PRASK
VI. ročník, 2019/20
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

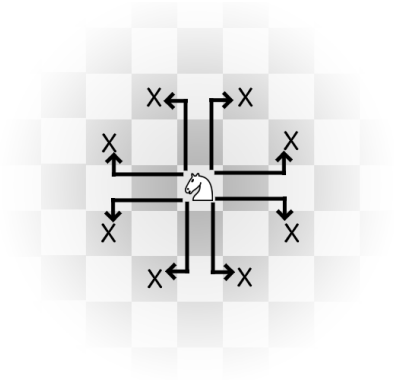
Vzorové riešenia 1. kola zimnej časti

1. Poskakujúci kôň

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

a) *(2 body) Políčko* $(0,0)$ *má bielu farbu a zvyšené políčka sú ofarbené šachovnicovo. Akú farbu bude mať políčko, na ktoré sa jazdec dostane po* X *skokoch? Prečo?*

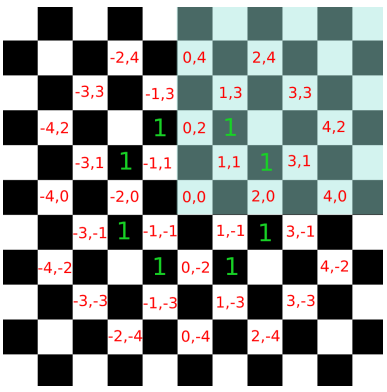
Všimnite si, že kôň pri každom skoku zmení farbu, na ktorej stojí. Túto vlastnosť je dobre vidieť na obrázku zo zadania, ktorý ukazuje všetky platné skoky koňa.



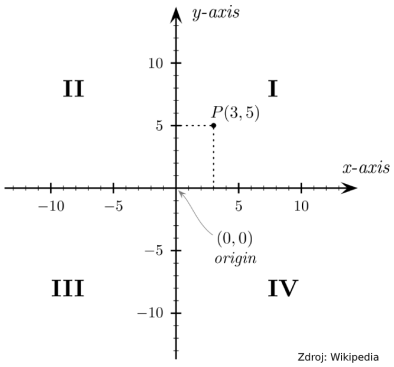
Kôň začal na bielom políčku a všetky jeho skoky vedú na čierne políčko. Analogicky ak kôň skočí z čierneho políčka, skončí na bielom políčku. Po dvoch skokoch teda vždy bude na farbe, na ktorej začínal. Preto ak je X párne, kôň skončí na bielom políčku, a naopak, ak je X nepárne, skončí na čiernom.

b) *(2 body) Jazdec zo svojho začiatočného políčka urobil presne dva skoky. Na koľkých rôznych políčkach môže teraz stáť? Aké súradnice majú tieto políčka?*

Vieme, kam sa jazdcom dá dostať na jeden skok (obrázok vyššie). Skúsime teda z každej z týchto pozícií skočiť s jazdcom znovu.



Na obrázku zelené jednotky značia, kam sa vieme dostať na jeden skok a červené súradnice sú práve súradnice pozícií, kam sa možno dostať na dva skoky. Všimnite si, že ak políčko $(0,0)$ posadíme do stredu súradnicovej sústavy, možné pozície kam sa vieme dostať na dva skoky, budú symetrické podľa osí pretínajúcich $(0,0)$ zvisle aj vodorovne. Stačí preto nájsť políčka v jednom z *kvadrantov*, napríklad prvom, ktorý je na obrázku označený bledomodrým pozadím. V rovine je kvadrant jeden zo štyroch výsekov určenými osami súradníc. Číslujeme ich $I - IV$ tak, ako na obrázku nižšie.



Políčka z prvého kvadrantu zobrazíme do ostatných podľa osových súmerností jednoducho – zmeníme im znamienka pri súradniciach. Napríklad políčku $(3, 1)$ v ostatných kvadrantoch zodpovedajú súradnice $(-3, 1)$, $(-3, -1)$, $(3, -1)$. Vďaka tomu všetky políčka, na ktoré sa vieme dostať na dva skoky, môžeme zapísať stručnejšie:

$$\{(xi, yi) \mid (x, y) \in \{(0,0), (2,0), (4,0), (1,1), (3,1), (0,2), (4,2), (1,3), (3,3), (0,4), (2,4)\}, i, j \in \{-1, 1\}\}$$

Vytvorili sme množinu, ktorá obsahuje všetky dvojice (x, y) z prvého kvadrantu a ich obrazy v ostatných kvadrantoch $(-x, y)$, $(-x, -y)$, $(x, -y)$.

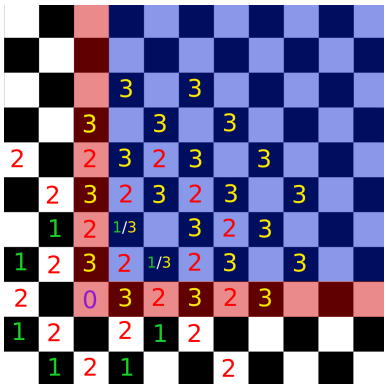
c) *(2 body) Koľko najmenej skokov potrebuje jazdec na to, aby sa dostal zo svojho začiatočného políčka na ľubovoľné zo 4 políčok, ktoré so začiatočným susedia stranou (políčka so súradnicami* $(0, 1)$, $(1, 0)$, $(0, -1)$ *a* $(-1, 0)$*)? Prečo to nejde na menej skokov?*

Všimnite si, že na pozíciu $(1, 0)$ sa vieme dostať napríklad z políčka $(3, 1)$, kam sme sa dostali na 2 skoky (viď. predchádzajúca podúloha). Na políčko $(1, 0)$ sa vieme teda dostať na najviac 3 skoky. Obdobne sa vďaka symetrii na najviac 3 skoky vieme dostať aj na políčka $(0, 1)$, $(0, -1)$ a $(-1, 0)$.

Prečo to nejde na menej skokov ako 3? Našli sme všetky políčka, ktoré vieme navštíviť na 0 skokov $((0, 0))$, 1 skok (obrázok z a)), 2 skoky (obrázok z b)) a v žiadnych z nich sa políčka susedné s $(0, 0)$ nenachádzali. Na tieto pozície sa vieme dostať na jeden skok až vďaka políčkam, ktoré sme objavili na 2 skoky. Spolu sú to tak 3 skoky.

d) *(2 body) Jazdec zo svojho začiatočného políčka urobil presne tri skoky. Na koľkých rôznych políčkach môže teraz stáť? Popíšte postup, ktorým ste sa dopracovali k svojmu riešeniu. Na rozdiel od podúlohy b) nemusíte do popisu písať súradnice týchto políčok.*

Podobne ako v podúlohe b) využijeme symetriu. Nájdeme všetky políčka v prvom kvadrante, kam sa vieme dostať na jeden skok, potom na dva, až na koniec na tri.



Z obrázku vidíme, že v jednom kvadrante je 22 políčok, ktoré vieme navštíviť práve na tri skoky. Nestačí ale 22 vynásobiť štyrmi za každý kvadrant, pretože políčka na osiach by sme započítali dvakrát. Rozdelíme si preto počty políčok na tie v strede $S = 16$ a tie na osiach $O = 6$. Na obrázku sú stredné políčka označené modrou a obvodové červenou farbou. Teraz stačí stredy započítať štyrikrát za každý kvadrant a osi iba dvakrát:

$$4 \cdot S + 2 \cdot O = 4 \cdot 16 + 2 \cdot 6 = 76$$

.

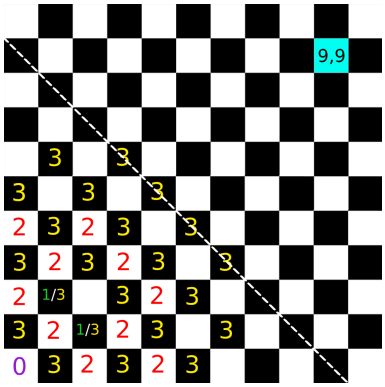
e) *(3 body) Koľko najmenej skokov potrebuje jazdec na cestu z políčka* $(0,0)$ *na políčko* $(9,9)$*? Prečo to nejde na menej? Popíšte postup, ktorým ste sa dopracovali k svojmu riešeniu.*

Skúsime postupne nachádzať políčka, na ktoré sa vieme dostať najskôr na $1, 2, 3, \ldots$ skoky. Predstavte si, že postupne z bodu $(0,0)$ odhaľujeme šachovnicu a na každé políčko, na ktoré skočíme, si zapíšeme, v ktorom skoku sa nám tam podarilo prvýkrát dostať.

Do bodu $(0,0)$ sa vieme dostať na 0 skokov. Z neho vieme postupne skočiť na osem políčok, ktoré sme predtým ešte nenavštívili. Tieto políčka označíme za *Vlnu 1*, pretože sú to políčka, na ktoré sa vieme dostať najskôr na 1 skok. Zapíšeme si preto na ich pozície 1 a skúsime z týchto 8 políčok postupne skočiť ďalej. To nám spolu dá $8 \cdot 8$ nových pozícií. Niektoré políčka možno navštívime dvakrát. Napríklad určite pri tomto kroku skočíme späť do počiatku $(0,0)$. Tu však počet skokov už nevylepšíme, pretože sme sa sem aktuálne dostali na 2 skoky, pričom pri $(0,0)$ sme si predtým poznačili, že sa tam vieme dostať na 0 skokov. Z bodu $(0,0)$ sa nám už preto neoplatí skákať ďalej, pretože by sme objavovali už objavené políčka a navyše s väčším počtom skokov!

Avšak na pozície, ktoré sme **navštívili prvýkrát**, si zapíšeme počet skokov 2 a označíme ich za *Vlnu 2*. Z *Vlny 2* potom skúsime skákať ďalej a vytvoríme *Vlnu 3*, z *Vlny 3* skáčeme ďalej a vytvoríme *Vlnu 4*, atď. Na obrázku môžete vidieť tento postup v prvom kvadrante, až pokiaľ nedosiahneme bod $(9,9)$ vpravo hore, označený bledomodrou farbou. Z obrázku vidíme, že na toto políčko sme sa prvýkrát dostali po 6 skokoch. Prečo to nejde na menej? *Vlna i* totiž obsahuje práve tie políčka, na ktoré sa vieme dostať najskôr na i skokov.

Alternatívne sa táto podúloha dala vyriešiť trochu jednoduchšie, s použitím symetrie stačilo spraviť prvé 3 kroky "vlny" (skúste sa zamyslieť prečo):



f) *(4 body) Popíšte všeobecný postup, ako by ste zisťovali, na koľko najmenej skokov sa vie jazdec dostať z políčka* $(0,0)$ *na políčko* (X,Y)*. Skúste popísať, prečo je váš postup správny.*

Vlna i z predošlej podúlohy obsahuje práve tie políčka, do ktorých sa vieme dostať najskôr na i skokov. Prečo? Ak sa na ľubovoľné, doteraz nenavštívené, políčko P vieme dostať z nejakého políčka z *Vlny i-1*, potom najkratšia cesta skokmi do políčka P musí mať dĺžku i, pretože P nebolo objavené v skorších vlnách. Zároveň do *Vlny i* pridáme všetky doteraz neobjavené políčka, na ktoré vieme skočiť z políčok *Vlny i-1*. *Vlna i* tak obsahuje všetky políčka, do ktorých sa vieme dostať najskôr na i skokov. Na záver, *Vlna 0* triviálne obsahuje všetky vrcholy s najkratšou cestou dĺžky *0*, pretože obsahuje jediné políčko $(0,0)$.

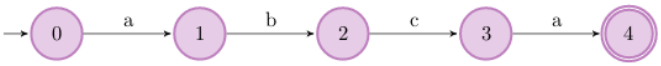
Na nájdenie najmenšieho počtu skokov potrebných na dosiahnutie bodu (X,Y) tak stačí pokračovať s "Algoritmom vlny" popísaným vyššie až dovtedy, pokiaľ nenarazíme na bod (X,Y). Ak naň narazíme v i-tej vlne, vieme, že najkratšia cesta do (X,Y) je dlhá i skokov.

"Algoritmus vlny" popísaný vyššie sa volá *BFS (Breadth first search)* alebo po slovensky *Hľadanie do šírky*. Všimnite si, že názov odpovedá presne tomu, čo algoritmus vykonáva. Postupne zo všetkých políčok v aktuálnej vlne skúsime objaviť ďalšie, čím objavujeme mapu do všetkých smerov, teda do šírky. Ide o takzvaný grafový problém (jednotlivé políčka tvoria vrcholy grafu). Ak sa chceš dozvedieť viac o grafoch v informatike a o tom ako sa prehľadávanie do šírky implementuje pomocou fronty, pozri sa do našej kuchárky alebo do knihy Průvodce labyrintem algoritmů.

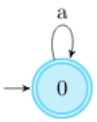
2. Receptomaty

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

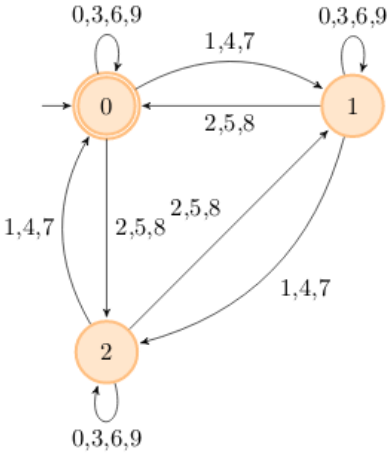
a) (1 body) Automat pre túto úlohu bol podobný ako v zadaní príkladu, jediná zmena bol akceptovaný recept ("abca" miesto "aab").



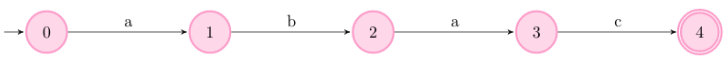
b) (2 body) Je diskutovateľné, či recept dĺžky 0 sa skladá iba z "a". Formálne môžeme povedať, že pre všetky kroky receptu "" (prázdny recept) platí, že sú "a". Uznávané sú však aj receptomaty, ktoré neakceptujú prázdne recepty. Na riešenie nám teda stačí spraviť takýto cyklus:



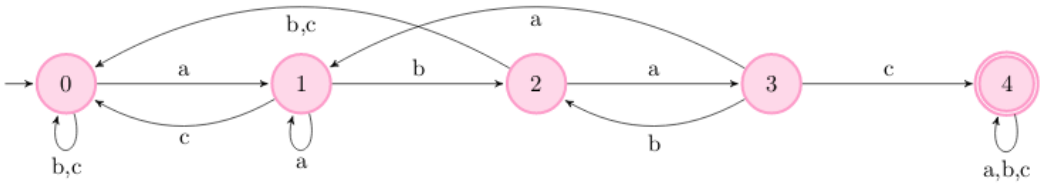
c) (3 body) Pre túto úlohu je dôležité si uvedomiť, ako funguje deliteľnosť tromi (ciferný súčet musí byť deliteľný tromi). Majme teda číslo, ktorého ciferný súčet je zvyšku 2 po delení 3. Pripojíme teraz k nemu číslo, ktoré dáva 1 po delení 3. Súčet 2+1 je 3 a teda po delení tromi bude ciferný súčet nulový. Príklad: 104 dáva zvyšok 2 po delení tromi. Ak pripojíme 1 (máme teda 1041), dostaneme číslo deliteľné 3. Model receptomatu, ktorý takto funguje, vyzerá takto:



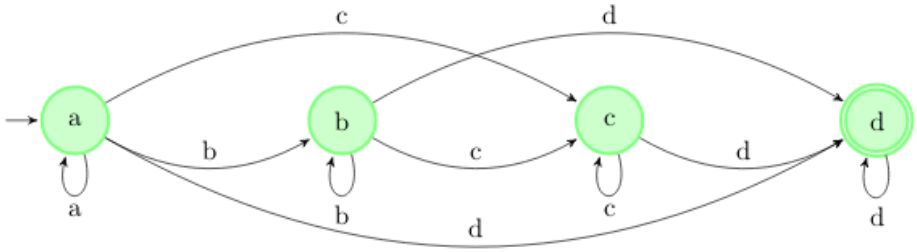
d) (3 body) Ak by sa podreťazec skladal len zo znakov, ktoré sa neopakujú (napr. "abdc"), bola by táto úloha jednoduchšia. Riešením by bolo cykliť sa v počiatočnom stave a pri reťazci, ktorý obsahuje daný podreťazec by sme len išli do ďalšieho stavu vždy po jednom znaku.



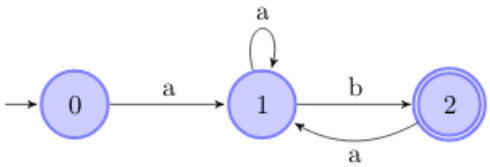
Keďže v podreťazci sa 2 krát nachádza znak “a”, pri prechádzaní receptu potrebujeme rátať s tým, že dané a je už začiatok iného podreťazca, ale tiež aj s tým že predošlý podreťazec môže ešte skončiť (napr. “ababacac”). Riešenie teda musí obsahovať prechody tak, aby rátalo s možnosťou nového reťazca. Správne riešenie teda vyzerá takto:



e) (3 body) V tejto úlohe si v každom stave budeme pamätať, aký najnižši znak doteraz bol. Pri znaku s vyššou hodnotou prejdeme prechodom do ďalšieho stavu.



f) (3 body) Automat je postavený ako 2 rovnaké časti za sebou, teda časť {1} a {3} je rovnaká ako {2} a {0} (až na počiatočný stav). Chceme teda spraviť automat ako jedna z tých častí, ale musíme to prepojiť tak, aby sme akceptovali aj prázdny recept (pretože aj pôvodný automat ho akceptuje). Riešenie teda vyzerá takto:



3. A tak sa Denis stratil

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

V tejto úlohe sme mali možnosť stretnúť sa so základnými Linuxovými príkazmi, ktoré sa vám v budúcnosti budú hodiť. Táto úloha bola špeciálna v tom, že ste sa k správnemu heslu mohli dostať viacerými spôsobmi, pričom bolo odporúčané hľadať pomoc na Googli. Dúfame, že sa vám hľadanie hesiel páčilo :)

Predtým, než môžeme začať riešiť jednotlivé podúlohy sa najprv musíme pripojiť na vzdialený počítač, kde boli heslá ukryté. Ak ste postupovali podľa návodu v zadaní, dostali ste sa do príkazového riadku (konzoly) kde ste vedeli písať príkazy.

Riešenia jednotlivých podúloh

1. Ak ste skúsili zadať príkaz `ls` zo zadania, tak ste zistili, že sa v priečinku nachádza súbor `level1.txt` *(a priečinky pomenované* `levelX`*, v ktorých sa nachádzali zvyšné podúlohy)*. Súbor vieme otvoriť príkazom `cat`, teda zadáme príkaz `cat level1.txt` a tešíme sa z prvého hesla.

2. V tomto leveli sa zoznámime s priečinkami. Do priečinku vojdeme príkazom `cd`. Príkazom `ls` zobrazíme obsah priečinku. Uvidíme, že je v ňom ďalší priečinok s názvom `tubudeheslo`. Zodáme `cd tubudeheslo`. Keď opäť spravíme `ls`, nájdeme jeden súbor `hesloklevelu2.txt`. V tomto súbore nájdeme heslo. Naspäť sa vieme dostať opakovaným použitím príkazu `cd ..`, ktorý nás vráti do rodičovkského priečinku.

3. Tu máme celý strom priečinkov, a niekde v nich je heslo. Vieme, že súbor s heslom sa volá `heslo.txt`. Máme viacero možností. Môžeme použiť príkaz `find -name heslo.txt` na nájdenie súboru s názvom `heslo.txt`. Iná možnosť je použiť napr. `ls -R` a rekurzívne vylistovať všetky priečinky.

4. Máme veľký súbor a v ňom má mať práve jeden riadok heslo. Na vyhľadávanie v texte slúži príkaz `grep`. Spravíme teda `grep „HESLO" level4-hesla` a nájde nám to daný riadok. *(Môže sa nám hodiť prepínač `-i`, ktorý spôsobí, že grep bude ignorovať veľkosť písmen. Takže riešenie je aj `grep -i „hEsLo" level4-hesla`)*

5. Vidíme dva súbory, ktoré sú podobné keď si ich otvoríme. Príkaz `ls -al` nám ale ukáže, že jeden je o trocha väčší ako druhý takže nemôžu byť rovnaké. Spravíme `diff level-hesla level5-hesla.old` a nájdeme jeden riadok, ktorý je odlišný, to je naše heslo.

6. Všetci používatelia na Linuxových systémoch sú napísaní v `/etc/passwd`. Prečítame tento súbor a nájdeme hľadaného používateľa.

7. Každý súbor má tri sady privilégii. Čo môže robiť vlastník, čo môže robiť skupina a čo môžu robiť všetci ostatní (a ešte niečo o čom zistíme v ďalšej sérii). Tri možné operácie sú čítanie, zápis a spúšťanie. Napr. `rwxr-x---` znamená, že vlastník môže súbor čítať *(**r**ead)*, meniť *(**w**rite)* aj spúšťať *(e**x**ecute)*. Používatelia zo skupiny ho môžu len čítať alebo spúšťať, ale nie meniť. A hocikto iný nemôže robiť so súborom nič. Oprávnenia získame príkazom `ls -al`. Vidíme, že sme vlastníkom súboru ale že ho nevieme čítať. Oprávnenia zmeníme príkazom `chmod`, napr. môžme spraviť `chmod u+r`, čím pridáme sebe (`user, u`) práva na čítanie (`read, r`).

8. Stačí spustiť program, ktorý nám hneď vypíše heslo. Programy spúšťame pomocou `./`, riešenie je teda `./program`.

9. Pomocou `ls -al` vieme zistiť, ktorému používateľovi patrí súbor `heslo.txt`. Za tohto používateľa sa prihlásime príkazom `su totojemenopouzivatelalevel9`. Potom vieme prečítať heslo.

10. Napr. `echo „Text_vo_vytvorenom_subore" > vytvoreny_subor`. Príkaz `echo` vypíše svoj argument na výstup. Príkaz > je špeciálny znak, ktorým presmerujeme výstup z príkazu do súboru.

11. Najjednoduchšie je použiť príkaz `cp meno_povodneho_suboru meno_noveho_suboru`, čím skopírujeme súbor.

12. Príkazom `rm` vieme zmazať súbor.

13. Príkazom `top` alebo `ps` si pozrieme, aké procesy momentálne bežia. Všimneme si proces so zvlášstnym názvom a ukončíme ho buď príkazmi `kill` alebo `pkill`.

14. Symlinky sú špeciálne typy súborov ktoré nemajú obsah ale len ukazujú na nejaký iný súbor. Keď otvoríme symlinku, je to rovnaké akoby sme otvorili súbor na ktorý symlinka ukazuje. Symlink vytvoríme príkazom `ln -s nazov_suboru nazov_symlinku`.

15. Priečinok je prázdny! Našťastie máme hint, ktorý nám hovorí, že v priečinku je niečo skryté. V linuxe sú skryté súbory tie, ktorých meno začína bodkou. Pomocou `ls -a` vieme zobraziť aj skryté súbory.

4. Sladké náramky

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

Úlohou bolo zostrojiť náramok, ktorého cukríky sa skladajú z cukríkov prvého ale aj druhého Timkinho náramku. Ak sa zamyslíme nad možným počtom takých náramkov, tak zistíme, že ich je strašne veľa. Teda riešenie, ktoré by generovalo všetky možné Romanove náramky by bolo príliš pomalé už aj na prvú sadu. Tadiaľto cesta nepôjde, a musíme vymyslieť niečo iné.

Prvý nástrel

Čo keby sme skúšali zostrojovať hľadaný náramok postupne? Najprv skúsime zistiť koľkokrát môžeme použiť cukrík a, potom b až po posledný typ cukríka z. Kedy môžeme vo výslednom náramku pouiť cukrík x? Iba vtedy, ak sa x nachádza v oboch náramkoch.

Toto pozorovanie stačí na riešenie, ktoré funguje v prvých 3 sadách. Budeme postupne prechádzať znakmi od a po z, a budeme tento znak hľadať v oboch náramkoch. Ak ho v oboch nájdeme, vypíšeme ho a vymažeme ho z oboch náramkov. Ak sa aspoň v jednom z nich nenachádza, tak skúsime ďalší znak. Časová zložitosť tohto riešenia je $O(r \times s)$, kde r, s sú dĺžky náramkov. Zistenie, či sa znak nachádza v oboch náramkoch trvá $O(r+s)$ a hľadať ho budeme najviac $min(r, s)$. Dokopy to je teda $O((r + s) \times min(r, s))$. Ak si ale povieme, že $r > s$, tak namiesto $O(r+s)$ vieme napísať iba $O(r)$ *(lebo s sa v zložitosti schová)* a namiesto $O(min(r, s))$ dostaneme iba $O(s)$ a teda dokopy dostávame $O(r \times s)$. Pamäťová zložitosť je lineárna (a bude lineárna aj v nasledujúcich riešeniach).

(V riešení si vieme pomôcť a namiesto vymazávania, vieme znaky v náramku prepisovať na hodnotu zodpovadajúcu ničomu.)

Listing programu (C++)

```
#include <bits/stdc++.h>
using namespace std;

int main() {
  // tieto prikazy sposobia rychlejsie nacitavanie
  ios_base::sync_with_stdio(false);
  cin.tie(0);

  string prvy, druhy;
  // vsimnite si, ze mozeme nacitavat aj do viac premennych naraz
  cin >> prvy >> druhy;

  string vysledok = "";
  // vsimnime si, ze znaky (typ char) mozeme priradit do typu int
  int znak = 'a';
  while (znak <= 'z') {
    // premenne, ktore budu obsahovat poziciu znaku ktory hladame
    int i, j;

    // skusime najst znak v prvom retazci
    for (i = 0; i < prvy.size(); i++) {
      if (znak == prvy[i]) break;
    }
    // skusime najst znak v druhom retazci
    for (j = 0; j < druhy.size(); j++) {
      if (znak == druhy[j]) break;
    }

    // ak sme znak nenasli v aspon 1 retazci, tak chceme hladat dalsi
    // inak chceme prepisat pozicie kde sme nasli znak, aby sme nehladali
    // donekonecna to iste
    if (i == prvy.size() || j == druhy.size()) {
      znak++;
    } else {
      vysledok += znak;
      prvy[i] = '_';
      druhy[j] = '_';
    }
  }

  if (vysledok.size() == 0) {
    cout << "neda sa" << endl;
  } else {
    cout << vysledok << endl;
  }
}
```

Triedenie

Tento algoritmus by sa dal vylepšiť, ak by sme si cukríky v oboch náramkoch najprv utriedili a potom tento znak vyhľadávali binárne.

Binárne vyhľadávanie funguje tak, že máme utriedenú postupnosť, v ktorej hľadáme nejaký prvok x. Pozrieme sa do **stredu** tejto postupnosti (označme tento prvok y). Ak $x < y$, tak y a všetky hodnoty, ktoré sú v postupnosti od neho napravo nás nezaujímajú (lebo sú ešte väčšie). Naopak, ak $x > y$, tak x a všetky hodnoty naľavo sú príliš malé. Keď zabudneme na zlú polovicu, ostane nám utriedená podpostupnosť polovičnej dĺžky, s ktorou vieme túto úvahu zopakovať. V každom kroku zahodíme polovicu, čo je dosť dobré a má logaritmickú zložitosť.

Ak predpokladáme, že $r > s$ tak vyhľadanie cukríku by nám trvalo namiesto $O(r)$ iba $O(log(r))$. Dokopy je teda zložitosť takého algoritmu $O(r \times log(r))$, pretože treba prvky utriediť a potom v nich logaritmicky vyhľadávať. Takéto riešenie už stačí na plný bodový zisk :)

Listing programu (C++)

```
#include <bits/stdc++.h>
using namespace std;

int main() {
```

```
  ios_base::sync_with_stdio(false);
  cin.tie(0);

  string prvy, druhy;
  cin >> prvy >> druhy;

  // utriedime curiky v naramkoch podla abecedy
  sort(prvy.begin(), prvy.end());
  sort(druhy.begin(), druhy.end());

  string vysledok = "";
  int znak = 'a';
  while (znak <= 'z') {
    // skusime najst znaky v oboch naramkoch. Funkcia lower_bound spravi binarne vyhladavanie
    // a v premennych i,j budu indexy na prvy prvok, ktory je rovnaky alebo vacsi ako znak, ktory
    // hladame alebo index mimo pokial su vsetky prvky mensie
    int i = lower_bound(prvy.begin(), prvy.end(), znak) - prvy.begin();
    int j = lower_bound(druhy.begin(), druhy.end(), znak) - druhy.begin();

    if (i == prvy.size() || prvy[i] != znak || j == druhy.size() || druhy[j] != znak) {
      znak++;
    } else {
      vysledok += znak;
      prvy[i] = '_';
      druhy[j] = '_';
    }
  }

  if (vysledok.size() == 0) {
    cout << "neda sa" << endl;
  } else {
    cout << vysledok << endl;
  }
}
```

Vzorové riešenie

Vzorové riešenie je ale efektívnejšie a aj jednoduchšie na naprogramovanie. Celá idea spočíva v tom, že nepotrebujeme znaky v náramkoch vôbec hľadať. Nás zaujímajú iba ich počty. Ak si spočítame, koľko je každého písmenka v oboch náramkoch, tak potom stačí postupne prechádzať písmenkami a vo výslednom náramku môžeme použiť písmeno x práve $min(P1[x], P2[x])$, kde $P1[x]$ a $P2[x]$ sú počty písmen x v prvom a druhom náramku.

Listing programu (C++)

```
#include <bits/stdc++.h>
using namespace std;

int main() {
  ios_base::sync_with_stdio(false);
  cin.tie(0);

  string prvy, druhy;
  cin >> prvy >> druhy;

  // vyuzijeme, ze znaky a...z maju hodnoty 97...122
  int znakyPrvy[150], znakyDruhy[150];
  // netreba zabudnut nastavit vsetky hodnoty na 0
  for (int i = 0; i < 150; i++) znakyPrvy[i] = znakyDruhy[i] = 0;

  // spocitame pocty znakov
  for (int i = 0; i < prvy.size(); i++) znakyPrvy[prvy[i]]++;
  for (int j = 0; j < druhy.size(); j++) znakyDruhy[druhy[j]]++;

  bool vypisal = false;
  for (int znak = 'a'; znak <= 'z'; znak++) {
    int pocet = min(znakyPrvy[znak], znakyDruhy[znak]);
    if (pocet != 0) {
      // premennu znak musime pretypovat na char, lebo inak by sme
      // vypisali cislo (napr. namiesto znaku 'a' by sa vypisalo 97)
      for (int i = 0; i < pocet; i++) cout << (char) znak;
      vypisal = true;
    }
  }

  if (!vypisal) cout << "neda sa" << endl;
  else cout << endl;
}
```

Riešenie v Pythone založené na rovnakej myšlienke vyžaduje znalosť funkcií `ord` a `chr`.

Listing programu (Python)

```
prvy = str(input())
druhy = str(input())

# vytvorime pole dvojic, pricom prva hodnota z dvojice bude
# pocitat pocet znakov v prvom retazci a druha v druhom
pocty_znakov = [[0] * 2 for pismeno in range(26)]
vypisal = False
```

```
# v pythone, je sa znak neda priradit do cisla a musime
# pouzit funkciu ord, ktora vrati ciselnu hodnotu pismenka
for pismeno in prvy:
  pocty_znakov[ord(pismeno) - ord('a')][0] += 1
for pismeno in druhy:
  pocty_znakov[ord(pismeno) - ord('a')][1] += 1

for index, dvojica in enumerate(pocty_znakov):
  for i in range(min(dvojica[0], dvojica[1])):
    # funkcia chr je opak funkcie ord a vrati z hodnoty znaku znak
    print(chr(index + ord('a')), end='')
    vypisal = True

if vypisal:
  print()
else:
  print('neda sa')
```

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

5. Kamenná cesta

Ak si zoberieme všetky dni, v ktorých vieme prejsť na druhý breh rieky, zadanie od nás chce, aby sme našli posledný (najväčší) deň, v ktorom je to možné. Ako prvé je preto dobré si rozmyslieť, ako veľké takéto číslo môže byť, čo nám ohraničí výsledok.

Existuje pre každý vstup aspoň nejaké riešenie? Kedže pre deň, v ktorom je i-ty kameň ešte použiteľný, k_i platí, že $k_i \geq 1$, tak ľahko vidieť, že v prvý deň vieme určite použiť všetky kamene a dostať sa na druhú stranu. Existuje aj horné ohraničenie? Inými slovami, existuje deň, kedy už nevieme cez rieku v žiadnom prípade prejsť?

Takéto ohraničenie by neexistovalo iba keby nepotrebujeme žiadny kameň na prejdenie rieky, kedže po určitom čase bude každý zaplavený. Kedže vieme preskočiť iba jeden kameň a pre počet kameňov n platí nerovnosť $n \geq 2$ tak máme zaručené, že musíme použiť aspoň jeden kameň na prejdenie rieky. Vieme teda, že rieku určite neprejdeme v deň, kedy už bude zatopený aj posledný kameň. Môžeme si uvedomiť, že tento deň je o jedna väčší ako maximum z hodnôt $k_1, k_2, \ldots k_n$.

Zistili sme, že správna odpoveď sa nachádza niekde medzi hodnotami 1 a $\max(k_1, k_2 \ldots k_n) + 1$. To je super, pretože môžeme postupne vyskúšať všetky tieto hodnoty a nájsť najväčšiu vyhovujúcu.

Riešenie hrubou silou

Povedzme, že máme nejaký konkrétny deň r. Chceme zistiť, či vieme prekročiť rieku iba pomocou doteraz nezatopených kameňov. Teda platí, že kameň j môžeme použiť iba ak $k_j \geq r$. Môžeme si vytvoriť pomocné pole o veľkosti n a doňho si zaznačiť, na ktoré políčka sa vieme dostať z ľavého brehu. Pri vypĺňaní i-teho políčka sa nám stačí pozrieť, či je možné sa dostať buď na políčko $i-1$ alebo $i-2$. Ak áno a platí, že $k_i \geq r$, tak sa vieme dostať v r-tý deň aj na r-té políčko a do pomocného poľa zaznačíme `True`, inak `False`. Na konci vieme z hodnôt na indexoch $n-1$ a n v našom pomocnom poli zistiť, či v daný deň vieme rieku prekročiť. Rieku prekročíme iba ak sa na jedno z týchto políčok vieme dostať.

Listing programu (C++)

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

using namespace std;

bool over(vector<int>& kam, int den)
{
    int n = kam.size();
    // na zaciatku oznacime vsetko ako nedosiahnutelne
    vector<bool> pom(n, false);

    if(kam[0] <= den) pom[0] = true;
    if(kam[1] <= den) pom[1] = true;

    for(int i=2; i<n; ++i) {
        if(kam[i] >= den && (pom[i-1] || pom[i-2])) {
            pom[i] = true;
        }
    }
    if (pom[n-1] == true || pom[n-2] == true) {
        return true;
    }
    else {
        return false;
    }
}

int main()
{
```

```
    int n; cin >> n;

    vector<int> kamene(n);
    for(int i=0; i<n; ++i) {
        cin >> kamene[i];
    }

    int maxim = kamene[0];
    for(int i=1; i<n; ++i) {
        maxim = max(maxim, kamene[i]);
    }

    for(int i=maxim; i>=1; --i) {
        if(over(kamene, i)) {
            cout << i << endl;
            break;
        }
    }
    return 0;
}
```

V pythone si ukážeme mierne inú implementáciu a to bez pomocného poľa. Je založená na pozorovaní, že z nášho pomocného poľa nám vždy stačí poznať iba posledné dve vypočítané hodnoty (pozície $i-2$ a $i-1$).

Listing programu (Python)

```
def over(kam, den):
    pred, pred2 = True, True

    for i in range(len(kam)):
        if (pred == False and pred2 == False):
            break

        pred2 = pred
        pred = False

        if (kam[i] >= den):
            pred = True

    return (pred or pred2)


n = int(input())
kamene = [int(_) for _ in input().split()]

maxim = max(kamene)

for i in range(maxim, 0, -1):
    if over(kamene, i):
        print(i)
        break
```

Toto riešenie je správne, ale má časovú zložitosť závislú od maximálneho k_i, ktoré môže byť oveľa väčšie ako n. Je teda pomalé, ale stačí na riešenie prvej a tretej sady.

Zlepšenie riešenia hrubou silou

Riešenie hrubou silou ide v skutočnosti vylepšiť ďalším pozorovaním. Samotné overovanie nejakého konkrétneho riešenia r nezrýchlime. To čo sme robili doteraz je, že sme overili všetky hodnoty medzi 1 až $\max(k_0, k_1, \ldots k_n)$ a vybrali najväčšie riešenie. Myšlienkou zrýchlenia je, že pokiaľ ide prejsť rieku v i-ty deň tak je to možné aj v každý skorší deň. Prečo? Pokiaľ sme v i-ty deň použili nejaké kamene, pomocou ktorých sme vedeli prejsť na druhú stranu rieky. Vieme presne tie isté kamene použiť aj v hociktorý skorší deň. To platí aj opačne, ak nevieme v i-ty deň prejsť rieku, nepôjde to ani neskôr. To preto, že neskôr bude zatopených už len viac kameňov. Toto nám umožnuje použiť techniku známu ako binárne vyhľadanie na nájdenie nášho riešenia.

Pointa binárneho vyhľadávania je, že namiesto overenia hodnôt 1 až $\max(k_0, k_1, \ldots k_n)$ nám postačuje overiť riešenie pre číslo, ktoré je v strede, medzi 1 a $\max(k_0, k_1, \ldots k_n)$. Označme si ho s. Pokiaľ platí, že v s-tý deň sa dá prejsť cez rieku tak vieme, že naše riešenie bude väčšie alebo rovné ako s. Pokiaľ v s-tý deň nevieme rieku prejsť, riešenie bude menšie ako s. V oboch prípadoch sa nám počet čísel, ktoré môžu byť riešením zmenšil približne o polovicu. Tento postup vieme ďalej opakovať na novo nájdené ohraničenia našeho riešenia až kým nám neostane jedno číslo, ktoré je výsledkom.

Listing programu (C++)

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

using namespace std;

// na overenie mozeme pouzit funkciu z predchadzajuceho riesenia
bool over(vector<int>& kam, int den);

int main()
```

```
{
    int n; cin >> n;

    vector<int> kamene(n);
    for(int i=0; i<n; ++i) {
        cin >> kamene[i];
    }

    int maxim = kamene[0];
    for(int i=1; i<n; ++i) {
        maxim = max(maxim, kamene[i]);
    }

    int l = 0, h = maxim+1;
    while (l+1 < h) {
        int stred = (l + h) / 2;

        if (over(kamene, stred)) {
            l = stred;
        }
        else {
            h = stred;
        }
    }

    cout << l << endl;

    return 0;
}
```

Iné riešenia

V predchádzajúcich riešeniach nám vadilo, že naše riešenia boli závislé od maximálneho k_i. Poďme skúsiť nájsť riešenie, ktoré je závislé len od n. Urobíme to tak, že sa pozrieme bližšie na samotnú hodnotu najlepšieho riešenia.

Vieme, že najlepšie riešenie leží medzi 1 a $\max(k_1, k_2, \dots k_n)$. Tvrdíme však, že toto riešenie musí byť rovná niektorému z k_i. Viete prečo? Povedzme si, že by nebolo. Potom ale určite existujú kamene, po ktorých vieme preskákať v tento deň a jeden z nich, označme si ho j, má najnižšiu hodnotu k_j. Naše riešenie nemôže byť najlepšie pretože od tohto riešenia existuje lepšie riešenie rovné k_j. To používa kamene z nášho predchádzajúceho riešenia a pritom je určite väčšie. Po tejto úvahe už vieme, že riešenie bude určite rovné niektorému z hodnôt $k_1, k_2, \dots k_n$. To znamená, že nám stačí overiť n hodnôt, každú v čase $O(n)$, čo nám prinesie riešenie s časovou zložitosťou $O(n^2)$.

Vzorové riešenie

Síce na vyriešenie úlohy na plný počet bodov stačilo aj riešenie z časti `Zlepšenie riešenia hrubou silou`, my si teraz ukážeme jedno ešte rýchlejšie a elegantnejšie riešenie. Jeho myšlienka je nasledujúca.

Kedy určite rieku nevieme prejsť? Predsa keď budú zatopené dva za sebou ležiace kamene! Takto veľkú medzeru totiž nevieme preskočiť a ostaneme zaseknutý. Dvojica kameňov i a $i+1$ bude zatopená keď sa zatopí ten neskorší, teda v čase $\max(k_i, k_{i+1}) + 1$. Zo všetkých dvojíc však chceme zobrať tú najskoršie zatopenú, prvá dvojica kameňov bude teda zatopená v čase $\min(\max(k_1, k_2) + 1, \max(k_2, k_3) + 1, \dots \max(k_{n-1}, k_n) + 1)$.

My si však musíme odpovedať aj na dôležitejšiu otázku a to, či sa dá prejsť na druhú stranu rieky vždy, keď ešte **neexistuje** dvojica za sebou idúcich zatopených kameňov. Ak totiž platí takéto tvrdenie, znamená to, že vyššie určená hodnota je správnou odpoveďou.

Predstavme si, že žiadne dva za sebou idúce kamene nie sú zatopené. Je jasné, že z ľavého brehu máme kam skočiť. V tom momente sa nachádzame na nezatopenom kameni. Ak je nasledujúci kameň nezatopený, jednoducho naň prejdeme a riešime problém ďalej. Ak však zatopený je, ten ďalší zatopený byť nemôže (lebo to by boli už dva za sebou idúce zatopené kamene). Preskočíme teda jeden zatopený kameň a pokračujeme rovnakou úvahou ďalej.

Práve sme dokázali, že v deň $\min(\max(k_1, k_2), \max(k_2, k_3), \dots \max(k_{n-1}, k_n)) + 1$ sa cez rieku určite nedostaneme, ale v ľubovoľný skorší deň to ešte pôjde, toto číslo mínus jeden je teda odpoveďou.

Listing programu (C++)

```
#include <iostream>
#include <vector>
#include <algorithm> // min(), max()

using namespace std;

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

    vector<int> kamene(n);
    for(int i=0; i<n; ++i) {
```

```
        cin >> kamene[i];
    }

    int odp = max(kamene[0], kamene[1]);
    for(int i=1; i<n-1; ++i) {
        odp = min(odp, max(kamene[i], kamene[i+1]));
    }

    cout << odp << endl;
    return 0;
}
```

Listing programu (Python)

```
n = int(input())
kamene = [int(_) for _ in input().split()]

odpoved = max(kamene[0], kamene[1])
for i in range(1, n-1):
    odpoved = min(odpoved, max(kamene[i], kamene[i+1]))
print(odpoved)
```