Riešenia kategórie B

B-II-1 Múr

Podúloha A

Na začiatok si môžeme všimnúť, že keď poznáme výšky všetkých stĺpcov v múre, vieme ľahko zistiť, aké vysoké majú byť stĺpce v zarovnanom múre. Jednoducho sčítame všetky výšky na začiatku a vydelíme ich počtom stĺpcov – číslom n. Výšku každého stĺpca v zarovnanom múre si označme k. Teraz vieme pre každý stĺpec povedať, či je v ňom správny počet kociek, či tam nejaké chýbajú, alebo či sú tam nejaké navyše.

Skúsme sa zamyslieť nad tým, ktorými kockami budeme musieť určite pohnúť. Určite budeme musieť posunúť všetky kocky v stĺpcoch, ktoré sú vyššie ako k. A keďže v tejto podúlohe ich môžeme presúvať ľubovoľne, určite sa ich oplatí dať do stĺpcov, ktorým nejaké kocky chýbajú. No a ak sme zarovnali všetky stĺpce, ktoré boli vyššie ako k, musí to znamenať, že sme zarovnali aj všetky stĺpce, ktoré boli nižšie, lebo ako vieme zo zadania, múr sa musí dať zarovnať. Posunieme teda naozaj minimálny počet kociek.

Algoritmus riešiaci túto podúlohu preto vyzerá nasledovne. Načítame si počet stĺpcov n a výšky jednotlivých stĺpcov si zapamätám do poľa. Spočítame výšky všetkých stĺpcov, aby sme zistili, koľkými kockami je tvorený celý múr a toto číslo vydelíme číslom n, čím dostaneme hodnotu k. Naposledy prejdeme pole obsahujúce výšky stĺpcov a vždy, keď nájdeme stĺpec vyšší ako k, pripočítame si k výsledku o koľko kociek prevyšuje tento stĺpec výšku k. Súčet týchto hodnôt bude hľadaný minimálny počet kociek, ktoré musíme presunúť.

Časová aj pamäťová zložitosť takéhoto riešenia je lineárna – $O(n)$, keďže iba párkrát prejdeme cez pole a čo-to posčitujeme.

Listing programu (C++)

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

int main () {
    int n, sucet = 0, spravna_vyska, vysledok = 0;
    cin >> n;
    vector<int> vysky(n);

    // Nacitame do pola a rovno spocitame celkovy pocet kociek.
    for (int i = 0; i < n; i++) {
        cin >> vysky [i];
        sucet += vysky [i];
    }

    // Spocitame spravnu vysku, aku maju mat vsetky stlpce v rovnom mure.
    spravna_vyska = sucet / n;

    // Pre kazdy stlpec zistime, ci a kolko ma nadbytocnych kociek.
    // Ak nejake ma, pridame ich k vysledku.
    for (int i = 0; i < n; i++) {
        if (vysky [i] > spravna_vyska) {
            vysledok += vysky [i] - spravna_vyska;
        }
    }

    cout << vysledok << endl;
}
```

Podúloha B

Čo by nám mohlo napadnúť ako prvé, je presúvať kocky z nejakého stĺpca, kde sú navyše do najbližšieho stĺpca, kde nejaké chýbajú. Tento postup ale nemusí byť správny, pokiaľ neuhádneme správne, v akom poradí budeme spracúvať prečnievajúce stĺpce. Vezmime si napríklad vstup s výškami $1, 2, 2, 3, 1, 2, 2, 2, 3$. V tomto prípade najlepšie, čo môžeme spraviť, je preniesť kocku zo štvrtého stĺpca na prvý (na 3 presuny) a z posledného na piaty (na 4 presuny), dokopy teda 7 presunov. Ak by sme ale zo štvrtého stĺpca naivne presúvali kocku na piaty (1 presun), z posledného by sme museli potom presúvať až na prvý (8 posunov), čiže dokopy 9 presunov.

Samozrejme, keby sme v tomto prípade najprv spracovali posledný stĺpec, dopracovali by sme sa k správnemu riešeniu. Táto myšlienka teda nie je úplne zlá. Treba však k problému pristupovať systematickejšie. V našom riešení budeme postupne spracovávať stĺpce v poradí zľava doprava.

Pre jednoduchosť začneme tým, že spočítame správnu výšku k a následne od každej z hodnôt a_i odpočítame k. Dostaneme takto hodnoty, ktoré nám hovoria, koľko kociek máme v príslušnom stĺpci navyše, resp. koľko nám ich tam chýba. Našim cieľom je popresúvať kocky tak, aby sme všetky hodnoty zmenili na nuly: nikde nič nechýba ani neprevyšuje. Toto spravíme tak, že postupne pôjdeme zľava doprava, pričom budeme presúvať kocky tak, aby naľavo od aktuálneho stĺpca už bolo všetko upratané.

Pozrime sa na prvý, najľavejší stĺpec:

1. Ak $a_1 = 0$ (v prvom stĺpci je presne správny počet kociek), nič netreba robiť.
2. Čo ak je číslo $a_1 = x$ kladné (t.j. máme tu kocky navyše)? Keďže sme na úplnom kraji, tieto prebytočné kocky musíme poslať doprava. Zaplatíme teda x eur, zmeníme a_1 na nulu a zväčšíme a_2 o x.
3. No a čo ak je číslo $a_1 = -x$ záporné? To znamená, že nám v tomto stĺpci x kociek chýba. Tieto kocky budeme niekedy v budúcnosti musieť presunúť z druhého stĺpca do prvého. Už teraz za tieto presuny zaplatíme x eur. Zmeníme a_1 na nulu a zmenšíme a_2 o x. (Kocky, ktoré doteraz chýbali v prvom stĺpci, odteraz chýbajú v druhom.)

   Všimnite si, že vlastne robíme to isté ako v bode 2. Keď sme mali tri kocky navyše, tak sme posunuli tri kocky o stĺpec doprava. Keď máme „navyše“ mínus päť kociek, jednoducho posunieme doprava mínus päť kociek `:-)`

Presne rovnakú úvahu potom zopakujeme pre druhý, tretí, ..., až n-tý stĺpec. Keď spracúvame hociktorý stĺpec, už je naľavo od neho všetko spracované, upratané a zaplatené. Takže ak máme kocky navyše, musíme ich poslať ďalej doprava (bod 2), a naopak, ak nám v aktuálnom stĺpci nejaké kocky chýbajú, musia nám sem časom prísť sprava (bod 3).

Takéto riešenie má zjavne lineárnu pamäťovú aj časovú zložitosť $O(n)$.

Listing programu (C++)

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

using namespace std;

int main () {
    int n, sucet = 0, spravna_vyska, vysledok = 0, x = 0;
    cin >> n;
    vector <int> vysky(n+1,0);

    // Nacitame, zratame sucet vysok a spravnu vyslednu vysku.
    for (int i = 0; i < n; i++) {
        cin >> vysky [i];
        sucet += vysky [i];
    }
    spravna_vyska = sucet / n;

    // Vypocitame kde kolko kociek mame navyse / kde kolko chyba
    for (int i = 0; i < n; i++) {
        vysky[i] -= spravna_vyska;
    }

    // Prechadzame zlava doprava stlpcami a postupne ich upravujeme.
    for (int i = 0; i < n; i++) {
        vysledok   += abs( vysky[i] ); // zaplatime za kocky iduce doprava/sprava
        vysky[i+1] += vysky[i];        // upravime pocty hovoriace kde kolko chyba
        vysky[i]    = 0;
    }

    cout << vysledok << endl;
}
```

Dôkaz správnosti

Ostáva nám ešte ukázať, že nami navrhnutý algoritmus je naozaj správny a múr zarovná najlacnejšie.

Predstavme si, že sme medzi každé dva stĺpce kociek umiestnili nejaký oddeľovač, napríklad list papiera. Celkový počet presunov kociek (teda počet eur, ktoré zaplatíme) vieme teraz šikovne sčítať tak, že samostatne pre každý oddeľovač spočítame tie presuny, pri ktorých kocka prechádza ponad neho.

Pozrime sa na jeden ľubovoľný oddeľovač. Samozrejme vieme spočítať, koľko kociek je momentálne naľavo a koľko napravo od neho. A takisto vieme zistiť, koľko kociek by malo byť naľavo a napravo od neho v zarovnanom múre. Z toho vieme vyrátať, koľko *najmenej* kociek určite musí prejsť ponad tento konkrétny oddeľovač.

Príklad. Predstavme si, že máme stĺpce výšok $10, 8, 11, 7, 10, 10, 14$. Pozrime sa na oddeľovač medzi štvrtým a piatym stĺpcom. Naľavo od tohto oddeľovača je momentálne $10 + 8 + 11 + 7 = 36$ kociek. V opravenom múre ich tam ale má byť až $10 + 10 + 10 + 10 = 40$. To znamená, že cez tento konkrétny oddeľovač musíme sprava doľava presunúť 4 kocky.

Samozrejme, je možné, že v nejakom riešení pre náš príklad by sme cez dotyčný oddeľovač presunuli 14 kociek sprava doľava a iných 10 zľava doprava. Aj takto sa dá dosiahnuť, že na konci bude naľavo o 4 kocky viac ako na začiatku – lenže v takomto prípade by sme pri presunoch kociek cez náš oddeľovač zaplatili až 24 eur namiesto 4 eur.

V optimálnom riešení sa nám však nemôže stať, aby ponad nejaký oddeľovač išli kocky oboma smermi – aj zľava doprava aj zprava doľava. Ak by sa niečo také stalo, tak naľavo od oddeľovača by bola kocka A, ktorá chce ísť napravo a napravo od oddeľovača je kocka B, ktorá chce ísť naľavo. Potom ale vieme ušetriť aspoň 2 eurá tak, že kockám A a B vymeníme ciele, a teda ani jedna z nich ponad náš oddeľovač nepôjde. V optimálnom riešení sa samozrejme nedá nič ušetriť, a teda takéto situácie v ňom nemôžu nastať. Inými slovami, v optimálnom riešení ponad každý konkrétny oddeľovač idú kocky iba jedným smerom.

Zostáva si len uvedomiť, že riešenie, ktoré nájde náš algoritmus, túto podmienku naozaj spĺňa. Zakaždým sme buď poslali nejaké kocky doprava, alebo nám kocky chýbali a vtedy sme ich sprava zobrali. Nikdy sme to však nekombinovali. Máme teda riešenie, v ktorom ponad každý oddeľovač prechádza iba nutné minimum kociek, ktoré je potrebné na to, aby sa nám múr zarovnal.

B-II-2 Miešanie kariet

Podúloha A.

Keď budeme balíček kariet miešať na Kleofášovom stroji stále dookola, eso v balíčku bude meniť svoju pozíciu. Možných pozícií esa v balíčku je n, teda najneskôr po n zamiešaniach sa nám musí stať, že eso sa ocitlo na pozícii, v ktorej už niekedy pred tým bolo. Keďže pozíciu esa po premiešaní ovplyvňuje iba jeho pozícia pred premiešaním, od tohto okamihu sa eso bude hýbať úplne rovnako, ako sa hýbalo, keď bolo na rovnakej pozícii naposledy. Inými slovami, jeho pozície sa budú periodicky opakovať. To znamená, že ak sa počas prvých n zamiešaní eso nedostalo na vrch balíčka, už sa tam nedostane nikdy. Stačí nám teda odsimulovať, ako sa bude hýbať eso počas prvých n zamiešaní. Ak sa niekedy počas toho dostane na vrch balíčka, vypíšeme, po koľkých krokoch to nastalo. Ak sa počas prvých n miešaní nedostane na vrch balíčka, môžeme vypísať, že sa tam nedostane nikdy.

Jedno zamiešanie vieme simulovať v konštantnom čase, keďže nám stačí pamätať si a aktualizovať iba pozíciu esa. Celý algoritmus teda pobeží v lineárnom čase.

Listing programu (C++)

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

int main() {
        //nacitanie vstupu
        int n;
        scanf("%d",&n);
        vector<int> a(n+1);
        for(int i=1; i<=n; i++)scanf("%d",&a[i]);
        int pozicia_esa;
        scanf("%d",&pozicia_esa);
```

```
        //simulacia
        bool ano = 0;
        for(int i=0; i<n; i++) {
                if(pozicia_esa == 1) {
                        ano = 1;
                        printf("ano\n");
                        printf("%d\n",i);
                        break;
                }
                pozicia_esa = a[pozicia_esa];
        }
        if(!ano)printf("nie\n");
        return 0;
}
```

Ďalším zaujímavým pozorovaním je, že keď sa eso prvý raz dostane do pozície, v ktorej už bolo, bude to určite pozícia, v ktorej bolo eso úplne na začiatku. Ak by to totiž bola nejaká iná pozícia, znamenalo by to, že sa do tejto pozície mohlo eso dostať z dvoch rôznych miest (z jedného miesta sa do nej dostalo prvý raz a z iného druhý raz). To ale nie je možné, keďže na každú pozíciu v balíčku sa vždy dostane práve jedna karta. Eso teda mení pozície periodicky už od začiatku.

Podúloha B.

Ako sme si ukázali v predošlej podúlohe, pozícia jednej konkrétnej karty sa pri miešaní bude meniť periodicky. Periódu, s ktorou sa opakuje pozícia esa, označme p a periódu kráľa označme q. Tieto dve periódy vieme obe nájsť v lineárnom čase jednoducho tak, že budeme simulovať pohyb esa (resp. kráľa – každú kartu simulujeme zvlášť), až kým sa nedostane naspäť do pozície, v ktorej začínalo. Pri tom si môžeme rovno všimnúť, po koľkých krokoch sa eso dostane na vrch balíčka (resp. po koľkých krokoch sa kráľ dostane na druhú pozíciu zhora). Ak zistíme, že srdcové eso sa nikdy nedostane na vrch balíčka, prípadne že kráľ sa nikdy nedostane na druhú pozíciu zvrchu, môžeme hneď vypísať, že situácia, ktorá Kleofáša zaujíma, nikdy nenastane.

Inak si počet krokov, po ktorých sa eso dostane na vrch balíčka, označme k a počet krokov, po ktorých sa kráľ dostane na druhú pozíciu zvrchu označme l. Vieme, že eso je na vrchu balíčka po k krokoch a potom každých p krokov, teda vždy, keď počet doteraz urobených krokov dáva zvyšok k po delení p. Podobne kráľ je na druhej pozícii zhora vtedy, keď počet doteraz urobených krokov dáva zvyšok l po delení q. Situácia, ktorá Kleofáša zaujíma, teda nastane vždy po takom počte krokov od začiatku, ktorý dáva zvyšok k po delení p a zvyšok l po delení q.

Najmenší spoločný násobok čísel p a q označme m. Všimnime si, že po m krokoch od začiatku budú kráľ aj eso na rovnakej pozícii, ako boli na začiatku. Od tohoto okamihu sa budú ich pozície meniť rovnako, ako sa menili od začiatku. Teda ak doteraz nenastala situácia, v ktorej je eso na vrchu balíčka a kráľ hneď pod ním, potom takáto situácia nenastane nikdy. Stačilo by nám teda odsimulovať pohyb kráľa a esa počas prvých m krokov a zistiť, či niekedy boli v správnej pozícii (a prípadne vypísať, kedy táto pozícia nastala). Keďže p aj q môžu byť rádovo n, ich najmenší spoločný násobok môže byť rádovo n^2, teda takéto riešenie by malo časovú zložitosť $O(n^2)$.

(Konkrétny príklad zlej situácie pre priamu simuláciu: máme $n = 2k$ kariet, pričom eso leží na cykle dĺžky $p = k$ a kráľ na inom cykle dĺžky $q = k-1$. V takejto situácii je možné, že sa hľadaná situácia prvýkrát vyskytne až po $k^2 - k - 1 = n^2/4 - n/2 - 1$ krokoch.)

Úlohu však vieme riešiť aj šikovnejšie. Stačí si uvedomiť, že situácia, ktorá Kleofáša zaujíma, nemôže nastať hocikedy. Vieme, že zaujímavá situácia môže nastať len keď je eso na vrchu balíčka, teda keď počet krokov od začiatku dáva zvyšok k po delení p. Prejdeme teda postupne všetky čísla, ktoré dávajú zvyšok k po delení p a sú menšie ako m (teda čísla $k, p+k, 2p+k, \ldots, \left(\frac{m}{p} - 1\right)p + k$) a pre všetky z nich overíme, či dané číslo dáva zvyšok l po delení q, teda či je po takomto počte krokov kráľ druhý zhora. Ak nejaké takéto číslo nájdeme, potom je to hľadaný počet krokov. Ak takéto číslo nenájdeme, potom sa kráľ a eso nikdy naraz nedostanú do situácie, ktorá Kleofáša zaujíma. Keďže $m \leq pq$, tak $\frac{m}{p} \leq q$, teda potrebujeme overiť najviac q čísel. No a keďže $q \leq n$, toto overovanie (a teda aj celý algoritmus) pobeží v lineárnom čase. Ak sme pri implementácii leniví a nechce sa nám počítať najmenší spoločný násobok dvoch čísel, môžeme prezrieť všetky podozrivé čísla až po pq, časovú zložitosť nám to nezhorší.

Listing programu (C++)

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

int main() {
        //nacitanie vstupu
        int n;
        scanf("%d",&n);
        vector<int> a(n+1);
        for(int i=1; i<=n; i++) scanf("%d",&a[i]);
        int pozicia_karty[2], cielova_pozicia[2] = {1, 2}; //index 0 ma eso, index 1 ma kral
        scanf("%d␣%d",&pozicia_karty[0], &pozicia_karty[1]);

        //zistovanie p, q, k, l (v nasom kode perioda[0], perioda[1], prvy_raz[0] a prvy_raz[1])
        int perioda[2], prvy_raz[2];
        for(int i=0; i<2; i++) {
                int pozicia = pozicia_karty[i];
                for(int j=0; j<n; j++) {
                        if(pozicia == cielova_pozicia[i]) prvy_raz[i] = j;
                        pozicia = a[pozicia];
                        if(pozicia == pozicia_karty[i]) {
                                perioda[i] = j+1;
                                break;
                        }
                }
        }

        //hladanie riesenia
        bool ano;
        for(int krokov=prvy_raz[0]; krokov<perioda[0] * perioda[1]; krokov += perioda[0]) {
                if(krokov % perioda[1] == prvy_raz[1]) {
                        ano = 1;
                        printf("ano\n");
                        printf("%d\n",krokov);
                        break;
                }
        }
        if(!ano)printf("nie\n");
        return 0;
}
```

B-II-3 Viac pokazených kalkulačiek

Postupne si ukážeme ku každej úlohe aspoň jedno možné riešenie. Zdôrazňujeme, že nejde o jediné správne riešenia – aj každý iný postup vedúci k zadanému cieľu si zaslúži body.

Podúloha A: Janka pripočíta 13

Pripočítať 12 by bolo ľahké, na to by stačilo stlačiť `+4+4+4=` a boli by sme hotoví. Ako ale pripočítať tú poslednú jednotku?

Asi najjednoduchší spôsob pripočítania jednotky vyzerá nasledovne: `*4+4/4`. Teda postupne číslo na displeji vynásobíme 4 (zmení sa z x na $4x$), potom pripočítame 4 (čím na displeji dostaneme $4x+4$) a na záver vydelíme 4 (čím sa displej zmení na želané $x+1$).

Jedným možným riešením úlohy je teda postup `+4+4+4*4+4/4=`. Iným riešením by bolo jednoducho 13-krát zopakovať pripočítanie jednotky.

Podúloha B: Mirko vyrába 47 z čísla 2

Mirko na svojej kalkulačke číslo 47 vyrobiť nevie. Totiž pomocou cifry 2 vie do kalkulačky zadať len párne čísla, no a rozdiel aj súčin párnych čísel je opäť párny. Vyrobiť na displeji nepárne číslo sa mu preto nemôže nikdy podariť.

Podúloha C: Mirko vyrába 1234

Len pomocou odčítania vie Mirko vyrobiť číslo 1234 tak, že začne z čísla 2222 a postupne, kým to ide, odčítava najskôr 222, potom 22 a nakoniec 2: `2222-222-222-222-222-22-22-22-22-2-2-2-2-2-2=`.

Ak chceme ešte nejaké stlačenia ušetriť, potrebujeme zapojiť aj násobenie. Namiesto 2222 si napríklad môžeme vyrobiť číslo 1776 ako `222*2*2*2`, čo nám ušetrí nejaké odčítavania:

`222*2*2*2-222-222-22-22-22-22-2-2-2-2-2=`.

Existuje ale aj pekný systematický spôsob ako 1234 (alebo ľubovoľné iné celé číslo) šikovne vyrobiť pomocou odčítaní a násobení. Využijeme pri ňom dvojkovú sústavu.

Ukážeme si to najskôr na malých príkladoch.

Číslo $16 = 2^4$ vieme zapísať ako `2*2*2*2`.

Číslo $14 = 2^4 - 2$ vieme zapísať ako `2*2*2*2 - 2`.

Číslo $12 = 2^4 - 2^2$ **nevieme** zapísať ako `2*2*2*2 - 2*2`, lebo nemáme priority operátorov: posledné `*2` by vynásobilo dvoma všetko. Môžeme si však upraviť jeho zápis nasledovne: $2^4 - 2^2 = (2^3 - 2^1) \cdot 2$. No a toto už vieme zapísať na našej kalkulačke ako `2*2*2-2*2`.

Podobne vieme postupovať aj vo všeobecnom prípade – presnejšie, pre hocijaké párne číslo. Zoberme napr. $56 = 2^7 - 2^6 - 2^3$. Jeho výpočet na našej kalkulačke môže vyzerať nasledovne: `2*2-2*2*2*2-2*2*2`, čiže $((2^2 - 2) \cdot 2^3 - 2) \cdot 2^2$.

Pre 1234 vieme pomocou mocnín dvojky vyrobiť nasledovný zápis:

$$1234 = 2^{11} - 2^9 - 2^8 - 2^5 - 2^3 - 2^2 - 2^1$$

Tomu zodpovedá nasledovný výpočet na kalkulačke: `2*2*2-2*2-2*2*2*2*2-2*2*2-2*2-2*2-2=`.

Na tento istý algoritmus sa môžeme dívať aj nasledovne:

- Ak chceme zapísať číslo 2, zapíšeme `2`.
- Ak chceme zapísať číslo tvaru $4k$, zapíšeme číslo tvaru $2k$ a na koniec pridáme `*2`.
- Ak chceme zapísať číslo tvaru $4k + 2$, zapíšeme číslo tvaru $2k + 2$ a na koniec pridáme `*2-2`.

Podúloha D: Tomáš vyrába číslo 3

Pokračujúc príklad v zadaní, `MR+MR+MR` vyrobí hodnotu -6, ktorú stačí vydeliť hodnotou -2, čím dostaneme želanú hodnotu 3. Jedným možným riešením úlohy je teda postupnosť tlačidiel `MR+MR+MR/MR=`.

Podúloha E: Tomáš vyrába celé čísla

Stláčať `MR` viackrát za sebou nemá rovnaký efekt ako stláčanie cifier: keď sme stlačili cifru `2` trikrát, dostali sme na displeji číslo `222`, ale keď trikrát stlačíme `MR`, trikrát nám to z pamäte vyvolá tú istú hodnotu. Aj po treťom stlačení bude teda na displeji stále tá istá hodnota – v našom prípade hodnota -2.

Našťastie vieme sčítať a násobiť, stále teda pripadá do úvahy použitie vhodnej číselnej sústavy. Správne riešenie bude až prekvapivo jednoduché.

Skôr, než si popíšeme samotné programy pre našu kalkulačku, pozrime sa na zaujímavú číselnú sústavu. Základom tejto sústavy bude práve číslo -2. Podobne ako v obyčajnej dvojkovej sústave, aj v tejto budeme používať len dve cifry: 0 a 1. Tejto sústave sa niekedy hovorí *negabinárna*.

Keď v sústave so základom z zapíšeme číslo s ciframi `abcde`, jeho hodnota je $a \cdot z^4 + b \cdot z^3 + c \cdot z^2 + d \cdot z^1 + e \cdot z^0$. Ak teda v našej sústave so základom $z = -2$ zapíšeme napr. číslo `10110`, jeho hodnota je $1 \cdot (-2)^4 + 0 \cdot (-2)^3 + 1 \cdot (-2)^2 + 1 \cdot (-2)^1 + 0 \cdot (-2)^0$.

Negabinárna sústava má jednu veľmi zaujímavú vlastnosť: *aj bez použitia znamienka mínus v nej vieme zapísať ľubovoľné celé číslo.*[1] To si teraz dokážeme.

Majme reťazec cifier `r` predstavujúci číslo r. Čo sa stane, ak za tento reťazec dopíšeme `0`? Všetky cifry sme tým posunuli o rád doľava, čím sme vlastne hodnotu každej z nich vynásobili číslom -2. Reťazec `r0` teda predstavuje číslo $-2r$. Podobne, reťazec `r1` predstavuje číslo o 1 väčšie, teda $-2r + 1$.

Všimnime si, že rovnako ako v klasickej binárnej sústave, aj tu platí, že čísla končiace `0` sú párne a čísla končiace `1` sú nepárne. Na tomto pozorovaní založíme aj náš algoritmus, ktorý ukáže, ako ľubovoľné celé číslo zapísať v negabinárnej sústave. (Algoritmus bude v podstate rovnaký ako pre klasickú dvojkovú sústavu.)

[1] Tento zápis je dokonca, až na prípadné nuly na začiatku, jednoznačný. Túto skutočnosť však nebudeme potrebovať.

- Ak chceme zapísať číslo 0, nezapíšeme nič a sme hotoví.
- Ak chceme zapísať nenulové párne číslo $2k$, zapíšeme číslo $-k$ a za jeho zápis dopíšeme 0.
- Ak chceme zapísať nepárne číslo $2k+1$, zapíšeme číslo $-k$ a za jeho zápis dopíšeme 1.

Vyššie uvedený postup je určite konečný, lebo v každom kroku sa absolútna hodnota čísla, ktoré ešte treba zapísať, zmenší. (Až na jednu výnimku. Ak ju nevidíte sami, uvidíte ju pri konci nasledujúceho príkladu.) Po konečnom počte krokov teda dosiahneme nulu.

Príklad:

- Číslo 47 zapíšeme tak, že zapíšeme -23 a za jeho zápis dopíšeme cifru 1.
- Číslo -23 zapíšeme tak, že zapíšeme 12 a za jeho zápis dopíšeme cifru 1.
- Číslo 12 zapíšeme tak, že zapíšeme -6 a za jeho zápis dopíšeme cifru 0.
- Číslo -6 zapíšeme tak, že zapíšeme 3 a za jeho zápis dopíšeme cifru 0.
- Číslo 3 zapíšeme tak, že zapíšeme -1 a za jeho zápis dopíšeme cifru 1.
- Číslo -1 zapíšeme tak, že zapíšeme 1 a za jeho zápis dopíšeme cifru 1.
- Číslo 1 zapíšeme tak, že zapíšeme 0 a za jej zápis dopíšeme cifru 1.
- Kvôli číslu 0 už nepíšeme nič.

Čítajúc od konca teda dostávame, že v negabinárnej sústave má číslo 47 zápis 1110011.

No a keď k nejakému celému číslu poznáme jeho zápis v negabinárnej sústave, ľahko ho vypočítame na našej kalkulačke. Napríklad pre číslo 47 by jeho výpočet na Tomášovej kalkulačke vyzeral nasledovne:

$$\underbrace{\texttt{MR}}_{1}\ \underbrace{\texttt{*MR+MR}}_{1}\ \underbrace{\texttt{*MR+MR}}_{1}\ \underbrace{\texttt{*MR}}_{0}\ \underbrace{\texttt{*MR}}_{0}\ \underbrace{\texttt{*MR+MR}}_{1}\ \underbrace{\texttt{*MR+MR}}_{1}\ \texttt{/ MR =}$$

B-II-4 Regulárne výrazy

Podúloha A

Prejdime si postupne jednotlivé obmedzenia:

- Určite jedna a možno druhá lokomotíva na začiatku: „`LL?`“.
- Niekoľko poštových vozňov na konci: „`P*`“.
- Keďže vozne prvej triedy sú pre zvyšných pasažierov neprístupné, musia byť tesne za lokomotívou alebo tesne pred poštou (alebo oboje!) a je ich ľubovoľne veľa: „`1*`“.
- V strede môže byť ľubovoľne veľa vozňov druhej triedy a niekde medzi nimi možno jeden jedálenský a jeden pre cestujúcich s deťmi: „`2*J?2*D?2*`“. Netreba však zabúdať, že vozne môžu byť v ľubovoľnom poradí: „`2*(J?2*D?|D?2*J?)2*`“.

Keď to celé spojíme, dostaneme „`LL?1*2*(J?2*D?|D?2*J?)2*1*P*`“.

Podúloha B

Táto podúloha bola najľahšia. Stačilo si uvedomiť, že požadovanú operáciu (vytvoriť vzorku, ktorej zodpovedajú všetky postupnosti znakov zodpovedajúce aspoň jednej z iných dvoch vzoriek) robí symbol pre logický or („`|`“). Zadané vzorky teda stačilo dať do zátvoriek a následne zlepiť pomocou tohto symbolu: „`( b*(ab*ab*)* ) | ( (a|b)*b(a|b)*b(a|b)* )`“. Medzery sme do vzorky pridali pre názornosť.

Podúloha C

Pre túto časť, bohužiaľ, neexistuje symbol, ktorý by ju za nás vyriešil. Budeme sa teda musieť zamyslieť, čo zadané vzorky znamenajú, zlúčiť tieto dve vlastnosti a napísať úplne novú vzorku.

Vo vzorke „`b*(ab*ab*)*`“ si možno všimnúť, že v zátvorke sa znak „`a`“ vyskytuje dva krát. Zátvorka sa ľubovoľne veľa krát opakuje a teda vo výsledku sa znak „`a`“ nachádza párny počet krát. Znak „`b`“ sa môže vyskytovať v ľubovoľných množstvách a na ľubovoľných pozíciách. Tejto vzorke teda zodpovedajú práve všetky reťazce s párnym počtom „`a`“-čok.

Vo vzorke „`(a|b)*b(a|b)*b(a|b)*`“ sú dva znaky „`b`“, okolo ktorých sa môžu vyskytovať oba znaky ľubovoľne. Je teda jasné, že vzorke zodpovedajú práve všetky reťazce s aspoň dvoma „`b`“-čkami.

My teda máme napísať vzorku, ktorej bude reťazec zodpovedať práve vtedy, keď má aj párny počet „`a`“-čok, aj aspoň dve „`b`“-čka. Ako na to? Každý takýto reťazec si môžeme schematicky predstaviť nasledovne:

(nejaký počet `a`) (prvé `b`) (nejaký počet `a`) (druhé `b`) (zvyšok reťazca)

My potrebujeme zabezpečiť, aby v prvej, tretej a piatej časti reťazca bol dokopy párny počet „`a`“-čok. To spravíme tak, že budeme mať štyri disjunktné časti vzorky podľa parity počtu „`a`“-čok v prvej a v tretej časti reťazca. Jedna z týchto štyroch možností je uvedená nižšie, ostatné tri vyzerajú analogicky.

(**párny** počet `a`) `b` (**nepárny** počet `a`) `b` (**ľubovoľný** reťazec s **nepárnym** počtom `a`)

Jednotlivé malé časti našich vzoriek môžu vyzerať napr. nasledovne:

- párny počet `a`: „`(aa)*`“
- nepárny počet `a`: „`a(aa)*`“
- reťazec s párnym počtom `a`: „`b*(ab*ab*)*`“
- reťazec s nepárnym počtom `a`: „`b*ab*(ab*ab*)*`“

Hľadanú vzorku teda dostaneme tak, že pomocou logického or („`|`“) spojíme nasledovné štyri vzorky:

- „ `(aa)* b (aa)* b b*(ab*ab*)*`“
- „`a(aa)* b (aa)* b b*ab*(ab*ab*)*`“
- „ `(aa)* b a(aa)* b b*ab*(ab*ab*)*`“
- „`a(aa)* b a(aa)* b b*(ab*ab*)*`“

Podúloha D

Aj v tejto podúlohe si výslednú vzorku najskôr slovne popíšeme. Základné vzorky sú tie isté ako v predchádzajúcej podúlohe, teraz však chceme, aby ani jedna výsledku nezodpovedala. Hľadáme teda vzorku pre reťazce s najviac jedným znakom „`b`“ a zároveň nepárnym počtom znakov „`a`“.

Ak takýto reťazec neobsahuje „`b`“, popisuje ho vzorka „`a(aa)*`“. Ak obsahuje „`b`“, sú dve možnosti: buď je pred ním párny a za ním nepárny počet „`a`“, alebo je to naopak. Prvá možnosť vedie k vzorke „`(aa)* b a(aa)*`“, druhá ku „`a(aa)* b (aa)*`“.

Výsledná vzorka je logický or vyššie uvedených troch možností.

Iná konštrukcia: Predstavme si, že takýto reťazec máme. Vieme z neho postupne zo začiatku aj z konca odoberať dvojice znakov „`a`“, až kým nám nezostane jadro, ktoré obsahuje práve jedno „`a`“-čko (aby bol ich celkový počet nepárny) a najviac jedno „`b`“-čko. Vieme to teda zapísať ako „`(aa)*(ab|ba|a)(aa)*`“.

TRIDSIATY PRVÝ ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Eduard Batmendijn, Michal Forišek, Marián Horňák, Jaroslav Petrucha
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2015