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

Vzorové riešenia 2. kola letnej časti

Chýba súbor vzor-prikl1.tex!!!

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

2. Rôzne Merlinove problémy

Podúloha a.

Pri sčítavaní čísel nezáleží, v akom poradí sú tieto čísla: $a+b=b+a$ (hovoríme, že sčítanie je komutatívna operácia). Preto keď v kúzle preusporiadame písmená, nezmení sa ich súčet. Príklady rôznych kúzel s rovnakým súčtom sú `ahoj`, `hoja`, `joha`, ale aj `ahoj` a `bgoj`.

S magickou hodnotou je to inak. Nedajú sa nájsť dve rôzne kúzla, ktoré by mali rovnakú magickú hodnotu. Predstavme si, čo by sa stalo, keby takéto kúzla existovali. Nájdime prvú pozíciu od konca, na ktorej sa písmena kúzel líšia. Označme si pozíciu týchto písmen od konca k. Rozdelme si magickú hodnotu kúzel na tri časti

- A = tá, do ktorej sa započítavajú písmena po pozícii k.
- B = tá, do ktorej sa započítava písmeno na pozícii k.
- C = tá, do ktorej sa započítavajú písmena pred pozíciou k.

Keď odčítame magické hodnoty rôznych kúzel, tak časti C sa vynulujú (lebo boli rovnaké). Časti B boli rôzne, takže ich rozdiel nie je nula, ale zároveň je v absolútnej hodnote menší ako 47^{k+1} (lebo je to najviac $25 \cdot 47^k$). Časti A sú v oboch kúzlach násobky čísla 47^{k+1} a keď ich odčítame, dostaneme tiež násobok čísla 47^{k+1}. Celkový rozdiel teda nemôže byť násobokom čísla 47^{k+1} a preto to nemôže byť nula.
Keďže rozdiel magických čísel rôznych kúzel nemôže byť nula, tak tieto magické čísla nemôžu byť rovnaké.

Ako teda môže Merlin rýchlo zisťovať, či sú dve kúzla vo *Veľkom zvitku* rovnaké? Pri otázke "Je i-te a j-te kúzlo vo veľko zvitku rovnaké?" stačí, aby namiesto pracného porovnávania kúzel porovnal ich magické hodnoty. Tie nájde na i-tom a j-tom riadku *Malého zvitku*. Už totiž vieme, že dve kúzla sú rovnaké vtedy a len vtedy, keď majú rovnaké magické hodnoty.
Fakt, že v *Malom zvitku* sú hodnoty *predpočítané*, je dôležitý. Inak by pri každej otázke Merlin počítal magickú hodnotu pre obe kúzla znovu. Musel by tak prejsť všetky znaky oboch kúzel.

Podúloha b.

To, čo chceme robiť, je postupne porovnať súčet hľadanej vzorky so súčtom každého 1000 znakového reťazca v knihe. Otázkou je len, ako efektívne zistiť všetky tieto súčty.
Súšet prvých 1000 čísel zistíme tak, že čísla jednoducho sčítame. Každý ďalší súčet však budeme vedieť zistiť na dve matematické operácie z predošlého súčtu tak, že odčítame prvé číslo a pripočítame jedno ďalšie číslo.
Napísané matematicky (T je reťazec obsahujúci Pendragonovu knihu, $T[i]$ je pozícia i-teho znaku v abecede a $t_i \dots t_j$ je podreťazec T obsahujúci znaky od i po j, vrátane):

$$T[i]+T[i+1]+\cdots+T[i+999]-T[i]+T[i+1000]=T[i+1]+T[i+2]+\cdots+T[i+1000]$$

Alebo alternatívne:

$$S(t_i \dots t_{i+999})-T[i]+T[i+1000])=S(t_{i+1} \dots t_{i+1000})$$

Podúloha c.

To, čo chceme robiť, je postupne porovnať magické číslo vzorky R s magickým číslom každého súvislého úseku v K. Otázka je len, ako efektívne zistiť magické číslo všetkých týchto úsekov.
Najprv si spočítame mocniny čísla 47 od 1 až po 47^r. Potom spočítame magické číslo R a prvých r písmen K.

Každé ďalšie magické číslo dostaneme tak, že predošlé číslo vynásobíme 47, odpočítame 47^r-krát hodnota prvého písmena a pripočítame hodnotu ďalšieho písmena v poradí.

Napísané matematicky:

$$47(47^{r-1}K_i + 47^{r-2}K_{i+1} + \cdots + K_{i+r-1}) - 47^r K_i + K_{i+r} = 47^{r-1}K_{i+1} + 47^{r-2}K_{i+2} + \cdots + K_{i+r}$$

Alebo alternatívne:

$$47 \cdot M(k_i \ldots k_{i+r-1}) - 47^r K_i + K_{i+r} = M(k_{i+1} \ldots k_{i+r})$$

Tu je program, ktorý načíta knihu K a reťazec R a vypíše všetky výskyty R v K.

Listing programu (Python)

```
K, R = input(), input()
k, r = len(K), len(R)

p = 47**r

def c(x):
    return ord(x)-ord('a')+1

magic_R, magic_K = 0, 0
for i in range(r):
    magic_R = magic_R*47 + c(R[i])
    magic_K = magic_K*47 + c(K[i])

for i in range(k-r+1):
    if magic_R == magic_K:
        print("vyskyt na pozicii", i+1)

    if i+r < k:
        magic_K = magic_K*47 - c(K[i])*p + c(K[i+r])
```

Všimnite si, že toto riešenie funguje aj ako riešenie podúlohy b.

Podúloha d.

Aby sme sa vyhli pracnému a pomalému porovnávaniu reťazcov znak po znaku, budeme opäť porovnávať len ich magické hodnoty. V tomto prípade teda potrebujeme rýchlo spočítať magické hodnoty každého začiatku a každého konca.

Sme si istí, že po prečítaní riešenia predošlej podúlohy by ste to hravo zvládli spraviť, preto si ukážeme trochu iné riešenie.

Vypočítame si najprv magické hodnoty len pre všetky začiatky reťazca T. Prvý začiatok je len jeden znak a má magickú hodnotu ako hodnota tohto znaku:

$$Z(1) = T[0]$$

Všimnite si, že Z indexujeme od 1. Magická hodnota každého ďalšieho začiatku je len hodnota ďalšieho znaku plus 47-krát predošlá magická hodnota:

$$Z(i) = T[i-1] + 47 \cdot Z(i-1)$$

Rozmyslite si, prečo to tak je. Magické hodnoty koncov dokážeme s konštantným množstvom operácií zistiť čisto z magických hodnôt začiatkov. Označme si dĺžku celého kúzla n. Potom Magická hodnota posledných i písmen je nasledovná:

$$K(i) = (Z(n) - 47^i \cdot Z(n-i)))$$

Takto by vyzeral program, ktorý načíta kúzlo zo vstupu a vypíše, koľko potrebuje kúzlo many.

Listing programu (Python)

```
S = input()
n = len(S)

def c(x):
    return ord(x)-ord('a')+1

Z = [0]
m47 = [1]

for i in range(n):
    m47.append(47*m47[-1])
    Z.append(c(S[i]) + 47*Z[-1])
```

```
mana = 0
for i in range(1, n+1):
    if Z[i] == Z[n] - m47[i] * Z[n-i]:
        mana += 1

print(mana)
```

Použitie v praxi

Na čo nám je toto celé užitočné a ako sa podobný princíp používa v praxi?

Ak by sme programy napísané vyššie spustili, zistili by sme, že nebežia oveľa rýchlejšie ako programy, ktoré by porovnávali reťazce znak po znaku. Totiž Python, narozdiel od čarodejníkov, nevie porovnávať veľké čísla konštantne rýchlo.

Magické číslo 10000 znakového reťazca bude mať vyše 16000 cifier a porovnať takéto čísla počítaču niečo trvá. Aby sme sa vyhli veľým číslam, budeme namiesto veľkých čísel používať len ich zvyšok po delení nejakým prvočíslom P. Tento zvyšok je menší ako P. Ak teda napríklad zvolíme $P = 1000000009$, čísla budú mať len 10 a menej cifier.

Pozrite si tieto dva programy:

Listing programu (Python)

```
POW = 47
MOD = 1000000009

K, R = input(), input()
k, r = len(K), len(R)

p = 1
for i in range(r):
    p = (p*POW) % MOD

def c(x):
    return ord(x)-ord('a')+1

magic_R, magic_K = 0, 0
for i in range(r):
    magic_R = (magic_R*POW + c(R[i]))%MOD
    magic_K = (magic_K*POW + c(K[i]))%MOD

for i in range(k-r+1):
    if magic_R == magic_K:
        print("vyskyt na pozicii", i+1)

    if i+r < k:
        magic_K = (magic_K*POW - c(K[i])*p + c(K[i+r]))%MOD
```

Listing programu (Python)

```
POW = 47
MOD = 1000000009

S = input()
n = len(S)

def c(x):
    return ord(x)-ord('a')+1

Z = [0]
mpow = [1]

for i in range(n):
    mpow.append(POW*mpow[-1] % MOD)
    Z.append((c(S[i]) + POW*Z[-1]) % MOD)

mana = 0
for i in range(1, n+1):
    if (Z[i] * mpow[n-i]) % MOD == (Z[n] - Z[n-i]) % MOD:
        mana += 1

print(mana)
```

Ide o prepis riešení z podúloh c a d s tým, že všetky medzivýsledky modulujeme číslom `MOD`. Číslo 47 bolo nahradené všeobecnou premennou `POW`, lebo čo nám bráni použiť iné číslo, však?

Dôsledky toho, že namiesto magického čísla použijeme zvyšok tohto čísla po delení číslom MOD sú dva:

- Celý program je teraz veľmi rýchly (spraví len toľko operácií ako je dĺžka vstupného reťazca, krát nejaká konštanta).
- Občas vypíšeme zlú odpoveď. Môže sa totiž stať, že dve rôzne magické čísla majú rovnaký zvyšok.

Pri vhodnej voľbe čísel `POW` a `MOD` (obe by mali byť prvočísla, hoci to samo o sebe vždy nestačí) bude šanca omylu malá, nebudeme však zachádzať do hlbších detailov, lebo by sme narazili na skutočnú mágiu[1].

Algoritmus, ktorý hľadá vzorku v texte pomocou tejto techniky sa volá Rabinov-Karpov algoritmus.

3. Akútny upgrade

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

Pred prečítaním tohto vzorového riešenia by ste mali mať isté základy práce v tomto modeli. Veľmi vám môže pomôcť riešenie predchádzajúcej úlohy či jej vzorové riešenie.

Podúloha A

V prípade, že ste poznali alebo vedeli vymyslieť cyklus bola táto úloha pomerne priamočiara. Všetko čo umocňovanie predstavuje je iba viacnásobné násobenie. Stačilo teda použiť jeden pomocný register. Doň vložiť jednotku a potom b-krát vynásobiť a-čkom.

```
get a; get b
put 1; get v

label cyklus
jeq i b koniec
put v; put a; mul; get v
put i; put 1; add; get i
jump cyklus

label koniec
put v; print; stop
```

Podúloha B

Najväčší spoločný deliteľ dvoch čísel je najväčšie číslo, ktoré delí prvé aj druhé číslo. Jeden spôsob ako ho nájsť je prejsť všetky čísla od 1 do $minimum(a, b)$. Toto riešenie však nemuselo prejsť do potrebného počtu krokov. To, čo sa dá v tomto prípade použiť je Euklidov algoritmus. Pri finálnej implementácií stačí v podstate prepísať pseudokód z wikipédie do nášho PRAStarého Kalkulátora.

```
label cyklus
get u; get v
jz v koniec
put u; put v; mod; get p;
put v; put p;
jump cyklus

label koniec
put u; print
```

Podúloha C

Pozornému oku neujde, že čísla na vstupe sú kladné. To pre nás znamená, že môžeme použiť nulu ako zarážku. V prvej fáze prejdeme všetky čísla na vstupe a za každé väčšie ako x, si za nulu pridáme jednu jednotku. Potom je našou úlohou už iba sčítať všetky jednotky, ktoré ostali v rúre. To je úloha, ktorú už z minulého vzorového riešenia dobre poznáme.

```
get x;
put 0;

label cyklus
get i;
jz i dalej
jgt i x pridaj
jump cyklus
```

[1] vysokoškolskú algerbu ;)

```
label pridaj
put 1;
jump cyklus

label dalej

label scitaj
get a;
jempty koniec
put a; add
jump scitaj

label koniec
put a
print
```

Podúloha D

Riešenie tejto podúlohy sa jemne podobná na riešenie predchádzajúcej podúlohy. Na začiatku si potrebujeme do registra vložiť jednotku. To preto aby sme mali s čím jednotlivé čísla porovnávať. To vieme urobiť napr. tak, že si vložíme do rúry nulu a jednotku. Preiterujeme všetky kladné čísla na vstupe, pokým zas nenarazíme na nulu. Potom si napr. do registra `j` vložíme jednotku, ktorá ostala na začiatku rúty. Potom prejdeme ešte raz všetky čísla na vstupe a za nulu si vkladáme jednu jednotku za každé číslo na vstupe, ktoré nie je jedna až pokým na jednotku nenarazíme. Potom dočítame čísla, ktoré ostali až pokým znovu nenarazíme na nulu. Potom už máme za nulou iba jednotku za každé číslo, ktoré nebolo jednotka, na ktoré sme narazili pred samotnou prvou jednotkou. Tieto čísla v rúre už stačí len sčítať s čím už máme prax z predchádzajúcej podúlohy. K tomuto súčtu potom nesmieme zabudnúť pripočítať jednotku.

```
put 0; put 1

label cyklus1
get i
jz i dalej1
put i
jump cyklus1

label dalej1
get j

put 0

label cyklusnajdi
get i
jeq i j cyklusnasiel
put 1
jump cyklusnajdi

label cyklusnasiel
get i;
jz i scitaj
jump cyklusnasiel

label scitaj
get a;
jempty koniec
put a; add
jump scitaj
```

```
label koniec
put a; put 1; add; print
```

Podúloha E

Táto úloha vám mala ukázať, že pomocou PRAStarého Kalkulátora ide robiť aj o čosi zložitejšie veci, ktoré možno poznáte aj zo sveta reálneho programovania. Kedže čísel na vstupe nie je veľa, stačilo si vybrať niektorý z pomalších ale jednoduchších triediacich algoritmov a naimplementovať ho. My sme si vybrali insertion sort. To je triediaci algoritmus, ktorý postupne berie prvky z poľa či listu a vkladá ich do usporiadanej postupnosti. Na začiatku v usporiadanej postupnosti nie je nič. Potom algoritmus postupuje tak, že vezme prvé číslo z neusporiadanej postupnosti a vloží ho na správne miesto do tej usporiadanej. Toto opakuje, pokým neusporiadana postupnosť nie je prázdna. V PRAStarom Kalkulátore bude náš algoritmus vyzerať podobne. V rúre sa budú nachádzať dve postupnosti oddelené 0. Vždy načítame prvé číslo z prvej postupnosti, doiterujeme cez prvú postupnosť, potom prechádzame druhú pričom hľadáme číslu správne miesto. Keď prvý krát stretneme v usporiadanej postupnosti prvok väčší ako naše umiestňované číslo, vložíme najskôr naše číslo a potom pokračujeme ďalej dovkladaním ostatku prvkov z usporiadanej postupnosti. Tento postup opakujeme, pokým nenarazíme pri hľadaní ďalšieho čísla na utriedenie na nulu.

```
get x; put 0; put x; put 0

label vyberdalsie
get v;
jz v usporiadane
jump prejdi1

label prejdi1
get i;
jz i vlozz
put i
jump prejdi1

label vlozz
put 0
label vloz
get i
jz i specialnevloz2
jgt i v specialnevloz1
put i
jump vloz

label specialnevloz1
put v; put i; jump prejdi2

label specialnevloz2
put v; jump dalej

label prejdi2
get i;
jz i dalej
put i
jump prejdi2

label dalej
put 0
jump vyberdalsie

label usporiadane
jump zbavnuly
```

```
label zbavnuly
get i
jz i vypis
put i
jump zbavnuly

label vypis
jempty koniec
print
jump vypis

label koniec
stop
```

Zdroj

Táto úloha bola pôvodne teoretickým modelom v Olympiáde v Informatike. Pokiaľ Ťa riešenie úloh ako sú v Prasku baví, odporúčame ti riešiť kategóriu B Olympiády v Informatike.

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

4. Sekanie bambusu

V tejto úlohe sme dostali n čísel. V každom momente sme mohli zobrať buď číslo zľava, alebo sprava. Našou úlohou bolo vytvoriť čo najdlhšiu rastúcu postupnosť.

Rýchlejšie riešenie nie je vždy zložitejšie

Najskôr sa pozrieme na možné riešenia hrubou silou. Nezľaknite sa však toho, že niektoré časti možno nebudú úplne zrozumiteľné hneď na prvý pohľad. Toto je totiž úloha, kde je vzorové riešenie jednoduchšie, ako riešenie hrubou silou.

Hrubá sila

Chceli by sme asi nejak vyskúšať všetky možnosti. Predstavme si, že sme už nejaké čísla zobrali a chceme zobrať ďalšie. Nachádzame sa teda v nejakom stave. Ten vieme popísať troma číslami. Tie budú hovoriť, kde je aktuálne ľavý okraj, kde je aktuálne pravý okraj a aké bolo posledné číslo, ktoré sme zobrali. Nazvime tieto čísla `lavy`, `pravy` a `posledne`. Zo stavu `(lavy, pravy, posledne)` sa vieme dostať do stavov `(lavy + 1, pravy, b[lavy])` a `(lavy, pravy - 1, b[pravy])`. Samozrejme, do prvého z uvedených stavov sa vieme dostať iba ak `b[lavy]` $>$ `posledne` a do druhého sa vieme dostať iba ak `b[pravy]` $>$ `posledne`.

Toto nám už stačí na napísanie rekurzívnej funkcie, ktorá bude postupne skúšať všetky validné postupnosti výberov strán, z ktorých berieme. Vždy, keď nájde nejakú postupnosť, ktorá je dlhšia, ako doteraz najdlhšia nájdená, zapamätá si ju ako novú doteraz najdlhšiu nájdenú. Na konci teda stačí vypísať túto najdlhšiu nájdenú postupnosť.

Výsledok teda získame zavolaním funkcie do stavu `(lavy = 0, pravy = n - 1, posledne = -1)`. Ako parameter `posledne` tu samozrejme môžeme použiť akékoľvek číslo menšie, ako je najkratšia možná dĺžka bambusu.

Časová zložitosť takéhoto riešenia vyzerá ako $O(2^n)$, pretože v každom rekurzívnom volaní sa zavoláme do dvoch ďalších. Ak ale vhodne spravíme podmienky v našej funkcii, a budeme sa správne rozhodovať, vieme z toho spraviť $O(n^3)$, či dokonca aj riešenie so zložitosťou $O(n)$. To, ako sa správne rozhodovať si ukážeme v nasledujúcich sekciách vzoráku. Pamäťová zložitosť bude $O(n)$.

Takéto rekurzívne riešenie však bude padať na tom, že rekurzia pôjde veľmi hlboko a minie sa nám pamäť.

Listing programu (C++)

```
#include <bits/stdc++.h>

using namespace std;

int n;
vector<int> v;
string ans, now;

void update_ans() {
    if(now.size() > ans.size())
        ans = now;
}
```

```
void backtrack(int left, int right, int last_used) {
    if(left > right)
        return;
    if(v[left] <= last_used && v[right] <= last_used)
        return;
    if(v[left] > last_used) {
        now.push_back('L');
        update_ans();
        backtrack(left + 1, right, v[left]);
        now.pop_back();
    }
    if(v[right] > last_used) {
        now.push_back('R');
        update_ans();
        backtrack(left, right - 1, v[right]);
        now.pop_back();
    }
}

int main() {
    cin >> n;
    v.resize(n);
    for(int i = 0; i < n; i++)
        cin >> v[i];

    backtrack(0, n - 1, -1);
    cout << ans << '\n';

    return 0;
}
```

Listing programu (Python)

```
b = []
ans, now = '', ''


def update_ans():
        global ans, now
        if len(now) > len(ans):
                ans = now


def backtrack(left, right, last_used):
        global ans, now
        if left > right:
                return
        if b[left] <= last_used and b[right] <= last_used:
                return
        if b[left] > last_used:
                now += 'L'
                update_ans()
                backtrack(left + 1, right, b[left])
                now = now[:-1]
        if b[right] > last_used:
                now += 'R'
                update_ans()
                backtrack(left, right - 1, b[right])
                now = now[:-1]


n = int(input())
b = [int(x) for x in input().split()]
backtrack(0, n - 1, -1)
print(ans)
```

Stále hrubá sila

Možno jednoduchšou možnosťou, ako spraviť riešenie hrubou silou, je postupne skúsiť všetky možnosti brania zľava a sprava bez rekurzie. Na to vieme použiť niečo, čo sa volá bitmask. Čísla sú v počítači reprezentované v dvojkovej sústave. Sú tam teda iba nuly a jednotky. V n-bitovom čísle je dokopy n núl a jednotiek. Predstavme si, že `0` bude reprezentovať výber čísla zľava a `1` bude reprezentovať výber čísla sprava. Potom napríklad číslo `105`, ktoré je v dvojkovej sústave `01101001`, bude znamenať náš postup výberov strany `LRRLRLLR`.

Pozrime sa na všetky bitmasky dĺžky 3. Budú to teda čísla $0, 1, 2, ...2^3 - 1$, teda čísla od 0 po 7 vrátane. Tieto čísla v dvojkovej sústave postupne zapíšeme ako `000`, `001`, `010`, `011`, `100`, `101`, `110`, `111`. Všimnime si, že naozaj žiadna iná kombinácia núl a jednotiek dĺžky 3 neexistuje.

Ak teda vygenerujeme postupne všetky bitmasky dĺžky n a pre každý overíme, akú najdlhšiu rastúcu postupnosť nám takýto bitmask prinesie, tak určite nájdeme nejaké správne riešenie. To je preto, že skúšame všetky možné postupy.

Naprogramovať niečo takéto je celkom jednoduché. Ako bitmasky budeme používať obyčajné čísla od 0 do $2^n - 1$. Na to sa vám bude hodiť poznať nejaké bitové operácie, no o nich sa tu nejdem rozpisovať a určite si o nich niečo zvládnete naštudovať aj sami.

Časová zložitosť je $O(2^n \cdot n)$, pretože skúšame 2^n bitmaskov a pre každý zistíme, akú dlhú rastúcu postupnosť vytvorí v $O(n)$. Pamäťová zložitosť bude $O(n)$.

Takéto generovanie bitmaskov je použiteľné vo veľmi veľa úlohách na napísanie riešenia hrubou silou. Určite sa teda oplatí ho naučiť používať, pretože vám môže priniesť aspoň čiastočné body v úlohách, ktoré neviete vyriešiť optimálne.

Listing programu (C++)

```
#include <bits/stdc++.h>

using namespace std;

int n;
vector<int> v;

int main() {
    cin >> n;
    v.resize(n);
    for(int i = 0; i < n; i++)
        cin >> v[i];

    string ans;
    // Ideme cez bitmasky od 0 po 2^n - 1
    for(int mask = 0; mask < (1 << n); mask++) {
        int last = -1, l = 0, r = n - 1;
        string now;
        for(int j = 0; j < n; j++) {
            if(mask & (1 << j)) {
                if(v[r] <= last)
                    break;
                last = v[r];
                r--;
                now.push_back('R');
            }
            else {
                if(v[l] <= last)
                    break;
                last = v[l];
                l++;
                now.push_back('L');
            }
        }
        if(now.size() > ans.size())
            ans = now;
    }
    cout << ans << '\n';

    return 0;
}
```

Variant A

V tejto variante boli všetky výšky bambusov rôzne.

Riešenie variantu `A` je veľmi jednoduché. Vždy chceme zobrať najmenšie číslo, ktoré je väčšie, ako posledné doteraz zobrané.

Prečo to funguje? Ak sú aj ľavé aj pravé číslo menšie, ako posledné zobrané, tak sme skončili. Ak je iba jedno z nich väčšie, ako posledné, tak nemáme moc na výber a zoberieme ho, lebo je to lepšie ako skončiť a nezobrať nič. Čo ak sú obe väčšie, ako posledné zobrané? Ak by sme zobrali väčšie z nich, tak to menšie už nikdy nezoberieme. Dostaneme teda postupnosť nejakej dĺžky d končiacu na nejaké číslo c. Ak zoberieme menšie z čísel, tak stále budeme môcť zobrať aj to väčšie. Určite tak teda dokážeme vytvoriť postupnosť dĺžky aspoň $d+1$ končiacu na c, čo je lepšie, ako prvá možnosť, lebo je dlhšia a končí na rovnaké číslo.

Časová zložitosť tohto riešenia je $O(n)$, čo je zjavne optimálne, pretože rovnako dlho nám trvá samotné načítanie vstupu. Pamäťová zložitosť bude tiež $O(n)$.

Variant B

V tejto variante už mohli byť niektoré výšky bambusov rovnaké.

Od riešenia variantu A k riešeniu variantu B už nechýba veľa. Postupujeme rovnako, ako vo variante A. Jediná vec, ktorá sa nám doteraz nemohla stať a teraz už môže je, že aj naľavo aj napravo budú rovnaké čísla. Tu si však stačí uvedomiť, že ak zoberieme jedno z nich, tak už nikdy určite nebudeme môcť zobrať to druhé. To znamená, že od momentu, keď sa nám stane, že aj vľavo aj vpravo sú rovnaké čísla, máme iba dve možnosti. Buď zoberieme čo najviac čísel iba zľava, alebo zoberieme čo najviac čísel iba sprava. Ako ale máme vedieť, ktorá z týchto možností je lepšia? Jednoducho skúsime obe a zoberieme tú z nich, ktorá nám pridá viac čísel do postupnosti.

Časová aj pamäťová zložitosť budú stále $O(n)$.

Listing programu (C++)

```cpp
#include <bits/stdc++.h>

using namespace std;

int n, l, r, last;
vector<int> v;
string ans;

void take_left(string &target) {
    target.push_back('L');
    last = v[l];
    l++;
}

void take_right(string &target) {
    target.push_back('R');
    last = v[r];
    r--;
}

void solve_for_different() {
    l = 0;
    r = n - 1;
    last = -1; // nieco mensie, ako su cisla na vstupe
    while(l <= r) {
        if(v[l] <= last && v[r] <= last)
            break;
        if(v[l] == v[r])
            break; // Tento pripad osetrime druhou funkciou
        if(v[l] > last && v[r] > last) {
            // Mozeme zobrat obe, ale my chceme to mensie z nich
            if(v[l] < v[r])
                take_left(ans);
            else
                take_right(ans);
        }
        else {
            if(v[l] > last)
                take_left(ans);
            else
                take_right(ans);
        }
    }
}

void solve_for_same() {
    // Ak po dokonceni solve_for_different nevyhovuje ani v[l], ani v[r], tak uz nic nezlepsime
    if(v[l] <= last && v[r] <= last)
        return;
    // Teraz urcite plati, ze v[l] == v[r]
    // Zoberieme teda co najviac cisiel iba zlava, alebo iba sprava
    // Vyberieme si lepsiu z tychto 2 moznosti

    int oldl = l, oldr = r, oldlast = last;
    string suffix_left, suffix_right;
    // Najskor vsetky zlava
    while(l <= r && v[l] > last)
        take_left(suffix_left);

    l = oldl;
    r = oldr;
    last = oldlast;
    // Teraz vsetky sprava
    while(l <= r && v[r] > last)
        take_right(suffix_right);

    // Zoberieme lepsiu z tych dvoch moznosti, v pripade rovnosti je nam jedno ktoru
    if(suffix_left.size() > suffix_right.size())
        ans += suffix_left;
    else
        ans += suffix_right;
}

int main() {
    cin >> n;
    v.resize(n);
    for(int i = 0; i < n; i++)
        cin >> v[i];
    solve_for_different(); // Toto staci na riesenie vstupov kde su vsetky cisla rozne
    solve_for_same(); // Toto doriesi aj tazsie vstupy
    cout << ans << '\n';

    return 0;
}
```

Listing programu (Python)

```python
n = int(input())
b = [int(x) for x in input().split()]

L = 0
R = n - 1
last = -1
ans = ''

# Najskor berme vzdy najmensie
```

```
while L <= R:
    if b[L] <= last and b[R] <= last:
        break
    if b[L] == b[R]:
        break
    if b[L] > last and b[R] > last:
        if b[L] < b[R]:
            ans += 'L'
            last = b[L]
            L += 1
        else:
            ans += 'R'
            last = b[R]
            R -= 1
    elif b[L] > last:
        ans += 'L'
        last = b[L]
        L += 1
    else:
        ans += 'R'
        last = b[R]
        R -= 1

# Teraz bud uz nevieme pokracovat, alebo b[L] == b[R]
if b[L] == b[R] and b[L] > last:
    suffix_left = ''
    suffix_right = ''
    oldL, oldR, oldlast = L, R, last

    while L <= R and b[L] > last:
        suffix_left += 'L'
        last = b[L]
        L += 1

    L, R, last = oldL, oldR, oldlast
    while L <= R and b[R] > last:
        suffix_right += 'R'
        last = b[R]
        R -= 1

    if len(suffix_left) >= len(suffix_right):
        ans += suffix_left
    else:
        ans += suffix_right

print(ans)
```

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

5. Krajina špiónov

Na vstupe dostaneme sieť špiónov, o ktorej platí:

1. **Každý špión sleduje práve jedného iného špióna.**
2. **Každý špión je sledovaný práve jedným iným špiónom.**

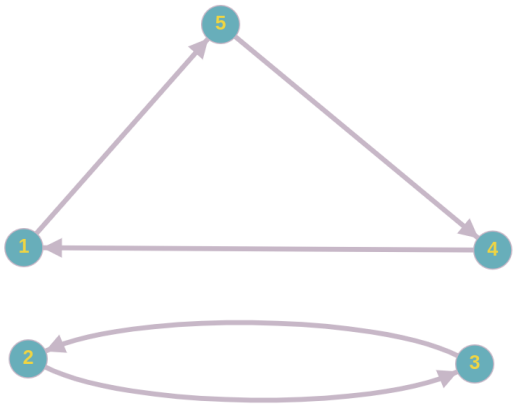
Našou úlohou je zistiť koľko najviac špiónov môžeme vybrať tak, aby boli sledovaní nevybranými špiónmi.

Špióni na grafe

Úlohu si môžeme vizualizovať pomocou grafu. Grafom myslíme "informatický" graf, nie matematický. *Informatický graf* obsahuje *vrcholy* (bodky) a *hrany* (čiary medzi vrcholmi).

Špióni budú vrcholy a vzťah "špión A sleduje špióna B" zakreslíme pomocou hrany z A do B. Napríklad:

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

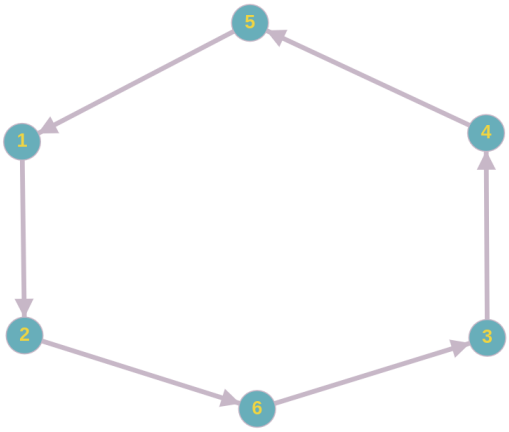


Hrany na našom grafe majú na jednej strane šípku. Takýmto hranám hovoríme *orientované*. Hrana vždy smeruje k sledovanému špiónovi.

Špióni tvoria cykly

Všimnite si, že vďaka podmienkam 1) a 2) vyššie platí, že do každého vrcholu vstupuje práve jedna hrana (hlavička šípky) a práve jedna hrana vystupuje (päta šípky). To znamená, že ak budeme od nejakého vrcholu A postupovať ďalej po hranách, vždy skončíme späť vo vrchole A. Cestu, ktorá začne a skončí v rovnakom vrchole nazývame *cyklus*. V okrajovom prípade použije takáto cesta všetkých n hrán, viď. nasledujúci príklad.

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



Koľko najviac špiónov môžeme v tomto príklade vybrať tak, aby boli sledovaní nevybranými špiónmi? Alebo inak, koľko najviac vrcholov môžeme vybrať tak, aby do nich nevstupovala hrana z už vybraného vrcholu? Odpoveď je 3. Na začiatku vyberieme ľubovoľného špióna a ďalej vyberáme každého druhého až pokiaľ neprejdeme celý cyklus. Pre cyklus párnej dĺžky d je tak odpoveď $d/2$. Podobne sa dá ukázať, že pre cyklus nepárne dĺžky je odpoveď $d/2$ zaokrúhlené nadol.

Takto dostávame myšlienku algoritmu: rozložíme graf na cykly a už vieme, že z každého cyklu môžeme vybrať polovicu vrcholov.

Riešenie rekurziou

Nasledujúce riešenie nájde cykly rekurzívne.

Listing programu (Python)

```
#!/usr/bin/python3

n = int(input())
# Spionov v programe cislujeme od 0.
# G je list reprezentujuci graf spionov. Spion i sleduje spiona G[i].
```

```
G = n * [None]
vstup = input().split()
for i in range(n):
  G[i] = int(vstup[i]) - 1

# List, v ktorom si pamatame ktore vrcholy uz algoritmus navstivil.
navstivene = n * [False]

# Rekurzivna funkcia, ktora dostane aktualny vrchol na cykle
# a zaciatocny vrchol cyklu.
def dlzka_cyklu(v, zaciatok):
  navstivene[v] = True
  if v == zaciatok:
    return 1
  # Posunieme sa v cykle o jeden vrchol dalej.
  return dlzka_cyklu(G[v], zaciatok) + 1

# Prejdeme vsetky vrcholy. Pri tych, ktore sme este nenavstivili,
# zistime dlzku cyklu, na ktorom lezia a zvacsime vysledok.
vysledok = 0
for v in range(0, n):
  if not navstivene[v]:
    vysledok += dlzka_cyklu(G[v], v) // 2

print(vysledok)
```

Každý cyklus objavíme práve raz, pretože pri objavovaní označíme všetky vrcholy cyklu za navštívené. Okrem toho iba v jednom cykle prejdeme všetky vrcholy. Časová zložitosť je tak $O(n)$. Pamäťová zložitosť je rovnako $O(n)$, pretože používame iba dve polia na načítanie vstupu obsahujúce n čísel.

Časová zložitosť je zjavne optimálna (lepšie riešenie ako $O(n)$ nevymyslíme, pretože potrebujeme načítať vstup). Zároveň ale tento program stačí iba na 9 bodov. Dôvodom je rekurzia, ktorá sa pri najväčšom vstupe môže zanoriť až 1 000 000-krát. Pretože hlboká rekurzia je pamäťovo náročná, program na testovači spadne na pretečenie pamäte a dostaneme hlášku "Chyba počas behu programu".

Riešenie bez rekurzie

Toto riešenie využíva rovnakú myšlienku ako to predošlé, ale rekurziu sme nahradili cyklom. To stačí na 15 bodov :)

Listing programu (Python)

```
#!/usr/bin/python3

n = int(input())
# Skratene sa da nacitanie vstupu z predosleho riesenia zapisat
# pomocou "list comprehension".
G = [int(x) - 1 for x in input().split()]
navstivene = n * [False]

def dlzka_cyklu(zaciatok):
  dlzka = 0
  v = zaciatok
  while not navstivene[v]:
    navstivene[v] = True
    dlzka += 1
    v = G[v]

  return dlzka

vysledok = 0
for v in range(0, n):
  if not navstivene[v]:
    vysledok += dlzka_cyklu(v, navstivene, G) // 2

print(vysledok)
```