PRASK
V. ročník, 2018/19
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

Vzorové riešenia 2. kola zimnej časti

vzorák napísal(a) Jitka+Emo
(max. 15 b za riešenie)

1. Podivný zošit

Pokračovanie úlohy z prvého kola s naoko rovnakými úlohami, ktoré však skrývali iné pravidlá. Postup na ich odhalenie sa však riadi podobnými trikmi. Skúšať malé čísla a zapájať fantáziu.

Level 1

Tabuľka má vypísať písmeno G. Pri písmenách si spomenieme na naše začiatky v prvej triede a na abecedu. Riešenie je číslo 6, lebo G je šiete písmeno abecedy.

Level 2

Máme vypísať násobok geparda. Čo to ale znamená? Ktorou vlastnosťou je známy gepard? Jeho rýchlosťou! Rýchlosť geparda nájdeme na internete a vynásobíme požadovanou konštantou, teda 531.54. Dostaneme číslo 58 469, čo je správny výsledok.

Level 3

Môžeme si všimnúť, že výstup má vždy rovnaký počet písmen ako zadané číslo cifier. Každé písmeno teda podáva informáciu o jednej cifre. Konkrétne, udáva prvé písmeno cifry. Niektoré cifry sa začínajú na to isté písmeno, napríklad `dva` a `deväť`, tabuľka ich však berie ako tú istú cifru a my si môžeme vybrať, ktorú z nich použijeme. Vypísať máme `JNJNDŠ`, čo si môžeme napríklad reprezentovať ako `jedna, nula, jedna, nula, dva, šesť`. Zodpovedajúci vstup je teda číslo 101 026.

Level 4

Postupným skúšaním čísel, si možeme všimnúť, že **prvočísla** nám vždy dajú hodnotu 2. Už si len stačí uvedomiť, že prvočísla sú čísla, ktoré majú práve 2 deliteľov. Stačí teda nájsť číslo, ktoré má 20 deliteľov, napríklad 432.

Level 5

Znalci slovenskej realistickej literatúri po prvých slovách odhalia, že v tejto tabľke sa nachádza celé dielo `Ťapákovci` od `B.S. Timravy`. Nám bežným smrteľníkom to trvá trochu dlhšie, ale tiež na to s pomocou Googlu časom prídeme. Zistíme, koľké je slovo `Rozpaprčená` v tomto diele, čím dostaneme výsledok 3 752.

Level 6

Každý si zažil, alebo zažije, nejakú astrologickú prednášku, ktorej interaktívny prvok je prepočítávanie vlastnej hmotnosti na **Zemi** na hmotnosť na rôznych iných planétach Slnečnej sústavy. Hmotnosť si môžeme reprezentovať ako gravitačnú silu pôsobiacu na teleso. Štandart pre zisťovanie gravitačnej sily (zrýchlenia) na iných planétach je gravitácia na **Zemi**, takzvané g. Ostatné gravitačné sily si vieme vyjadriť ako násobok g. Aby sme vyriešili úlohu, musíme najprv zistiť, na ktorej planéte sa nachádzame. Na to nám poslúži prvá cifra, ktorá udáva poradie planéty od Slnka. Tabuľka od nás chce hmotnosť 8913.28 na planéte **Merkúr**. Gravitačná silu **Merkúru** je $0.38g$. Na získanie správneho vstupu musíme požadovanú hmotnosť vydeliť nájdenou konštantou, čím získame 23 456. Na začiatok ešte pripojíme 1 na reprezentáciu **Merkúru** a máme odpoveď.

Level 7

Z formátu výstupu vidíme, že ide o nejaký zápis dátumu a času. Ak si stanovíme začiatočný bod času, každý dátum vieme reprezentovať počtom sekúnd, ktoré od tohto momentu prešli. Práve tento prístup využíva systém na určovanie času `Unix time`. V skratke, ide o systém, ktorý pre zadaný dátum a čas povie, koľko sekúnd prešlo od polnoci 1.1.1970. Na tento prepočet existujú rôzne kalkulačky, ale dá sa to samozrejme aj vypočitať. Počet uplynutých sekúnd od polnoci 1.1.1970 po 28.12.2018 05 : 10 : 15 je 1 545 973 815.

Ak vás zaujíma ako `Unix time` presne funguje a chcete si riešenie sami vypočítať, možete si to naštudovať napríklad na wikipédii.

Level 8

Vo výstupe si všimneme meno `Lothar Collatz`. Známy `Collatzov problém` je dodnes nedokázaný problém hovoriaci o postupnosti čísel, ktorá je definovaná tak, že začína kladným celým číslom n. Nasledujúci člen sa z predchádzajúceho vypočíta nasledovne:

- Ak je predchádzajúci člen párny, vydelí sa 2.
- Ak je predchádzajúci člen nepárny, vynásobí sa 3 a pripočíta sa 1.

Predpokladá sa, že bez ohľadu na to, akú hodnotu n si zvolíme na začiatku, postupnosť časom vždy dosiahne hodnotu 1.

Našou úlohou je nájsť číslo n, pre ktoré má postupnosť 20 členov. Tekéto číslo je napríklad 27.

Level 9

Po vyskúšaní zopár čísel zistíme, že vypísané číslo je vždy menšie ako to zadané. Taktiež platí, že pre **prvočíslo** p dostaneme vždy hodnotu $p - 1$. Uvedomme si, že **prvočísla** sú delitaľné iba číslom 1 a sebou samým. Takže hodnota $p - 1$ udáva počet čísel nesúdelitaľných s **prvočíslom** p od neho menších.

(*Dve prirodzené čísla n, k sú nesúdeliteľné ak platí, že $NSD(n, k) = 1$. Preto je číslo 1 nesúdeliteľné s každým číslom.*)

Tabuľka teda vracia počet menších čísel, ktoré sú nesúdeliteľné so zadanou hodnotou. My hľadáme číslo so 432 nesúdeliteľmi menšími prirodzenými číslami, napríklad 1890.

Funkcia na výpočet počtu prirodzených čísel menších ako zadané číslo n s ním nesúdeliteľných sa nazýva Eulerova funkcia a označuje sa gréckym písmenom $\phi(n)$.

Level 10

Po chvíli skúšania si uvedomíme, že dostávame postupne písmenká v abecede. Keď však zadáme zlomové číslo 26, dostaneme `BA`. Keď poskúšame ďalej, tak si uvedomíme, že to funguje podobne ako s číslami. Jednolivé písmenká si môžeme predstaviť ako cifry $A = 0, B = 1, \cdots, Z = 25$. Čislo 26 sa potom správa podobne ako v desiatkovej sústave číslo 10. Správne číslo sme mohli nájsť postupným zlepšovaním odhadu, alebo sme si ho presne vypočítali (skúste vymyslieť ako :)).

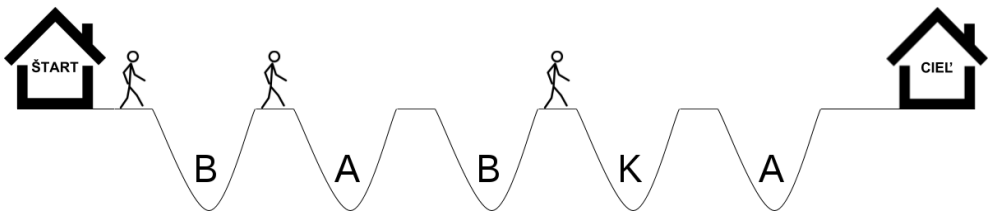
2. Reťazce v Lemingove

vzorák napísal(a) Žaba
(max. 15 b za riešenie)

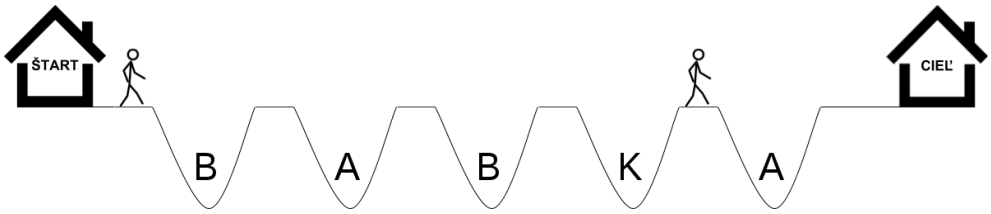
Podúloha a.

V prvej podúlohe si stačilo poriadne prečítať, ako sa lemingovia hýbu, a následne tento pohyb odsimulovať na zadanom slove `BBABKAABABABKA`. *(Vo webovej verzii tohto vzoráku sa nachádza animované gifko, na ktorom si môžete pozrieť celú simuláciu. Odporúčame sa na to pozrieť.)*

Po prečítaní `BBAB` sú lemingovia usporiadaní takto:



a po prečítaní `BBABKAABABABK` nasledovne:

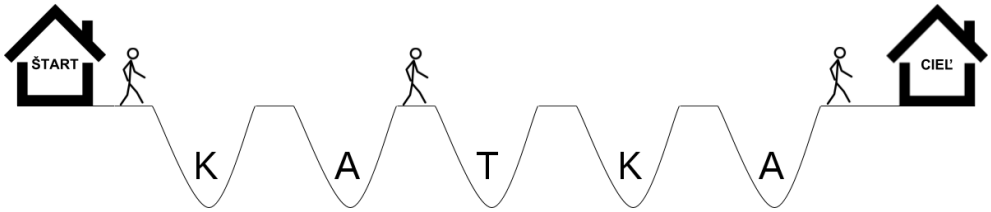


Otázka v tejto podúlohe okrem toho bola, koľkokrát sa v texte `BBABKAABABABKA` nachádza slovo `BABKA` a ako to súvisí s pohybom lemingov. Ľahko zistíme, že v tomto slove sa slovo `BABKA` nachádza na dvoch miestach. Keď si navyše odsimulujeme pohyb lemingov, všimneme si, že vždy keď toto slovo končí, jeden leming príde do cieľového domčeka. Ak sme totiž prečítali slovo `BABKA` postupne sa nad každou jamou objavil most. Leming, ktorý sa zo štartovacieho domčeka vynoril keď sme začali čítať túto časť textu, tak má vždy most, na ktorý vie stúpiť a postupne príde do konca. Táto vlastnosť platí aj naopak. Ak leming príde do cieľového domčeka, znamená to, že počas jeho cesty sa pred ním museli objavovať mosty a teda sme prečítali slovo `BABKA`. Vidíme, že počet slov tvorených písmenami z jám v zadanom texte je rovnaký ako počet lemingov, ktorým sa podarilo prísť do cieľa.

Podúloha b.

Našou úlohou je vymyslieť 15 písmenný text, ktorý obsahuje slovo `KATKA` čo najviac krát. Je jasné, že ak budeme toto slovo opakovať jedno za druhým, zopakujeme ho 3 krát a vytvoríme text `KATKAKATKAKATKA`.

Ide to však aj lepšie a k tomu nám pomôžu práve lemingovia. Je napríklad jasné, že hľadaný text bude začínať slovom `KATKA...`, toto je totiž najrýchlejší spôsob ako dosiahnuť prvú zhodu a z každého textu, ktorý by takto nezačínal by sme mohli prvé písmeno odstrániť bez toho aby sme zmenšili počet výskytov. Takisto už vieme, že počet výskytov slova `KATKA` v texte je totožný počtu lemingov, ktorí prešli ponad jamy s písmenami slova `KATKA`. Vytvorme si teda takéto jamy a prečítajme začiatok textu, teda `KATKA....` Dostaneme nasledovnú situáciu.



Ako vieme dosiahnuť ďalší výskyt slova `KATKA`? Tak, že niektorého leminga dostaneme do cieľa. A najrýchlejšie to zvládne leming, ktorý stojí za druhou jamou. Aké písmená musíme prečítať aby prišiel do cieľa? Predsa tie, ktoré sú v jamách pred ním, teda `T`, `K` a `A`. Pridaním troch písmen dostaneme ďalší výskyt a náš text vyzerá: `KATKATKA....` V tomto momente je však situácia úplne rovnaká ako na obrázku a najbližšie k cieľu je opäť leming stojaci za druhou jamou. Dostávame preto text `KATKATKATKATKAT`, v ktorom sa slovo `KATKA` nachádza štyrikrát.

Samozrejme, k tomuto riešeniu sa dalo prísť čisto pozorovaním a skúšaním, je však pekné sa pozrieť na to, ako toto riešenie vlastne súvisí s pohybom lemingov a čo nám vedia o riešení prezradiť.

Podúloha c.

V podúlohe c. nastala situácia, v ktorej sme sledovali ako sa jeden leming postupne dostal až k cieľovému domčeku. Bohužiaľ sme si však nevšímali žiadnych iných lemingov a zaujíma nás, či ich polohu vieme napriek tomu určiť.

Koľko informácie teda vieme odvodiť z toho, že jeden leming stojí pred cieľom? V skutočnosti pomerne veľa. Ako sme si ukázali už v podúlohe a., to že sa tento leming dostal až na koniec znamená, že sa pred ním vždy musel objaviť most, aby po ňom prešiel a dostal sa ďalej. Tým pádom poznáme niekoľko posledných písmen, ktoré boli vyslovené. Úplne presne, vieme, že tieto písmená boli postupne všetky písmená z jám.

A to nám už stačí na vyriešenie celej úlohy. Uvedomme si totiž, že ak sa nejaký leming nachádza niekde uprostred cesty, tak sa zo štartu vynoril neskôr ako leming stojaci pri cieli. Tým pádom všetky písmená, ktoré

tento leming za svojho života počul, počul aj leming pri cieli. A síce nevieme, či tam skutočne je, vieme si odvodiť, čo by počul keby tam bol a skontrolovať či to sedí s jamami, ponad ktoré musel prejsť.

Ukážme si to trochu konkrétnejšie. Nech sa v jamách postupne nachádzajú písmená $j_1, j_2 \dots j_n$. To sú zároveň aj posledné písmená, ktoré boli vyslovené. Teraz by sme chceli zistiť, či sa za prvou jamou nachádza leming. Ak by tam bol, tak počul posledné písmeno, ktoré bolo vyslovené, teda j_n. Ale ponad prvú jamu mohol prejsť iba ak sa nad ňou objavil most, teda ak platilo, že $j_1 = j_n$.

A čo s pozíciou za druhou jamou? Je tam leming? Postupujme úplne rovnako. Ak by tam bol, tak postupne počul písmená j_{n-1} a j_n. Tieto písmená sa ale musia zhodovať s prvými dvoma jamami, teda znakmi j_1 a j_2.

Vyriešme teraz túto úlohu všeobecne. Označme si $j[zac : kon]$ slovo, ktoré dostaneme zo slova j ak vyberieme iba písmená na pozíciách zac až kon. Napríklad ak `j = UURUURUU`, tak `j[2:5] = URUU` a `j[4:8] = UURUU`. Potom platí, že za i-tou jamou je leming práve vtedy, ak `j[n-i+1:n]` (písmená, ktoré leming počul) a `j[1:i]` (jamy, nad ktorými musel prejsť) sú rovnaké.

V príklade na obrázku, kde je `j = UURUURUU` sa teda lemingovia nachádzajú za prvou jamou (`j[1:1] = U = j[8:8]`), druhou jamou (`j[1:2] = UU = j[7:8]`) a piatou jamou (`j[1:5] = UURUU = j[4:8]`). A samozrejme aj pred prvou, lebo tam je leming v podstate vždy.

Mimochodom, druhý možný prístup je, že ak poznáme všetky posledné písmená, ktoré boli vyslovené, môžeme jednoducho pochod lemingov odsimulovať a dostať sa tak k rovnakému riešeniu.

Hľadanie vzorky v texte

Pre zaujímavosť, všetko v tomto vzoráku úzko súvisí s hľadaním vzorky v texte. Úlohe, v ktorej máme v dlhom texte nájsť nejakú všetky výskyty kratšej vzorky, slova. V podstate klasická možnosť `Find` rôznych editorov. Dokonca, hrátky s lemingami vedú k optimálnemu riešeniu.

Základná myšlienka je rovnaká. Ak by sme simulovali pochod lemingov nad jamami s písmenami vzorky, ľahko nájdeme jej výskyty. Simulovať pohyb všetkých lemingov je však pomalé, oveľa jednoduchšie je sledovať len jedného, ktorý je najbližšie k cieľu. Tento leming sa totiž posunie dopredu ak sa ďalšie písmeno textu zhoduje s jamou, ktorá je pred ním, alebo padne a umrie. Jediný problém nastane, ak leming umrie. V tomto momente totiž musíme nájsť najbližšieho živého leminga, aby sme ho začali sledovať.

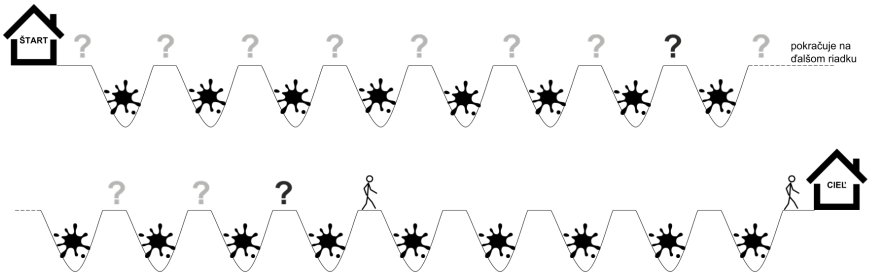
A pri tom vieme využiť podúlohu c. Vidíme totiž, že zo znalosti pozície niektorého leminga vieme zistiť, kde sa nachádzajú tí za ním. A dokonca to nesúvisí s textom, ktorý čítame, pretože ten je určený pozíciou nami sledovaného leminga. Túto informáciu si teda vieme vypočítať ešte pred tým, ako začneme čítať text. V podstate nás bude pre každú pozíciu zaujímať: Ak je na tejto pozícii živý leming, kde naľavo od neho je najbližší živý leming?

S touto informáciou je potom posúvanie leminga počas čítania textu jednoduché. Ak ide dopredu, tak ho proste posunieme a ak umrie, tak sa pozrieme do našej predpočítanej tabuľky aby sme zistili, ktorý leming ho nahradí.

Samozrejme, samotný algoritmus je ešte o niečo komplikovanejší, existuje napríklad spôsob ako si túto tabuľku predpočítať naozaj efektívne bez toho, aby sme simulovali celý proces, toto sú však hlavné myšlienky, ktoré sa v riešení používajú. V prípade, že by ste si chceli o tom prečítať viac, tento algoritmus sa volá KMP (podľa tvorcov Knuth, Morris a Pratt) a článok o ňom nájdete aj v KSP kuchárke.

Podúloha d.

Situácia, ktorú potrebujeme vyriešiť, vyzerá nasledovne:



Nepoznáme písmená v jamách a jediné čo vieme sú pozície dvoch najpravejších lemingov. Ako zistíme, či sú lemingovia aj za siedmou a jedenástou jamou? Postup z podúlohy c. nám nepomôže, pretože nepoznáme písmená a teda nevieme odsimulovať ako sa lemingovia hýbali.

Uvedomme si, že situácie sa v lemingove opakujú a sú závislé iba na tom, aké písmená sme čítali. To znamená, že vždy keď prečítame celé slovo z jám a leming sa ocitne pred cieľom, teda za sedemnástou jamou,

bude najbližšie k nemu leming za dvanástou jamou. Pozícia leminga pred cieľom totiž presne určuje čo počul leming za jamou 12 a tieto písmená sú stále rovnaké. Dokonca, z podúlohy c. vieme povedať, že ak si slovo z jám označíme `S`, tak platí `S[1:12] = S[6:17]`.

Predstavme si, že situáciu z obrázku chceme zopakovať, pokúsime sa teda leminga za 12 jamou dostať na koniec. Najbližších päť písmen budeme teda hovoriť presne to, čo potrebuje, aby sa mu vytvoril most. Nevadí, že tie písmená teraz nepoznáme, nám stačí len to, že by sme to vedeli spraviť, keby sme ich poznali, teda že taká možnosť existuje.

V okamihu ako sa tento leming ocitne pred cieľom, tak za 12 jamou sa musí objaviť nový leming. Lebo ak je na konci, tak situácia sa musí opakovať s tým, čo vidíme na pôvodnom obrázku. Z otáznikov sa teda po piatich písmenách vynoril leming a postavil sa za 12 jamu. Kde však tento leming začínal? Vráťme sa o 5 krokov späť a tento leming stojí za 7 jamou. Ukázali sme si teda, že za siedmou jamou sa musel nachádzať leming. Dokonca, rovnakým postupom by sme vedeli odôvodniť, že aj za druhou jamou je leming – to bude totiž ten, ktorý sa po piatich písmenách postaví za siedmu jamu.

Môže byť leming za jamou 11? Pozrime sa, čo by to znamenalo, keby tam bol. Posledné prečítané písmeno muselo vytvoriť most nad jamami 12 aj 11, obaja totiž prežili a prešli po nej. Teda `S[11] = S[12]`. Vráťme sa teda o krok dozadu. Opäť musí platiť, že posledné písmeno vytvorilo most aj nad 11 aj nad 10 jamou, teda `S[10] = S[11]`. Tento postup však môžeme opakovať až do úplného začiatku. Z toho vyplýva, že všetky písmená v jamách 1 až 12 musia byť rovnaké. Na začiatku sme si však uvedomili, že `S[1:12] = S[6:17]`, tým pádom `S[12] = S[17]`, `S[11] = S[16]` ... `S[1] = S[6]`. Teda nie len prvých 12 písmen v jamách je rovnakých, všetky písmená sú rovnaké.

To však nie je možné, ak by totiž všetky písmená boli rovnaké, tak ak sa nejaký leming dostane až k cieľu, za každou jednou jamou sa musí nachádzať leming. Nestalo by sa teda, že za jamami 13 až 16 by žiadni neboli. Tým pádom, chyba musí byť v našom predpoklade. Za jamou 11 sa leming nemôže nachádzať.

Podúloha e.

Ostáva nám nájsť najkratšiu periódu zadaného reťazca. Perióda dĺžky p je definovaná tak, že každá dvojica písmen vzdialených od seba o p písmen je rovnaká. Teda musí platiť, že `S[i] = S[i+p]` pre všetky možné i, čo sú všetky čísla od 1 po $n - p$. To sa však dá napísať aj tak, že `S[1:n-p] = S[p+1:n]`. Takýto zápis sme však už niekde videli nie? Toto bol presne zápis, ktorým sme zisťovali, či sa za jamou $n - p$ nachádza leming, ak je ďalší leming pri cieli.

Vidíme teda, že z pozícií dvoch posledných lemingov vieme určiť periódu celého textu. Stačí si odsimulovať pochod lemingov v prípade, že postupne čítame písmaná, ktoré sa nachádzajú nad jamami. Keď sa najpravejší leming nachádza pred domčekom, pozrieme sa, kde je k nemu najbližší živý leming. Počet jám medzi nimi udáva periódu celého reťazca. Táto perióda je navyše najkratšia možná.

Napriek tomu ako jednoducho správanie lemingov vyzeralo, pomohlo nám odvodiť si pomerne široké závery o reťazcoch všeobecne. Nielenže sme dokázali hľadať vzorku v texte, dokázali sme dokonca zistiť aj periódu zadaného reťazca.

3. Audio Práskači 2

vzorák napísal(a) Roman
(max. 15 b za riešenie)

K jednotlivým podúlohám sme pre vás opäť pripravili videovzoráky.

4. Sladká odmena

vzorák napísal(a) Denis
(max. 0 b za riešenie)

Riešenie simuláciou

Prvé dva vstupy boli relatívne malé a na získanie prvých 6 bodov si stačilo poriadne prečítať zadanie a napísať program, ktorý bude palacinku po palacinke aplikovať popísané pravidlá.

Každý tanier si môžeme predstaviť ako jednu premennú, v ktorej si pamätáme, koľko palaciniek sa na ňom aktuálne nachádza. A keďže chceme používať viacero veľmi podobných premenných, uložíme si ich do poľa `taniere[]`. Palacinky na prvom tanieri budú na pozícii `taniere[0]`, na druhom tanieri na pozícii `taniere[1]`... Pri položení novej palacinky najprv zvýšime číslo `taniere[0]`. Následne sa pozrieme, či na tomto tanieri nie sú 3 palacinky. Ak sú, nastavíme túto hodnotu na 0 a 1 pripočítali k `taniere[1]`. S tanierom na pozícii 1 musíme však opäť spraviť to isté, mohlo sa totiž stať, že sa na ňom zrazu nachádzajú tri palacinky. Pomocou `while` cyklu teda upravujeme počty palaciniek na tanieroch. V okamihu, keď niektorý z tanierov netreba upravovať sa totiž môžeme zastaviť.

Listing programu (Python)

```
n, k = map(int, input().split())
taniere = [0]*k

for i in range(n):
    taniere[0] += 1
    pozicia = 0
    while taniere[pozicia] == 3:
        taniere[pozicia] = 0
        taniere[pozicia + 1] += 1
        pozicia += 1

print(*taniere, sep='')
```

Na prvý pohľad by sa nám mohlo zdať, že časová zložitosť takéhoto riešenia je $O(n^2)$. Pre každú palacinku totiž musíme upraviť prinajhoršom všetkých n tanierov. V skutočnosti je však táto zložitosť výrazne lepšia. Skúsme sa zamyslieť, koľko palaciniek prejde tanierom každého člena rodiny.

Cez prvú osobu musí prejsť každá palacinka. K druhej osobe sa však dostane iba každá tretia palacinka, to bola totiž situácia, v ktorej človek jedna posunul palacinku človeku dva. Táto vlastnosť však platí všeobecne. Ak som i-ty človek v rade, tak vždy poslušne počkám, kým mám na tanieri 3 palacinky a až potom pošlem 1 človeku $i+1$. Tým pádom ale človek $i+1$ uvidí iba tretinu palaciniek človeka i. Prvý človek ich teda uvidí n, druhý $\frac{n}{3}$, tretí $\frac{\frac{n}{3}}{3} = \frac{n}{3^2}$, štvrtý $\frac{n}{3^3}$ atď.

Každé posunutie palacinky je pritom operácia, ktorú musíme spraviť v našom programe. Celková časová zložitosť takéhoto riešenia je preto:

$$n + \frac{n}{3} + \frac{n}{3^2} + \frac{n}{3^3} + \ldots$$

Nie je to na prvý pohľad síce jasné, ale súčet takejto postupnosti nikdy neprekročí hodnotu $2n$. V skutočnosti takýto súčet nikdy neprekročí ani hodnotu $1.5 \cdot n$. Prečo je tomu tak si nebudeme vysvetľovať, môžete si to však predstaviť tak, že sa snažíte prejsť vzdialenosť $2n$, ale každý ďalší váš krok je o tretinu menší ako ten predošlý. Nech sa snažíte ako chcete, ku koncu sa približujete príliš pomaly.

Trojková sústava

K riešenie za plný počet bodov je dobré vedieť, čo je to trojková sústava. Ako sa neskôr ukáže, táto znalosť nám vie riešenie celkom značne zjednodušiť.

Trojková sústava je číselná sústava, ktorá zapisuje hodnoty pomocou troch symbolov – 0, 1 a 2. Bežne sme zvyknutí pracovať s číslami v desiatkovej sústave, kde na vyjadrenie rádov používame jednotky, stovky, tisícky atď. čo sú všetko mocniny desiatky. Napríklad číslo 241_{10} vieme zapísať ako $2 \cdot 10^2 + 4 \cdot 10^1 + 1 \cdot 10^0$.

V trojkovej sústave budeme preto na vyjadrenie rádov používať mocniny trojky. Napríklad zápis čísla 38_{10} by v trojkovej sústave vyzeral ako 1102_3, čo opäť vieme zapísať ako $1 \cdot 3^3 + 1 \cdot 3^2 + 0 \cdot 3^1 + 2 \cdot 3^0 = 38$. V desiatkovej sústave sa číslo prenáša do vyššieho rádu vtedy, ak už dosiahlo desať násobok svojej hodnoty, keďže desať násobok rádu 10^i je vlastne hodnota rádu 10^{i+1}. Toto isté platí aj o trojkovej sústave, s tým rozdielom, že číslo prechádza do vyššieho rádu už pri trojnásobku rádu i.

Ako teda vieme trojkovú sústavu využiť pri našom riešení?

Vzorové riešenie

Predstavme si, že v našom zadaní by sme namiesto posúvania palaciniek po troch ich posúvali až v momente, keď by sme ich na tanieri mali 10. Vtedy by nám počty palaciniek na tanieroch tvorili zápis čísla v desiatkovej sústave. Nech je na tanieroch postupne 3, 5 a 1 palacinka. Keďže na najpravejší tanier sa dostane iba každá stá palacinka, to, že sa tam nachádza značí, že sme už museli poposúvať 100 palaciniek. Na druhom mieste je však číslo 5 a na tento tanier sa dostane iba každá desiata palacinka. To znamená, že navyše k tým 100, z ktorých sa jedna dostala na tanier tri a zvyšné boli zjedené sme museli pridať 50 ďalších palaciniek. A číslo 3 na prvom tanieri značí, že navyše ich bolo treba ešte 3. Dokopy sme teda museli poposúvať $1 \cdot 100 + 5 \cdot 10 + 3 \cdot 1 = 153$ palaciniek. Navyše si môžeme všimnúť, že pridanie ďalšej palacinky na prvý tanier je rovnaké, ako keby sme k tomuto číslu pripočítali 1.

V našom prípade však neposúvame po desiatich, ale už po troch. Výsledné počty palaciniek teda reprezentujú číslo nie v desiatkovej, ale trojkovej sústave. A pridanie palacinky je rovnako pripočítanie čísla 1. Z toho vyplýva, že ak chceme kde sa po pridaní n palaciniek nachádza koľko z nich, musíme zistiť, aký je zápis čísla n v trojkovej sústave.

Prevod do trojkovej sústavy bude vyzerať nasledovne. Ak chceme v desiatkovej sústave vedieť, koľko jednotiek sa v danom čísle nachádza, potrebujeme poznať jeho zvyšok po delení desiatkou. Číslo teda vydelíme, zapamätáme si výsledok delenia a zvyšok. Ak chceme vedieť ďalší rád, výsledok delenia opäť vydelíme a zvyšok

nám určí početnosť ďalšieho rádu. Takto postupujeme až kým nedosiahneme nulu ako výsledok delenia. Tento istý postup vieme použiť aj pri prevádzaní čísla do trojkovej sústavy, akurát namiesto delenia 10 budeme deliť 3.

Ako bude teda vyzerať náš program? Na vstupe dostaneme dve čísla n a k. Ako sme si vyššie popísali, naším cieľom bude previesť číslo n z desiatkovej sústavy do trojkovej, čo spravíme pomocou postupného delenia a zapisovania si zvyškov. A keďže naše číslo musíme vypísať na k cifier, pridáme na koniec niekoľko núl.

Listing programu (C++)

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

using namespace std;

int main(int argc, const char * argv[]) {
    long long n;
    int k;
    cin >> n >> k;

    string vystup = "";
    // pokým je n väčšie ako 0 opakujeme vyššie opísaný proces
    while (n > 0) {
        // zvyšok po delení 3 si rovno zapíšeme do nášho výstupu,
        // preto ho potrebujeme prekonvertovať na string
        vystup += to_string(n % 3);
        n /= 3;
    }
    // Keďže sa môže stať, že nie každému sa pri stole ušli palacinky,
    // pre zvyšok ludí doplníme 0
    while (vystup.size() < k) vystup += "0";

    cout << vystup << "\n";
    return 0;
}
```

Je tento prístup rýchlejší ako ten predtým? Áno, pretože v každom kroku sa dozvieme počet palaciniek na ďalšom tanieri. To znamená, že nám stačí iba k krokov, aby sme sa dopracovali k výsledku. Výsledná zložitosť je preto $O(k)$.

vzorák napísal(a) Andrej
(max. 15 b za riešenie)

5. Kopa neumytého riadu

Pri programoch, ktoré majú niečo simulovať, je vždy dobré si vyskúšať zopár príkladov vstupu a pochopiť, ako má daný program fungovať. To vedie k riešeniu, ktoré síce nie je rýchle, ale zato je určite správne. Objavený postup totiž stačí prepísať pomocou programovacieho jazyka.

Úvod do zásobnika

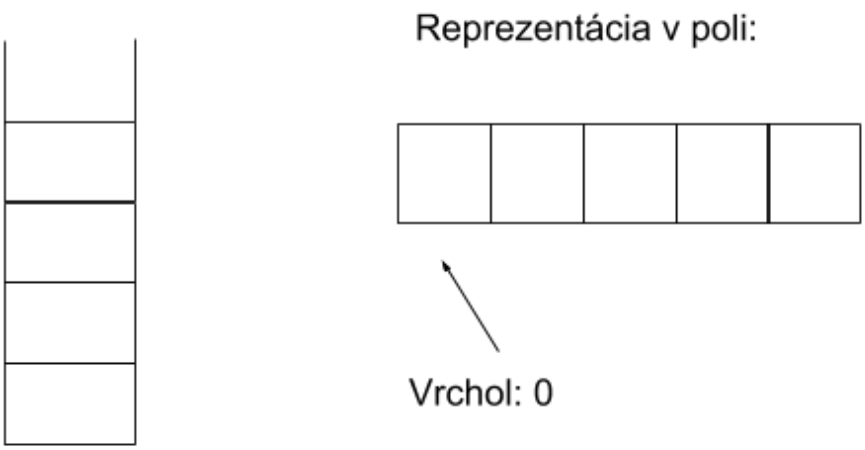
Pokiaľ viete čo je dátová štruktúra zásobník, môžete túto časť spokojne preskočiť.

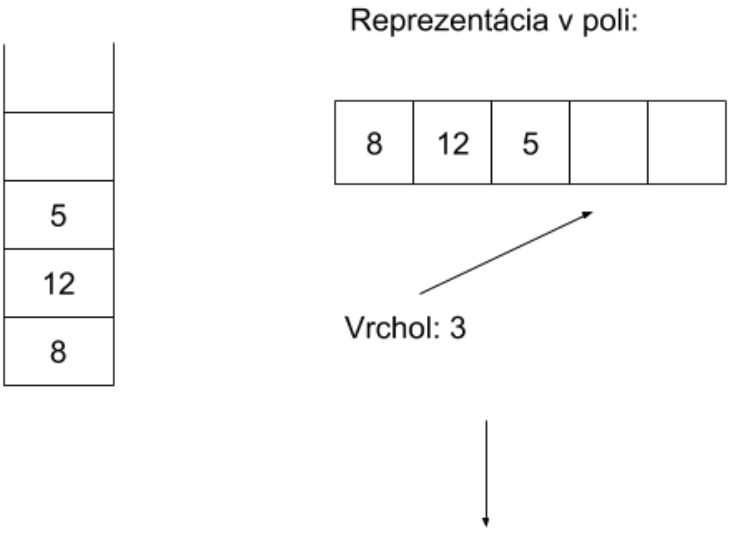
Kopa tanierov, ktorú simulujeme sa v mnohom správa ako *abstraktná dátová štruktúra* s názvom zásobník. Nenechajte sa vystrašiť. Slovné spojenie abstraktná dátová štruktúra iba vyjadruje, že napriek tomu, že na uloženie informácií, napríklad čísel, používame pole a pamäť počítača, dáta máme uložené v špeciálnej, nami zvolenej forme.

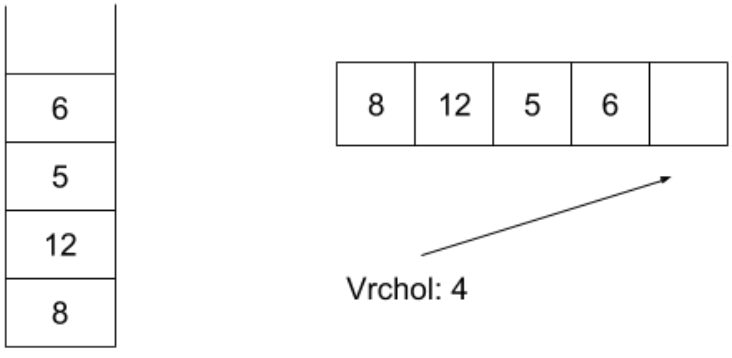
Inak povedané, počítaču pripadá uloženie dát náhodné, pre nás má však hlbší zmysel. Práve to vyjadruje slovo abstrakcia. Zásobník má ako aj iné dátové štruktúry dve hlavné funkcie – vieme doň prvky (napr. čísla) pridávať a vieme z neho prvku odoberať. Obe tieto operácie však majú svoje obmedzenia. V prípade zásobníku platí, že pridávať a odoberať prvky vieme iba z vrchu zásobníka.

Predstavte si, že máte kopu tanierov. Keď nejaký špinavý pridávate, položíte ho na vrch kopy. A keď nejaký idete umyť, je asi prirodzené začať tým najvrchnejším, čím ho z vrchu kopy odoberiete. Takto funguje aj dátová štruktúra zásobník. Obmedzenie toho, ako vieme vkladať a vyberať nám dáva určitú silu. Môžete si všimnúť, že keď do zásobníka vložíme niekoľko čísel tak platí, že posledné vložené je na vrchu zásobníka. Predposledné vložené čislo je zas priamo pod vrchom zásobníka atď.

Zásobník vieme ľahko simulovať pomocou poľa. Vytvoríme si dostatočne veľké pole a jednu premennú, v ktorej si pamätáme index na miesto v zásobníku. Tento index nám ukazuje, kam si poznačiť vkladané číslo. Po jeho pridaní do poľa index zväčšíme. Pri odstraňovaní stačí naopak index iba zmenšiť.







Príklady implementácie zásobníka v jazykoch C++ a Python si môžete pozrieť tu:

Listing programu (C++)

```
#include <iostream>
using namespace std;
const int MAXI_VELKOST = 1000;
```

```
int zasobnik[MAXI_VELKOST];
int vrchol = 0;

void pridaj(int x)
{
    zasobnik[vrchol] = x;
    vrchol++;
}

void odober_vrch()
{
    if(vrchol>0) vrchol--;
}

int vrat_vrch()
{
    if(vrchol>0) return zasobnik[vrchol-1];
    return -1;
}

int velkost()
{
    return vrchol;
}

int main()
{
    pridaj(10);
    pridaj(80);

    cout << "Velkost zasobnika je " << velkost() << endl;

    pridaj(30);

    odober_vrch();
    cout << "Na vrchu je: " << vrat_vrch() << endl;

    return 0;
}
```

Listing programu (Python)

```
zasobnik = []

def pridaj(x):
    zasobnik.append(x)

def odober_vrch():
    zasobnik.pop()

def vrat_vrch():
    if len(zasobnik) > 0:
        return zasobnik[-1]
    else:
        return -1

def velkost():
    return len(zasobnik)

pridaj(10)
pridaj(80)

print("Velkost zasobnika je " + str(velkost()))

pridaj(30)

odober_vrch()

print("Na vrchu je: " + str(vrat_vrch()))
```

Ak si chcete o zásobníku a niektorých ďalších jednoduchých dátových štruktúrach prečítať o kúsok viac, odporúčame zájsť do KSP kuchárky.

Pomalšia simulácia kopy

Ak už vieme čo zásobník je, môžeme sa pustiť do prvého riešenia. Vidíme, že naša kopa tanierov sa správa presne ako zásobník. Navyše však musíme vedieť zistiť aj to, aká je najmenšia hodnota čísel v zásobníku.

Pokiaľ máme zásobník uložený tak ako sme si spomínali vyššie, teda v poli či liste, je jednoduché pri každom príkaze typu `NAJSPINAVSI` prejsť týmto poľom a zistiť, aké najmenšie číslo sa v ňom nachádza. Toto riešenie nám prinesie zaslúžených 5 bodov, ale na väčšie vstupy nebude dostatočne rýchle. Dôvodom je, že pri každom takomto príkaze hľadáme minimum spomedzi všetkých čísel v zásobníku, ktorých tam môže byť časom naozaj veľmi veľa.

Listing programu (C++)

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

```cpp
using namespace std;

const int MAXI_VELKOST = 1000000;

int zasobnik[MAXI_VELKOST];
int vrchol = 0;

void pridaj(int x)
{
    zasobnik[vrchol] = x;
    vrchol++;
}

void odober_vrch()
{
    if(vrchol>0) vrchol--;
}

int vrat_vrch()
{
    if(vrchol>0) return zasobnik[vrchol-1];
    return -1;
}

int velkost()
{
    return vrchol;
}

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

    for(int i=0;i<n;++i)
    {
        string operacia;
        cin >> operacia;

        if(operacia == "PRIDAJ")
        {
            int cislo;
            cin >> cislo;

            pridaj(cislo);
        }
        else if(operacia == "UMY") odober_vrch();
        else
        {
            int mini = -1;

            if(velkost() != 0) mini = zasobnik[0];

            for(int i=0; i<velkost(); ++i) mini = min(mini, zasobnik[i]);

            cout << mini << "\n";
        }
    }

    return 0;
}
```

Listing programu (Python)

```python
zasobnik = []

def pridaj(x):
    zasobnik.append(x)

def odober_vrch():
    zasobnik.pop()

def vrat_vrch():
    if len(zasobnik) > 0:
        return zasobnik[-1]
    else:
        return -1

def velkost():
    return len(zasobnik)


n = int(input())

for i in range(n):
    vstup = input()

    if vstup == "NAJSPINAVSI":

        if velkost() == 0:
            print(-1)
            continue

        mini = 10**10
        for i in zasobnik:
```

```
            if (i<mini):
                mini = i;

        print(mini)

    elif vstup == "UMY":
        if velkost() > 0:
            odober_vrch()
    else:
        x = int(str(vstup[7:]))
        pridaj(x)
```

Rýchlejšia simulácia kopy

Tento odstavec je venovaný druhej sade, kde sú čísla, ktoré sa nachádzajú v kope resp. v zásobníku z malého rozsahu. Ako vieme tento fakt využiť pri riešení? Použijeme techniku, ktorá sa v takýchto prípadoch používa pomerne často.

Technika spočíva v tom, že si v pomocnom poli budeme na index i značiť počet čísel i v zásobníku. Zásobník bude aj naďalej fungovať rovnako, toto pomocné pole nám pomôže iba pri hľadaní najmenšieho čísla.

Keď do zásobníka pridáme číslo i, zväčšíme v pomocnom poli číslo na i-tom políčku o 1, pri vyberaní zo zásobníka ho zase zmenšíme. Pri hľadaní minima nám zjavne stačí prejsť toto pomocné pole a nájsť prvé políčko, ktoré obsahuje nenulové číslo. Uvedomme si, že napriek tomu, že zásobník môže byť obrovský, naše pomocné pole je, vďaka obmedzeniu na maximálnu veľkosť čísel na vstupe, malé. Toto riešenie nám prinesie tiež 5 bodov. Implementácia popísaného riešenia môže vyzerať napríklad takto:

Listing programu (C++)

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

using namespace std;

const int MAXI_VELKOST = 1000000;

int zasobnik[MAXI_VELKOST];
int vrchol = 0;

void pridaj(int x)
{
    zasobnik[vrchol] = x;
    vrchol++;
}

void odober_vrch()
{
    if(vrchol>0) vrchol--;
}

int vrat_vrch()
{
    if(vrchol>0) return zasobnik[vrchol-1];
    return -1;
}

int velkost()
{
    return vrchol;
}

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

    int kompres[1001] = {0};

    for(int i=0;i<n;++i)
    {
        string operacia;
        cin >> operacia;

        if(operacia == "PRIDAJ")
        {
            int cislo;
            cin >> cislo;

            pridaj(cislo);
            ++kompres[cislo];
        }
        else if(operacia == "UMY")
        {
            --kompres[zasobnik[vrchol-1]];
            odober_vrch();
        }
        else
        {
            int mini = -1;
            for(int i=0;i<1001;++i) if(kompres[i]>0)
```

```
            {
                mini = i;
                break;
            }
            cout << mini << "\n";
        }
    }

    return 0;
}
```

Listing programu (Python)

```
# vytvorime si list na zasobnik a potom na pomocne pole
zasobnik = []
kompresia = [0] * 1001

def pridaj(x):
    zasobnik.append(x)

def odober_vrch():
    zasobnik.pop()

def vrat_vrch():
    if len(zasobnik) > 0:
        return zasobnik[-1]
    else:
        return -1

def velkost():
    return len(zasobnik)


################################


n = int(input())

for i in range(n):
    # nacitame operaciu a podla toho sa rozvetvime
    vstup = input()

    if vstup == "NAJSPINAVSI":
        # ak hladame najspinavsi tanier, staci prejst pomocne pole a najst prvy
        # prvok, ktory sa v zasobniku nachadza aspon raz
        mini = -1

        for i in range(1001):
            if kompresia[i] > 0:
                mini = i
                break

        # vypiseme tento najmensi prvok
        print(mini)

    elif vstup == "UMY":
        # skontrolujeme, ci je co umyt
        if velkost() == 0:
            continue

        # zmensime pocet krat, kolko sa posledny prvok nachadza v zasobniku
        kompresia[zasobnik[-1]] = kompresia[zasobnik[-1]] - 1

        # vyberieme ho
        odober_vrch()

    else:
        # zistime, ake cislo mame pridat
        x = int(str(vstup[7:]))

        # upravime pomocne pole
        kompresia[x] = kompresia[x] + 1

        # a pridame ho do zasobnika
        pridaj(x)
```

Kombinácia riešení

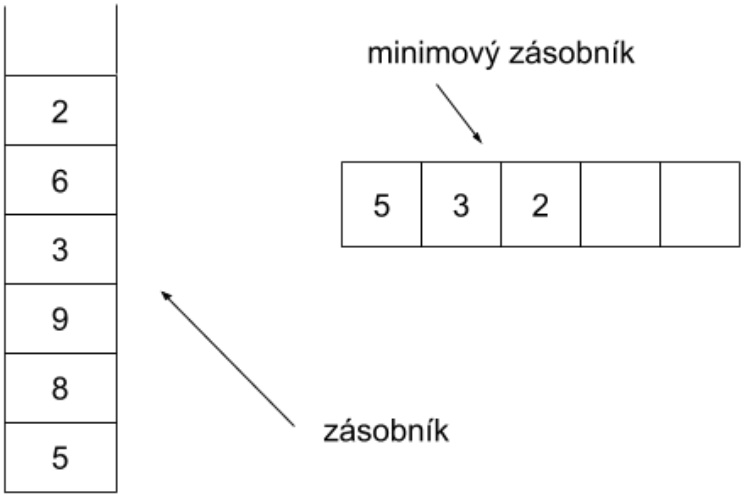
V prípade, že ste prišli na obidve vyššie spomenuté riešenia, darí sa vám získať po 5 bodov na rôznych sadách vstupov. Môžeme ich preto šikovne spojiť do jedného. V tomto riešení si opäť vytvoríme pomocné pole, do ktorého si budeme značiť počet čísel i v zásobníku a toto pole budeme používať na hľadanie najmenšieho čísla. Ak sa ale stane, že na zásobník by sme mali pridať príliš veľké číslo, prepneme náš program do druhého módu, v ktorom bude pri hľadaní minima prechádzať celý zásobník. V druhej sade takúto zmenu riešenia nespravíme, lebo čísla budú malé a v prvej sade síce budeme prechádzať celý zásobník, ale ten nemôže byť príliš veľký. Výsledkom je riešenie za 10 bodov.

Vzorové riešenie

Predstavme si, že v zásobníku sú uložené nejaké čísla a poznáme najmenšie z nich, ktoré si označíme x.

V prípade, že na zásobník pridáme číslo väčšie ako x, minimum sa nezmenilo a môžeme pokojne pokračovať ďalej. Problém nastane iba v prípade, že na zásobník sa pridá číslo y, ktoré je menšie ako x. V tom prípade sa novým minimom má stať číslo y. Uvedomme si však dôležitú vec. Kým bude číslo y v zásobníku, je menšie ako x a preto x minimom nebude. Ak ale y zo zásobníka odstránime, zásobník bude vyzerať rovnako, ako tesne predtým, ako sme doň y vložili. Tým pádom sa x stane opäť minimom.

Z toho vyplýva, že pri zmene minima si jeho starú hodnotu môžeme poznačiť, pretože sa nám v budúcnosti môže ešte zísť. Situácia je však o niečo komplikovanejšia. Zoberme si totiž vyššie popísaný príklad, v ktorom y nahradilo minimum x a to sme si teda niekam poznačili. Skôr ako y zo zásobníka odstránime však doň pridáme číslo z, ktoré je ešte menšie ako y. V tomto prípade sme v podobnej situácii. Ak sa totiž v budúcnosti stane, že z zo zásobníka odstránime, y sa opäť stane minimom a my by sme si to mali zapamätať. K číslu x si teda poznačíme aj číslo y. Keď potom z odíde zo zásobníka, za nové minimum označíme posledné poznačené číslo, teda y. A keď bude odstránené aj to, opäť siahneme po poslednom poznačenom čísle, teda x.



V skutočnosti sa nám tu formuje druhý zásobník, do ktorého si značíme akúsi "históriu miním". Keď pridávame do zásobníka číslo, ktoré je väčšie ako doterajšie minimum, nič sa nedeje. Ak však pridávame číslo menšie, musíme zmeniť aktuálne minimum na novú hodnotu a tú starú si zapíšeme do zásobníka miním. Pri vyberaní čísel zo zásobníka sa nič nedeje, kým neodstránime aktuálne najmenší prvok. V takom prípade sa totiž minimum musí zmeniť a my siahneme po vrchu zásobníka miním, kde na nás čaká správna hodnota, ktorú z tohto zásobníka tiež odstránime.

Takéto riešenie je v skutočnosti veľmi rýchle. Pri každej zmene zásobníka totiž nové minimum vypočítame v konštantnom čase. Buď sa totiž nezmení, alebo je prepísané pridaným číslom, alebo sa nastaví na hodnotu najvyššieho čísla v zásobníku histórie miním. Celková časová zložitosť je preto $O(n)$. Vo vzorovej implementácii navyše používame ešte jeden trik. Najmenšie číslo si do histórie miním vkladáme po každom pridaní prvku. Vďaka tomu nemusíme kontrolovať, či odstraňované číslo je aktuálnym minimom, po vrchu historického zásobníka siahneme zakaždým. Navyše nám to s ľahkosťou vyrieši prípady, keď sa v zásobníku nachádza viacero rovnakých čísel, čo sme predtým museli špeciálne ošetrovať.

Listing programu (C++)

```
#include <iostream>
#include <climits>

using namespace std;

const int MAXI_VELKOST = 1000000;

int zasobnik[MAXI_VELKOST];
int minimum[MAXI_VELKOST];

int vrchol = 0;

void pridaj(int x)
{
    int t = INT_MAX;

    if(vrchol != 0) t = minimum[vrchol-1];

    minimum[vrchol] = min(t, x);
    zasobnik[vrchol] = x;
```

```
    vrchol++;
}

void odober_vrch()
{
    if(vrchol>0) vrchol--;
}

int vrat_vrch()
{
    if(vrchol>0) return zasobnik[vrchol-1];
    return -1;
}

int velkost()
{
    return vrchol;
}

int vrat_minimum()
{
    if(vrchol == 0) return -1;
    return minimum[vrchol-1];
}

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

    for(int i=0;i<n;++i)
    {
        string operacia;
        cin >> operacia;

        if(operacia == "PRIDAJ")
        {
            int cislo;
            cin >> cislo;

            pridaj(cislo);
        }
        else if(operacia == "UMY")
        {
            odober_vrch();
        }
        else
        {
            cout << vrat_minimum() << "\n";
        }
    }

    return 0;
}
```

Listing programu (Python)

```
zasobnik = []
minimum_zasobnik = []

def pridaj(x):
    if len(minimum_zasobnik) == 0:
            minimum_zasobnik.append(x)
    else:
            minimum_zasobnik.append(min(x, minimum_zasobnik[-1]))
    zasobnik.append(x)

def odober_vrch():
    zasobnik.pop()
    minimum_zasobnik.pop()

def vrat_vrch():
    if len(zasobnik) > 0:
        return zasobnik[-1]
    else:
        return -1

def velkost():
    return len(zasobnik)


#######################################

n = int(input())

for i in range(n):
    vstup = input()

    if vstup == "NAJSPINAVSI":

        if velkost() == 0:
            print(-1)
            continue

        print(minimum_zasobnik[-1])
```

```
elif vstup == "UMY":
    if velkost() == 0:
        continue

    odober_vrch()

else:
    x = int(str(vstup[7:]))
    pridaj(x)
```