musí postarať o tom, aby sme zaplnili novovzniknutý pravý dolný roh. A k tomu všetkému sa musí aj zachovať. Preto potrebujeme pravidlo `M`→`????`. Je jasné, že obe nové oranžové rastliny sa v ďalšom roku musia chovať ako `C` a preto dva zo štyroch `?` nahradíme týmto písmenom. Na mieste aktuálnej modrej rastliny sa ďalej už nič diať nebude, a teda tam patrí rastlina `Z`. No a posledný otáznik musíme nahradiť `M`. Pretože aj v ďalšom roku budeme potrebovať, aby sa niektorá rastlina postarala o pravý dolný roh.

Naše riešenie teda bude začínať s jednou rastlinou `M` a bude mať pravidlá `M`→`ZCCM`, `C`→`ZC` a `Z`→`Z`.

3. Pofiderné kasíno

vzorák napísal Prefix
(max. 15 b za riešenie)

Hlavným cieľom tejto úlohy bolo ukázať vám, že mnohé programy obsahujú chyby, ktorými sa dá tento program prinútiť robiť niečo iné, ako jeho tvorca zamýšľal. Vedieť hľadať takéto chyby je veľmi užitočné, hlavne pri opravovaní vlastných riešení. Jednou z veľkých chýb bývajú náhodné čísla – na počítači sa totiž nedajú generovať skutočne náhodné čísla, preto sa používajú takzvané pseudonáhodné čísla.

Ľubovoľný program sa v každom momente nachádza v nejakom stave – tento stav si môžme predstaviť ako celú pamäť programu, teda hodnoty jeho jednotlivých premenných, a číslo riadka, ktorý sa momentálne vykonáva. Je ľahké ukázať, že stavov je konečne veľa – program má predsa obmedzenú pamäť, napríklad veľkosťou pamäte RAM, a program tiež nemôže mať nekonečne veľa riadkov. Zároveň má program pre každý stav jasne definované, do akého ďalšieho stavu má prejsť. Počítač totiž nemá vlastnú vôľu a nemá sa podľa čoho rozhodovať.

Musí sa pevne riadiť tým, čo sme mu určili. V tomto momente si možno poviete, že máme predsa príkazy, z ktorých vieme skočiť na rôzne miesta v programe – napríklad taký príkaz `if`. Ale `if` sa tiež rozhoduje podľa hodnôt príslušných premenných, teda podľa aktuálneho stavu.

Keby to samozrejme bolo také jednoduché, nevedeli by sme s počítačmi nijako interagovať a boli by nám vcelku nanič – chceme, aby sme vedeli zvonka meniť stav programu, napríklad pomocou klávesnice, myšky, momentálneho času, alebo nejakých iných senzorov. Väčšina týchto operácii je ale dosť drahá na čas – program na to, aby mohol získať nejaké vonkajšie informácie musí komunikovať s operačným systémom, čo trvá o dosť dlhšie ako keď si len počíta nejaké veci sám pre seba.

Z tohoto dôvodu je ťažké na počítači generovať skutočne náhodné čísla – program má konečné množstvo stavov, čiže ak by sme naprogramovali náhodný generátor, po nejakom čase by sa buď zastavil a prestal nám dávať náhodné čísla, alebo by sa dostal do stavu, v akom už bol, a začal by nám dávať znova tie isté čísla, čo teda rozhodne náhodné nie je.

Tento problém sa dá riešiť dvoma spôsobmi – prvým je, že použijeme generátor ktorý má dosť veľké množstvo stavov a teda by sa zacyklil po veľmi dlhom čase. Tieto čísla sú ale stále pseudonáhodné. Ak chceme na počítači získavať skutočne náhodné čísla[1], musíme od niekiaľ získať túto náhodnosť ktorá občas pomení stavy v programe. Tu prichádzajú do hry vonkajšie vstupy – napríklad pohyby myši, senzory, ktoré merajú teplotu procesora, momentálny čas atď.. Ako sme už ale zmienil, zisťovať tieto veci trvá dlhšie a preto si väčšinou vystačíme s nejakými dosť dobrými pseudonáhodnými generátormi.

Podúloha a)

Náhodný generátor v podúlohe a) mal veľmi málo stavov, preto stačilo len pozerať na to, aké čísla generuje. Po chvíli ste si mohli všimnúť, že sa začali opakovať a vtedy je už jasné, čo sa stane najbližšie – pozrieme sa, aké číslo sa po aktuálnom vypísalo naposledy a toto číslo tipneme.

Podúloha b)

Táto úloha vám mala ukázať, že pre počítače neplatia rovnaké pravidlá teoreticky aj prakticky – máme kód, ktorý vyzerá byť teoreticky správny, no má závažnú chybu. Problém je, že premenná typu `int` nemôže obsahovať ľubovoľne veľké číslo. Takáto premenná má len 32 bitov a teda obmedzený rozsah. Na väčšine požítačov má `int` so znamienkom rozsah od $-2\,147\,483\,648$ do $2\,147\,483\,647$. V pamäti počítača je `int` reprezentovaný 32 bitmi, ktoré môžu mať hodnoty 0 alebo 1. Na to aby sa odlišili záporné čísla od kladných, sa používa prvý bit – všetky kladné čísla ho majú nastavený na 0, záporné zasa na 1.

Na zapisovanie kladných čísel sa používa binárna sústava, otázka však je, ako zapísať čísla záporné. Najjednoduchšia možnosť je, aby číslo $-x$ bolo zapísané rovnako ako x, akurát by sa líšilo v prvom bite, ktorý reprezentuje znamienko. Takýto prístup má však dva problémy. Číslo 0 je zapísané dvoma spôsobmi (hoci -0 neexistuje) a hlavne vypočítať $-x + x$ (a súčty celkovo) je za pomoci binárnych operácii veľmi zložité – môžete si to vyskúšať na papiery.

Preto sa používa iný spôsob – dvojkový doplnokový kód. Hodnota $-x$ sa z x určí nasledovne. Všetky jednotky v pôvodnom čísle zmeníme na 0, všetky 0 na jednotky, a následne k tomuto novému číslu pripočítame 1. Sčítavať čísla, teraz môžeme pomocou klasického binárneho súčtu. Súčet čísel x a $-x$ teraz určite vyjde 0 – ak by sme nepripočítali 1, dostali by sme súčtom ich zápisov číslo kde by všetkých 32 bitov bolo nastavených na 1. Keďže ale my pripočítavame ešte 1, pretečie nám súčet do ďalšieho bitu, ktorý neexistuje, a v 32 bitoch, ktoré nás zaujímajú, nám zostanú teda samé 0. A naviac, aj 0 má v tomto prípade jediný zápis – 32 bitov nastavených na 0.

Našou úlohou je teraz nájsť jedno špeciálne číslo. Vieme totiž, že $-z$ sa v programe vyhodnotí ako dvojkový doplnok z. A existujú len dve čísla, ktorých dvojkový doplnok v 32 bitoch sa rovná pôvodnému číslu – je to číslo $-2\,147\,483\,648$ a číslo 0.

Takže chceme dosiahnúť, aby premenná z mala nakoniec hodnotu práve $-2\,147\,483\,648$, keďže hodnotu 0 dosiahnuť nevieme. Prvým krokom je zistiť hodnotu a, čo vieme spraviť zadaním $x = 1, y = 0$. V tomto prípade nám kasíno povie hodnotu a. S jej pomocou už nie je vypočítať také x a y, aby sa $x \cdot a + y = -2\,147\,483\,647$ a keďže toto číslo vynásobené -1 bude opäť to isté a teda záporné, podarí sa nám vyhrať.

Podúloha c)

Tento náhodný generátor mal už viac stavov, stále sa ale dali všetky vyčísliť – po maximálne 9973 volaniach by sa už určite dostal do stavu, v ktorom už bol a od toho momentu by bol zacyklený.

Vieme to však robiť aj rýchlejšie ako skúšať 9973 čísel. Stačí si uvedomiť, že pri hodnotách a a b sa stačí pozerať na ich zvyšok po delení 9973. Ak by napríklad v jednom prípade bolo $a = 1$ a v druhom by bolo

[1] A aj tu je stále otázka, či sú dostatočne náhodné.

$a = 9974$, tak výsledok by bol rovnaký.

A keďže možností pre a a b nie je veľa, len 9974 pre každé z nich, dokopy nám stačí vyskúšať iba $9974 \cdot 9974 = 99\,480\,676$ možností. A na to, aby sme určili, ktorá z týchto možností je správna nám stačia tri prvé čísla, ktoré vytvorí generátor v kasíne. S pomocou krátkeho programu, ktorý vyskúša všetky tieto možnosti a zistí, či sedia na vypísané čísla dostaneme hodnotu ďalšieho tipu za pár sekúnd.

Listing programu (Python)

```
#Nacitame 3 cisla - pouzijeme pythonovsky trik
x,y,z = [int(x) for x in input().split()]
m = 9973

#Skusime pre kazde a,b ci by davali take nahodne cisla ako sme dostali
for a in range(m):
    for b in range(m):
        if (a*x+b)%m == y and (a*y+b)%m == z:
            print(a,b,(a*z+b)%m)
```

Podúloha d)

Tu to začalo byť ťažké – zatiaľčo predtým sme sa snažili oklamať slabo navrhnuté generátory, pythonovský `randint` má dosť veľké množstvo stavov, ktoré sa len tak rýchlo nezopakujú.

Prvý riadok kódu je seed generátora. Seed náhodného generátora je nastavenie jeho stavu na nejakú hodnotu. Ako sme si vysvetlili, náhodný generátor sa vždy nachádza v nejakom stave. Pomocou funkcie seed tento stav explicitne nastavíme. Z toho vyplýva, že ak dva rovnaké pseudonáhodné generátory nastavíme na rovnaký seed, budú nám generovať rovnaké čísla.

V kóde vidíme, že na začiatku sa generátor naseeduje na aktuálny čas. Toto je dosť bežná prax hlavne v `C` a `C++` – na začiatku programu v nich má generátor vždy rovnaký stav, preto by pri spustení dával vždy tie isté čísla. Musíme teda zobrať niečo mimo programu, a aktuálny čas je na to dostatočne vhodný – môžme totiž očakávať, že program sa spustí väčšinou v iných časoch, pretože len málokto pustí ten istý program dva krát presne naraz. Toto ale nie je dokonalá technika – hacker by mohol napríklad zmeniť časti svojho systému tak, aby funkcia `time` vracala vždy rovnaký čas, alebo by mohol skúsiť program spustiť viac krát naraz.

V pythone sa seed časom deje automaticky. Keď importneme knižnicu `random`, automaticky zoberie momentálny čas a použije ho ako seed. Do nášho ukážkového kódu sme to napísali explicitne len preto, aby vám to trochu uľahčilo riešenie úlohy.

Je jasné, že ak by sa nám podarilo zistiť čas, ktorým sa generátor naseedoval, vieme potom nastaviť vlastný náhodný generátor na rovnaký seed a generovať rovnaké čísla ako server.

Opäť si vieme všimnúť, že možných seedov nie je veľa. Ich rozsah vieme zistiť vložením vypísaných časov do online konvertera časov na Unix timestamp, ktoré vracia funkcia time. Keď vkládame časy do konvertera, musíme si však dať pozor. Konverter očakával od nás čísla v časovom pásme GMT, zatiaľčo slovenský čas je buď GMT+1 alebo GMT+2, podľa toho, či je letný alebo zimný. Preto skôr ako sme daný čas konvertovali, museli sme od neho odpočítať 2 hodiny.

Potom si už len stačí vypísať prvých pár čísel, ktoré vygeneruje kasíno, nech vieme, aké čísla hľadať. Následne vieme postupne vyskúšať všetky možné seedy, pre každý vygenerovať viac ako 1000 čísel, a vyskúšať, či sa tých pár, ktoré nám povedalo kasíno zhoduje s nejakými za sebou idúcimi z tých, čo sa si vygenerujeme pre daný seed. Ak áno, objavili sme správny seed a zistiť hodnotu nasledujúceho čísla je už ľahké.

Listing programu (Python)

```
from random import randint,seed
#Najprv zoberieme prvy mozny seed a posledny mozny seed
zaciatok_cas, koniec_cas = [int(x) for x in input().split()]
#Zoberieme nejaku postupnost nahodnych vygenerovanych cisel
randomNums = [int(x) for x in input().split()]

# Vyskusame postupne vsetky mozne hodnoty seedu
for skusany_seed in range(zaciatok_cas,koniec_cas+1):
    seed(skusany_seed)
    arr = []
    for x in range(1500):
        # Tu zaciname - chceme naplnit nas zoznam momentalnych nahodnych cisel
        if len(arr) < len(randomNums):
            arr.append(randint(0,1000000))
        # Pokial ho uz mame naplneny, ked pridavame nove cislo tak uberieme
        # prve a pridame nove na koniec
        else:
            arr = arr[1:]
            arr.append(randint(0,1000000))
        # Otestujeme ci nas momentalny zoznam k nahodnych cisel sa rovna k ktore
        # hladame
        if arr == randomNums:
            print(skusany_seed, randint(0,1000000))
```

vzorák napísal Baška
(max. 15 b za riešenie)

4. Pyramídy

Spôsobov ako riešiť túto úlohu je mnoho. V tomto vzorovom riešení sa postupne pozrieme na niekoľko z nich, začínajúc od najmenej efektívnych.

Skúšame všetky možnosti

Ako prvé asi každému napadlo vyskúšať všetky možnosti. Postupne budeme skúšať výšku prvej pyramídy (túto výšku si označíme a) od 1 až po n, druhej pyramídy (túto výšku si označíme b) tiež od 1 po n. Pre každú dvojicu a, b si vieme vypočítať koľko kociek na tieto dve pyramídy potrebujeme:

$$\frac{a \cdot (a+1)}{2} + \frac{b \cdot (b+1)}{2}$$

Ak je tento súčet rovný n, tak sme našli riešenie a môžeme skončiť.

Takéto riešenie by malo časovú zložitosť $O(n^2)$, čo stačí na vyriešenie prvej sady[2]. Môžeme si však všimnúť, že skúšame veľa zbytočných výšok. Pyramída s výškou n bude mať určite viac ako n kociek (pre n väčšie ako 1). Pokúsme sa teda určiť hornú hranicu pre a a b tesnejšie.

Otázka je, aké najväčšie môže byť a, aby platilo, že počet kociek pyramídy výšky a je menší alebo rovný n. Matematicky zapísané:

$$\frac{a \cdot (a+1)}{2} \leq n$$

Jedna možnosť je vyjadriť si z toho hodnotu a, čo naozaj spravíme na konci nášho vzorového riešenia. Ak sme ale lenivý, alebo nevieme ako sa riešia kvadratické rovnice, môžeme si pomôcť počítačom. Vhod nám príde cyklus `while`, ktorý beží tak dlho, kým platí nejaká podmienka. V našom prípade teda horeuvedená nerovnica.

Listing programu (C++)

```
#include <iostream>

using namespace std;

int main() {
    long long n;
    cin>>n;
    bool nasla = false;
    int a = 1;
    while (a*(a+1)/2 < n) {
        int b = 1;
        while (b*(b+1)/2 < n) {
            if (a*(a+1)/2 + b*(b+1)/2 == n) {
                nasla = true;
                break;
            }
            b++;
        }
        if (nasla) break;
        a++;
    }
    if (nasla) cout<<"ANO"<<endl;
    else cout<<"NIE"<<endl;
    return 0;
}
```

Časová zložitosť nášho riešenia závisí od toho, koľkokrát sa zopakujú naše dva `while` cykly. Označme si najväčšie číslo, pre ktoré platí nerovnica

$$\frac{a \cdot (a+1)}{2} \leq n$$

premennou max_a. Časová zložitosť bude potom $O(max_a^2)$. My by sme to však chceli vyjadriť pomocou premennej n. Ako si ukážeme neskôr, hodnota max_a je zhruba $\sqrt{n}$ a preto je zložitosť nášho riešenia $O(n)$.

Predpočítanie veľkostí pyramíd

V predošlom riešení, sme si vždy znovu a znovu počítali veľkosť pyramídy s výškou a. Skúsme si teda tieto hodnoty vypočítať vopred.

```
    long long maxa = 1;
    while (maxa*(maxa+1)/2 < n) maxa++;
    vector<long long> pyramidy(maxa);
    for (long long i = 0; i<maxa; ++i) {
        pyramidy[i] = i*(i+1)/2;
    }
```

[2] Prípadne prvých dvoch, závisí od rýchlosti jazyka.

Využitím `while` cyklu, si vieme zistiť maximálnu hodnotu a – hodnota max_a. Následne si vytvoríme pole takejto veľkosti. Použijeme `vector`, ktorý sa správa takmer ako polia ktoré poznáte. Na začiatku mu ale vieme v zátvorke povedať, akú má mať veľkosť[3]. Následne sme do jeho jednotlivých políčok vypočítali príslušné hodnoty. V poli `pyramidy` teda máme všetky prípustné veľkosti pyramíd. Presnejšie, hodnota `pyramidy[i]` označuje, koľko kociek potrebujeme na postavenie pyramídy výšky i.

Rovnako ako v predchádzajúcom riešení by sme mohli vyskúšať všetky dvojice veľkostí pre hodnoty a a b a overiť, či sa `pyramidy[a]+pyramidy[b]` rovná n. To by však naše riešenie neurýchlilo. Ukážeme si preto dva rýchlejšie prístupy využívajúce toto pole.

Dvaja bežci

Vezmime si prípad, keď máme na vstupe číslo $n = 1194$, ktoré vieme dostať ako $66 + 1128$. Pre takéto n bude hodnota $max_a = 49$, takže posledná hodnota v poli `pyramidy` bude číslo 1225. Pozrime sa teraz na súčet najmenšieho a najväčšieho čísla, teda $1 + 1225$. Vidíme, že tento súčet je priveľký. Z toho ale vyplýva, že na číslo 1225 sa už nikdy nemusíme pozerať, pretože aj súčet s najmenším možným číslom je príliš veľký. Predposledné číslo v našom poli je číslo 1176. Ale $1 + 1176 < 1194$, teda tento súčet je primalý. To ale znamená, že vo výsledku sa nemôže nachádzať ani 1, pretože ani jej súčet s najväčším prípustným číslom nie je dosť veľký. Preto sa posunieme na ďalšie číslo v poradí, teda číslo 3, s ktorým spravíme to isté. Opäť zistíme, že $3 + 1176$ je príliš malé a 3 sa vo výsledku nachádzať nebude.

Zhrňme si to. Budeme mať ukazovatele do nášeho poľa – takzvaných bežcov. Prvý z nich bude začínať na najmenších číslach a bude sa celý čas posúvať v poli doprava (k väčším číslam), druhý začne na najväčších číslach a bude sa celý čas posúvať doľava (k menším číslam). Vždy, keď bude súčet veľkostí oboch bežcov menší ako n, posunieme prvého bežca o jedno doprava, aby sme súčet zväčšili, a keď bude súčet väčší ako n, posunieme druhého bežca o jedno doľava. Ak sa bude súčet čísel, na ktoré bežci ukazujú rovnať n, tak sa môžeme zastaviť, pretože sme našli riešenie.

Listing programu (C++)

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
    long long n;
    cin>>n;

    long long maxa = 1;
    while (maxa*(maxa+1)/2 < n) maxa++;
    vector<long long> pyramidy(maxa);
    for (long long i = 0; i<maxa; ++i) {
        pyramidy[i] = i*(i+1)/2;
    }

    int zlava = 1;
    int zprava = maxa-1;
    bool nasla = false;
    while(zlava <= zprava) {
        long long sucet = pyramidy[zlava] + pyramidy[zprava];
        if (sucet < n) {
            zlava++;
        } else if (sucet > n) {
            zprava--;
        } else {
            nasla = true;
            break;
        }
    }

    if (nasla) cout<<"ANO"<<endl;
    else cout<<"NIE"<<endl;
    return 0;
}
```

Ostáva nám odhadnúť časovú zložitosť. V riešení máme jeden `while` cyklus, ktorý vždy buď zväčší premennú `zlava`, zmenší premennú `zprava`, alebo ukončí program. Vezmime si prípad, že odpoveď je „`NIE`", takže `while` cyklus skončí na podmienke `zlava <= zprava`. V konečnom dôsledku sme premennú `zlava` zväčšili k krát, pre nejaké (možno nulové) k. Rovnako sme premennú `zprava` zmenšili l krát pre nejaké (možno nulové) l. Platí teda, že $1 + k = (max_a - 1) - l$, takže $k + l = max_a - 2$. Takže náš `while` cyklus sa vykoná najviac $max_a - 2$ krát, z čoho vyplynie časová zložitosť $O(max_a)$, teda $O(\sqrt{n})$.

Binárne vyhľadávanie

Teraz sa na to pozrime inak. Máme na vstupe číslo 1194 a budeme postupne prechádzať poľom a každé z

[3] A okrem toho má oproti klasickým poliam ešte zopár výhod, ktoré ale v tomto vzorovom riešení nebudeme potrebovať.

čísel skúsime ako veľkosť prvej pyramídy. Napríklad, máme číslo 66. Vieme teda, že veľkosť druhej pyramídy musí byť $1194 - 66 = 1128$. Ak to naozaj je veľkosť nejakej pyramídy, tak sa musí nachádzať v poli `pyramidy`. Ako zistíme či sa číslo 1128 nachádza v poli `pyramidy`?

Určite poznáte hru "Mysli si číslo!". Ja si myslím číslo do 1024, a vy tipujete aké číslo to je. Vždy keď si tipnete číslo, tak ja vám poviem "viac", "menej" alebo "správne". Najlepšou stratégiou je povedať ako prvé číslo 512. Podľa odpovede totiž viete zistiť, v ktorej polovici čísel sa nachádza hľadané číslo. Buď poviem "viac" a moje číslo je potom v intervale $513 - 1\,024$, alebo poviem "menej" a ostane vám interval $1 - 511$ (alebo ste ho trafili). V oboch prípadoch sa vám však interval možností zmenší na polovicu. No a na intervale polovičnej veľkosti viete spraviť to isté. Znova tipnete číslo uprostred tohto inetrvalu, a podľa odpovede zmenšíte jeho veľkosť na polovicu.

Rovnaký prístup môžeme použiť, ak hľadáme číslo v usporiadanom poli. Ako konce intervalu si budeme pamätať indexy do poľa. Na začiatku bude ľavý koniec `zac` na prvom prvku poľa, a pravý `kon` za koncom poľa (ukazuje na index poľa kde už nie je žiaden prvok). Stred poľa určíme ako `stred = (kon+zac)/2`. Pozrieme sa na stredný prvok, a ak je menší ako číslo ktoré hľadáme, tak posunieme `zac` na `stred`. Ak je väčší, tak posunieme `kon` na `stred`. Skončíme, v momente, keď platí `kon = zac + 1`. Potom `zac` ukazuje na hľadaný prvok (ak sa tam nachádza), a `kon` ukazuje hneď na prvok vpravo od neho.

```
int zac = 1, kon = maxa;
while (kon-zac > 1) {
    int stred = (kon+zac)/2;
    if (pyramidy[stred] <= hladame) zac = stred;
    else kon = stred;
}
```

Tento algoritmus sa volá binárne vyhľadávanie. Jeho časová zložitosť na k prvkoch je $O(\log k)$.

Keď už teda vieme v poli binárne vyhľadávať, tak si môžme jednoducho prejsť celé pole a k číslu `pyramidy[i]` skúsiť v poli nájsť číslo `n-pyramidy[i]`.

Listing programu (C++)

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
    long long n;
    cin>>n;

    long long maxa = 1;
    while (maxa*(maxa+1)/2 < n) maxa++;
    vector<long long> pyramidy(maxa);
    for (long long i = 0; i<maxa; ++i) {
        pyramidy[i] = i*(i+1)/2;
    }

    bool nasla = false;
    for (int i = 1; i<maxa; ++i) {
        long long hladame = n - pyramidy[i];
        int zac = 1, kon = maxa;
        while (kon-zac > 1) {
            int stred = (kon+zac)/2;
            if (pyramidy[stred] <= hladame) zac = stred;
            else kon = stred;
        }
        if(pyramidy[zac] == hladame) {
            nasla = true;
            break;
        }
    }

    if (nasla) cout<<"ANO"<<endl;
    else cout<<"NIE"<<endl;
    return 0;
}
```

Predpočítanie hodnôt nám trvá $O(max_a)$, a potom máme jeden `for`-cyklus, počas ktorého robíme binárne vyhľadávanie. Samotný `for`-cyklus sa vykoná max_a krát, takže aj binárne vyhľadávanie spravíme max_a krát. Časová zložitosť bude teda $O(max_a \cdot \log(max_a))$, čo je teda $O(\sqrt{n} \cdot \log \sqrt{n})$.

Dopočítanie hodnoty b

Využijeme niečo z predošlého prístupu. Na vstupe máme 1194 a veľkosť prvej pyramídy 66, a chceme zistiť či $1128 = 1194 - 66$ je tiež platná veľkosť pyramídy. Nedá sa to zistiť nejak jednoduchšie ako binárnym vyhľadávaním? No na to, aby toto číslo bolo veľkosťou nejakej pyramídy musí existovať celé číslo b, pre ktoré platí

$$1128 = \frac{b \cdot (b+1)}{2}$$

Takže stačí si vyjadriť b. Číslo 1128 si označíme ako zv (veľkosť druhej pyramídy).

$$zv = \frac{b \cdot (b+1)}{2}$$

$$2zv = b \cdot (b+1)$$

$$0 = b^2 + b - 2zv$$

Posledný výraz sa volá kvadratická rovnica. Táto rovnica môže mať až dva rôzne riešenia, ktoré si vieme vyjadriť vzrocom:

$$D = \sqrt{1 - 4(-2zv)} = \sqrt{1 + 8zv}$$

$$b_{1,2} = \frac{-1 \pm D}{2}$$

Našou úlohou je teraz nájsť kladné celé číslo b, ktoré spĺňa túto rovnicu. Ak sa nám ho podarí nájsť, tak vieme, že zo zv kociek vieme poskladať pyramídu. Preto musí byť odmocnina z $1 + 8zv$ celé číslo. Ak je, tak môžeme D dosadiť do druhého vzorca. Z tohto vzorca ľahko vidno, že v našom prípade bude jedno z možných riešení vždy záporné. Toto riešenie nás nezaujíma. Preto stačí D dosadiť iba do $b = \frac{D-1}{2}$. Po dosadení skontrolujeme, že sme dostali kladné číslo. Okrem toho aj skotrolujeme, či je $D - 1$ párne číslo[4]. Ak všetko platí, tak sme našli naše riešenie :D

Listing programu (C++)

```
#include <iostream>
#include <cmath>

using namespace std;

int main() {
    long long n;
    cin>>n;

    bool nasla = false;
    long long a = 1;
    while (a*(a+1)/2 < n) {
        long long c = a*(a+1) - 2*n;
        long long D2 = 1 - 4*c;
        if (D2 >= 0) {
            long long D = round(sqrt(D2));
            if (D*D == D2) {
                if ((D-1)%2 == 0) {
                    long long b1 = (-1 + D) / 2;
                    if (b1 > 0) {
                        nasla = true;
                        break;
                    }
                }
            }
        }
        a++;
    }
    if (nasla) cout<<"ANO"<<endl;
    else cout<<"NIE"<<endl;
    return 0;
}
```

V riešení používame iba jeden `while` cyklus, ktorý, ako už vieme, bude trvať $O(max_a)$, teda $O(\sqrt{n})$.

Prečo je $O(max_a) = O(\sqrt{n})$?

Vieme že pre max_a musí platiť rovnosť:

$$\frac{max_a \cdot (max_a + 1)}{2} = n$$

$$max_a \cdot (max_a + 1) = 2n$$

Ak by sme zanedbali tú plus jednotku, hodnotu max_a by sme odhadli ako odmocnicu z $2n$. Čo pri použití O-notácie znamená, že: $O(max_a) = O(\sqrt{2n}) = O(\sqrt{n})$.

Samozrejme, hodnotu max_a vieme vyjadriť aj exaktnejšie. Máme kvadratickú rovnicu $max_a^2 + max_a - 2n = 0$ a jej kladné riešenie je

$$max_a = \frac{-1 + \sqrt{1 + 8n}}{2}$$

[4] To v skutočnosti netreba, pretože odmocnina z $1 + 8zv$ nemôže byť párna, takže $D - 1$ bude vdžy párne.

Pri O-notácii, ktorú používame na zapisovanie časovej zložitosti však aj tak väčšinu konštánt zanedbávame a preto prídeme k rovnakému výsledku, že $O(max_a) = O(\sqrt{n})$.

5. Poriadna výzva

vzorák napísal Andrej
(max. 15 b za riešenie)

Vzorové riešenie tohto príkladu je písané spôsobom, ktorý ho umožňuje čítať postupne, po sadách. Ak vás teda úloha bavila a stále by ste ju chceli vyriešiť, ale neviete si s ňou dať rady, odporúčame prečítať vzorové riešenie iba po sadu, na ktorej ste sa zasekli. Odtiaľ potom zapojiť mozgové závity a skúsiť úlohu doriešiť bez ďalšej pomoci. Ak už chcete mať tento príklad za sebou alebo sa tešíte na úplny vzorák, môžete ho samozrejme prečítať celý na jeden raz.

A teraz sa už pozrime na myšlienku riešenia. Riešenie je vcelku priamočiare, vyhľadaniu slov v osemsmerovke sa nemáme ako vyhnúť. Neostáva nám preto nič iné ako zvoliť vhodnú implementáciu. Samozrejme, zložitosť vyhľádavania, ktorú bude riešenie vyžadovať sa bude v jednotlivých sadách líšiť.

Predtým, než skočíme na samotné riešenie súťažnej úlohy a implementáciu, bolo by vhodné si ukázať ako v tejto úlohe (a v typovo podobných), šikovne a jednoducho načítať a uložiť dáta zo vstupu.

string

V `C++` existuje na uloženie reťazca znakov dátova štruktúra s názvom `string`. Na `string` sa v skutočnosti môžeme pozrieť iba ako na pole `char`ov. Vieme sa v ňom, rovnako ako v poli, pozrieť na jednotlivé indexy a zistiť, aký znak sa na nich nachádza.

Samozrejme, ak by bol `string` iba pole `char`ov (`char[]`), nebol by ničím zaujímavý. Prináša nám teda isté výhody. Do premennej tohto typu môžeme načítavať rovnako ľahko ako do premennej typu `int`, `char`, či `double`. Napríklad funkcia `cin` pri načítaní do `stringu` číta znaky až pokým nenarazí na znak konca riadku, či na medzeru, takže funguje rovnako ako pri načítavaní čísel.

Tomuto typu dokonca nie je ani nutné povedať, ako veľký bude reťazec, ktorý do neho chceme vložiť. Pri načítaní zo vstupu sa totiž sám zväčší na potrebnú veľkosť.

Ak si nie sme istí alebo nevieme, akú veľkosť má naš `string`, teda koľko znakov v ňom je, tak môžme použiť jeho vlastnosť `.size()`. Tá nám vráti jeho veľkosť, teda číslo určujúce, koľko znakov sa v ňom nachádza.

Posledné vylepšenie je, že si vieme jednoducho vytvoriť pole takýchto stringov, teda v podstate dvojrozmerné pole charov a pristupovať k jednotlivým políčkam pomocou indexov rovnako, ako sme zvyknutí napríklad pri dvojrozmernom poli intov.

Nasledujúci príklad demonštruje, ako sa so stringom pracuje.

Listing programu (C++)

```
#include<iostream>
#include<vector>
#include<string>
//string sa nachadza v rovnomennej kniznici
using namespace std;
int main()
{
    //inicializacia stringu
    string ex1 = "text";
    string ex2;//prazdny string
    string ex3 = "ohodnotenie";
    cout << ex1 << endl;//vypisanie
    if(ex2.size() == 0)cout<<"Prazdny string."<<endl;
    cin >> ex2;//nacitanie

    //X.size() vracia dlzku stringu X
    cout << "String ktory si zadal ma dlzku " << ex2.size() << endl;

    //pristupovanie mozne na indexy v rozsahu 0 az X.size()-1
    for(int i = 3; i<ex3.size(); ++i)ex3[i] = 'a';

    //vypis po znakoch tiez nie je problematicky
    for(int i = 0; i<ex3.size(); ++i)cout<<ex3[i];
    cout<<endl;

    //vytvorenie vectoru stringov ? ziadny problem
    vector<string>dvojrozmerne(4);//vytvori vector velkosti 4

    dvojrozmerne[0] = "banan";
    dvojrozmerne[3] = "prask";

    cout << dvojrozmerne[0][3] << endl;//vypise konkretny znak, teda 'a'

    dvojrozmerne[3][4] = 'x';//upravime konkretny znak
    cout << dvojrozmerne[3] << endl;//vypise string prasx
    return 0;
}
```

Ak ste sa s týmto dátovým typom ešte nestretli, odporúčame vám sa s ním v tejto fáze pohrať a uistiť sa, že poznáte jeho základnú funkcionalitu. Keď už máme predstavu o tom, ako funguje `string` v `C++`, môžme sa postupne pustiť do jednotlivých sád.

Sada 1

Pre prvú sadu je hľadanie vcelku jednoduché, keďže osemsmerovka je tvorená iba jedným riadkom. To znamená, že sa v nej slovo môže nachádzať iba v dvoch smeroch.

Stačí preto nájsť dané slovo v nejakom texte, našej osemsmerovke. Text tvorí v tomto prípade iba jeden riadok, takže našou úlohou je napísať funkciu, ktorá nájde v jednom stringu nejaký iný. Tento problém sa zdá byť jednoduchý, ale v informatike je veľmi populárny a existuje naň množstvo rôznych algoritmov s odlišnými časovými zložitosťami.

My si však vystačíme s najjednoduchším postupom, ktorý nám napadne: Pre každú pozíciu skúsime, či sa na nej nemôže v texte začínať naše slovo. Ak sa prvé písmená zhodujú, pozrieme na ďalšie písmeno v texte a porovnáme ho s ďalším písmenom v našom hľadanom slove. Ak sa zhodujú, pokračujeme, pokým neprídeme na posledné písmeno. Ak sa na nejakom indexe nezhodujú, skončíme a začneme skúšať od ďalšieho začiatku.

Nasledovný program demonštruje kód funkcie, ktorá dostane na vstupe dva stringy a vypíše indexy prvého stringu, na ktorých začína výskyt druhého stringu. V programátorskom žargóne sa často prvý string nazýva text a druhý pattern alebo text a vzorka.

Listing programu (C++)

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

void nachadza_sa(string& text, string& vzorka)
{
    for(int i=0;i<text.size();++i)
    {//pre každý index v texte

        if(i + vzorka.size() > text.size())return;
        //ak je dĺžka textu 10 a sme na 8 indexe, pattern
        //dlzky 4 uz urcite nenajdeme, lebo by koncil mimo textu

        for(int j=0;j < vzorka.size();++j)
        {
            if(text[i + j] != vzorka[j])break;
            //porovnovame pokym sa nelisi text od vzorky, vtedy skoncim
            if(j == vzorka.size()-1)cout<<"Vzorka najdena od "<<i<<endl;
            //ak som uz porovnal posledny prvok a neskoncil, tak mam zhodu
            //na indexoch i az i + vzorka.size() - 1
        }
    }
}

int main()
{
    string text, vzorka;
    cin >> text >> vzorka;
    nachadza_sa(text, vzorka);
    return 0;
}
```

Toto isté nám už len stačí naimplementovať pre našu osemsmerovku. Rozdielom bude len to, že ak dané slovo niekde nájdeme, tak si políčka, na ktorých sa nachádzalo označíme za vyškrtnuté. Toto škrtanie spravíme v inom, pomocnom poli.

Musíme si ešte uvedomiť, že slovo môže ležať aj v opačnom smere ako zľava doprava. Ak sa slovo nachádza v osemsmerovke sprava doľava, použijeme malý trik. Predstavíme si naše slovo otočené naopak (prečítame ho odzadu). Uvedomme si, že v tomto prípade sa bude nachádzať v smere zľava doprava. A tak okrem hľadania každého slova budeme v osemsmerovke hľadať aj jeho otočenie (reverz). Ako príklad uvedieme hľadanie slova `ba` v texte `kavab`. Samotné slovo sa síce v texte nenachádza v smere zľava doprava, ale v opačnom smere áno. Keď si hladané slovo otočíme, dostaneme `ab`. Ten už v texte úspešne nájdeme. Ako domácu úlohu si rozmyslite, že tento postup bude fungovať v každom prípade.

Listing programu (C++)

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>//potrebne z dovodu reverse()

using namespace std;

int main()
{
    int r, s, n;
```

```
    string osemsmerovka, slovo, rev_slovo;
    cin >> r >> s >> n;
    cin >> osemsmerovka >> slovo;
    //načítame vstup

    rev_slovo = slovo;//skopirujeme slovo do pomocnej premennej
    reverse(rev_slovo.begin(), rev_slovo.end());
    //tato funkcia nam v linearnom case obrati string
    //v rev_slovo sa teraz nachadza povodne slovo obratene
    vector<bool>vyskrtnute(s, false);
    //na zaciatku nie je nic vyskrtnute
    bool najdene = false;
    int dlzka_slova;
    for(int i=0;i < s;++i)
    {
        dlzka_slova = slovo.size();

        for(int j=0;j < dlzka_slova;++j)
        {
            if(osemsmerovka[i+j] != slovo[j])break;
            if(j == dlzka_slova-1 )najdene = true;
        }

        for(int j=0;j < dlzka_slova;++j)
        {
            if(osemsmerovka[i+j] != rev_slovo[j])break;
            if(j == dlzka_slova-1 )najdene = true;
        }

        if(najdene)//ak sme nasli reverse alebo povodne slovo, zaznacime si
        {
            for(int j=0;j < dlzka_slova;++j)vyskrtnute[i+j] = true;
            break;
        }
    }

    for(int i=0;i < s;++i)
    {
        if(vyskrtnute[i] == false)
        {
            cout<<osemsmerovka[i];
            //ak nie je vyskrtnute... vypiseme
        }
    }
    cout<<endl;

    return 0;
}
```

Sada 2

Keď sme už dokázali napísať program riešiaci prvú sadu, stačí spraviť zopár zmien a budeme vedieť riešiť aj druhú sadu. Naše prvé riešenie prepíšeme na funkciu `najdi()`, ktorá nájde v texte zadané slovo a ak sa tam nachádza, vyškrtne ho podobne ako pri riešení prvej sady. Potom túto funkciu už len $2n$-krát zavoláme na nájdenie n slov a ich n reverzov.

Listing programu (C++)

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>

using namespace std;

vector<bool>vyskrtnute;

void najdi(string& osemsmerovka, string& slovo)
{
    bool najdene = false;
    int s = osemsmerovka.size(), dlzka_slova;
    for(int i=0;i < s;++i)
    {
        dlzka_slova = slovo.size();

        for(int j=0;j < dlzka_slova;++j)
        {
            if(osemsmerovka[i+j] != slovo[j])break;
            if(j == dlzka_slova-1 )najdene = true;
        }

        if(najdene)
        {
            for(int j=0;j < dlzka_slova;++j)vyskrtnute[i+j] = true;
            return;
        }
    }
}

int main()
{
    int r, s, n;
    string osemsmerovka, slovo, rev_slovo;
```

```
    cin >> r >> s >> n;
    cin >> osemsmerovka;

    vyskrtnute.resize(s, false);

    for(int i=0;i<n;++i)
    {
        cin >> slovo;
        rev_slovo = slovo;
        reverse(rev_slovo.begin(), rev_slovo.end());

        najdi(osemsmerovka, slovo);
        najdi(osemsmerovka, rev_slovo);
    }

    for(int i=0; i < s; ++i)
    {
        if(vyskrtnute[i] == false)
        {
            cout<<osemsmerovka[i];
        }
    }
    cout<<endl;

    return 0;
}
```

Sada 3, 4 a 5

Čo ak chceme ísť ďalej a vyriešiť osemsmerovku s viac riadkami? To čo si musíme uvedomiť je, že v tomto prípade sa môže hľadané slovo nachádzať vo všetkých ôsmych smeroch, s čím sme v prvých dvoch sadách nemuseli počítať.

Predstavme si, že chceme zistiť, či sa hľadané slovo začína na pozícii (i, j). Najskôr si musíme zvoliť smer, v ktorom sa toto slovo bude v osemsmerovke nachádzať. Potom však musíme vedieť porovnať, či sa k-ty znak nášho slova rovná písmenku, ktoré je od pozícii (i, j) v danom smere vzdialené na k krokov. Ako to však naimplementovať?

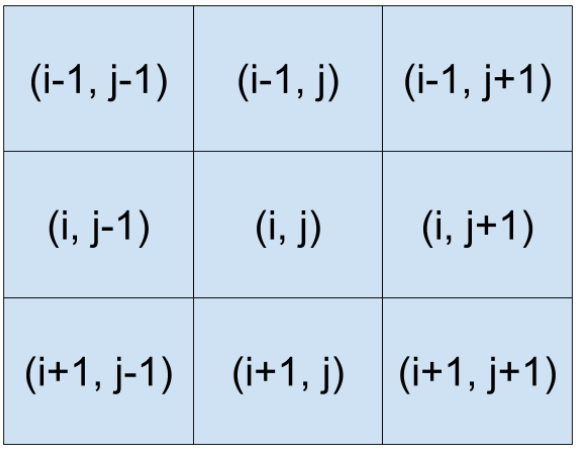
Pokúsme sa napísať jednoduchú funkciu, ktorej zadáme ako parametre súradnice začiatočného políčka, počet krokov a jeden zo smerov, v ktorom sa chceme pohnúť. Funkcia nám potom vypíše index políčka, na ktorom skončíme.

Na začiatok si jednotlivé smery očíslujeme. Smer nahor má číslo 0. Ostatné smery si očíslujeme v smere hodinových ručičiek, teda napr. smer doprava bude mať číslo 2, smer dole číslo 4 atď.

7	0	1
6	(i, j)	2
5	4	3

Ak si predstavíme, že sme začínali na pozícií (i, j) a chceme vedieť nultý prvok (indexujeme od 0) v nejakom smere, bude to vždy prvok (i, j). Ak chceme vedieť prvý prvok v zadanom smere, vieme to zistiť vcelku jednoducho. Nech sme sa pohli na akékoľvek políčko vo vzdialenosti 1 od nášho pôvodného políčka (i, j). Vieme, že jednotlivé indexy sa nám zmenia maximálne o 1. V prípade, že sme sa pohli jedným zo smerov nahor (0, 1, 7), tak od prvej súradnice odčítame 1 (inými slovami pričítame -1). Pretože index riadka sa zmenšil práve o jedna. Ak sme sa pohli jedným zo smerov (2, 6), je nám jasné, že prvá súradnica, teda index riadka, sa nijak nezmení – pričítame 0. A nakoniec, ak sme sa pohli do jedného zo smerov (3, 4, 5), tak k prvej súradnici 1 pričítame.

Rovnakú úvahu vieme urobiť aj pre druhú súradnicu kde pričítavame 1 ak sa hýbeme do smerov (1, 2, 3), nepričítavame nič pri smeroch (0, 4) a odčítavame 1 pri smeroch (5, 6, 7). Takže ak sa chceme z políčka (i, j) pohnúť smerom 5, tak vieme, že sa dostaneme na políčko $(i + 1, j - 1)$.



Vytvorme si preto 2 pomocné polia `zmena_x[]` a `zmena_y[]` indexované smerom. Na l-tej pozícii poľa `zmena_x[]` bude dĺžka, o ktorú sa máme posunúť na riadku pri smere l. Analogicky, pole `zmena_y[]` bude obsahovať na l-tej pozícií dĺžku, o ktorú sa musíme posunúť v stĺpci pri smere l. Ak sme teda na pozícii (i, j) a chceme sa pohnúť o jeden krok v smere $smer$, indexy novej pozície budú $(i+zmena\_x[smer], j+zmena\_y[smer])$.

Všeobecnejšie, ak nás zaujíma kde sa nachádzame po k takýchto krokoch, stačí tieto dve čísla pričítať k-krát. Budeme sa teda nachádzať na políčku s indexom $(i + k \cdot zmena\_x[smer], j + k \cdot zmena\_y[smer])$.

Nasleduje implementácia požadovanej funkcie. Môžete si v nej pozrieť, čo presne obsahujú polia `zmena_x[]` a `zmena_y[]` a porovnať ich s predchádzajúcim obrázkom.

Listing programu (C++)

```
#include<iostream>
#include<vector>
#include<string>

using namespace std;

int zmena_x[]={-1, -1, 0, 1, 1, 1, 0, -1};
int zmena_y[]={0, 1, 1, 1, 0, -1, -1, -1};

void kde_skoncim(int i, int j, int pocet_krokov, int smer)
{
    int nove_i, nove_j;
    nove_i = i + pocet_krokov * zmena_x[smer];
    nove_j = j + pocet_krokov * zmena_y[smer];
    cout<< "Suradnice su: " << nove_i << ";" << nove_j <<endl;
}

int main()
{
    int i, j, pocet_krokov, smer;
    cin >> i >> j >> pocet_krokov >> smer;
    kde_skoncim(i, j, pocet_krokov, smer);
    return 0;
}
```

V tomto momente máme už všetko, čo potrebujeme na to, aby sme napísali novú verziu našej funkcie `najdi()`, ktorá si poradí aj so zložitejšími osemsmerovkami.

V tejto funkcii si pre všetky pozície v osemsmerovke a všetkých osem smerov overíme, či sa v danom smere nenachádza hľadané slovo. Ak áno, vyškrtáme ho a prejdeme na ďalšie slovo. Takto už vieme riešiť všetky ostatné sady a dostávame aj vzorové riešenie. Maličkosťou je, že dobrým zvykom je polia `zmena_x[]` a `zmena_y[]` označovať `dx[]`, resp. `dy[]`. Vyhnete sa tým zbytočnému písaniu a budete používať rovnakú konvenciu.

Listing programu (C++)

```
#include<iostream>
#include<vector>
#include<string>

using namespace std;

int dx[] = {-1, -1, 0, 1, 1, 1, 0, -1};
int dy[] = {0, 1, 1, 1, 0, -1, -1, -1};

vector<vector<bool> >vyskrtnute;
//tu si pamatam co este nemam vyskrtnute

void najdi(vector<string>&osemsmerovka, string &hladane)//funkcia najdi vyskrtne slovo "hladane"
{
```

```
    int r = osemsmerovka.size(), s = osemsmerovka[0].size(), akt_x, akt_y;
    //r, s su rozmery osemsmerovky

    for(int i=0;i<r;++i)for(int j=0;j<s;++j)//pre vsetky indexy
    {
                for(int k=0;k<8;++k)//pre vsetky smery
                {
                    for(int d=0;d<hladane.size();++d)//prechadzam slovo kym mam zhodu
                    {
                        //vypocitam si, kde momentalne som
                        akt_x=i+d*dx[k];
                        akt_y=j+d*dy[k];

                        if(akt_x >= 0 && akt_y >= 0 && akt_x<r && akt_y < s && osemsmerovka[akt_x][akt_y] == hladane[d]);
                        else break;//ak som von zo svojho pola alebo nemam zhodu na d-tom indexe v slove tak koncim
                        if(d == hladane.size()-1)//ak som dosiel az na koniec slova, tak ho uz len vyskrtnem a skoncim
                        {
                            for(int dd=0;dd<hladane.size();++dd)
                            {
                                akt_x=i+dd*dx[k];
                                akt_y=j+dd*dy[k];
                                vyskrtnute[akt_x][akt_y]=true;
                            }
                            return;
                        }

                    }
                }
    }
}

int main()
{
    ios_base::sync_with_stdio(false);
    string hladane;
    int r, s, n;
    cin >> r >> s >> n;

    vector<string>osem(r);
    vyskrtnute.resize(r, vector<bool>(s, false));

    for(int i=0;i<r;++i) cin >> osem[i];

    for(int f=0;f<n;++f)
    {
        cin >> hladane;
        najdi(osem, hladane);
    }

    for(int i=0;i<r;++i)for(int j=0;j<s;++j)
    {
        if(!vyskrtnute[i][j])cout<<osem[i][j];
    }
    cout<<endl;
    return 0;
}
```

Čo sa týka časovej a pamäťovej zložitosti, pri hľadaní slova sa nám v najhoršom prípade môže stať, že v každom smere sa písmená na mriežke líšia od nášho slova až v poslednom znaku. Pri tomto procese spravíme $8 \cdot |slovo|$ porovnaní, kde $|slovo|$ je počet znakov slova. Slovo v najhoršom prípade budeme vyhľadávať z každej pozície mriežky, teda z $r \cdot s$ pozícií. Pre jedno slovo teda dostávame približne $r \cdot s \cdot 8 \cdot |slovo|$ krokov.

Pretože pri O-notácii zanedbávame konštanty, môžeme beztrestne vyškrtnúť 8 a dostaneme časovú zložitosť vyhľadávania jedného slova $O(r \cdot s \cdot |slovo|)$. Slov je ale na vstupe n a spravíme preto rádovo $r \cdot s \cdot |slovo_1| + r \cdot s \cdot |slovo_2| + ... + r \cdot s \cdot |slovo_n|$ krokov. Všimnime si, že $r \cdots$ môžeme z výrazu vynať a dostaneme tak $r \cdot s \cdot (|slovo_1| + |slovo_2| + ... + |slovo_n|)$. Súčet $|slovo_1| + |slovo_2| + ... + |slovo_n|$ je rovný celkovému súčtu slov. Celkovo preto dostávame časovú zložitosť $O(r \cdot s \cdot sucet\_dlzok\_slov)$.

Pamäťová zložitosť je $O(r \cdot s + sucet\_dlzok\_slov)$, kedže si musíme pamätať údaje pre mriežku a slová na vstupe.