Riešenia kategórie B

B-I-1 Fľaše

Na začiatku si zistíme, koľko vody vlastne má byť na konci v každej fľaši: sčítame všetky objemy v_i a výsledok vydelíme počtom fliaš.

Fľaše si teraz môžeme roztriediť do dvoch nových polí. Do jedného poľa si uložíme poradové čísla fliaš, v ktorých je vody primálo, do druhého zase čísla tých, v ktorých je vody priveľa. (Ak náhodou máme aj nejaké, v ktorých je vody akurát, tie nedáme ani do jedného z nových polí.)

Ako je dobré prelievať? Zoberme si ľubovoľnú fľašu y v ktorej je vody primálo a ľubovoľnú fľašu z, v ktorej je zas vody priveľa. Budeme liať vodu z fľaše z do fľaše y, až kým aspoň jedna z nich nebude obsahovať presne to správne množstvo vody. Inými slovami, lejeme dovtedy, kým buď vo fľaši y nestúpne, alebo vo fľaši z neklesne množstvo vody na presne želanú hranicu.

Ešte inými slovami, ak vo fľaši y chýba v_y mililitrov vody a vo fľaši z máme v_z mililitrov vody navyše, prelejeme presne $\min(v_y, v_z)$ mililitrov vody.

Každou vyššie popísanou akciou vyrobíme aspoň jednu novú fľašu, v ktorej je vody akurát. Po nanajvýš $n-1$ takýchto preliatiach teda už musí byť vo všetkých fľašiach presne rovnaké množstvo vody.

V nižšie uvedenom programe používame na uloženie našich dvoch typov fliaš zásobníky. Z nich si vždy vyberieme jednu príliš prázdnu a jednu príliš plnú fľašu, prelejeme správne množstvo vody, a ak v jednej z nich je stále primálo/priveľa vody, vrátime ju do príslušného zásobníka. Takto vieme každé preliatie odsimulovať v konštantnom čase, a teda celková časová zložitosť nášho programu je priamo úmerná počtu fliaš.

Listing programu (Python)

```
# nacitame vstup, inicializujeme premennu na vystup, vypocitame kolko vody chceme v kazdej flasi

N = int( input() )
V = [ int(x) for x in input().split() ]
riesenie = []
ciel = sum(V) // N

# vytvorime si zasobniky s cislami flias, kde je vody privela / primalo

privela, primalo = [], []
for n in range(N):
    if V[n] > ciel: privela.append(n)
    if V[n] < ciel: primalo.append(n)

# kym sa nam neminuli zle flase, prelievame:

while len(primalo) > 0:
    # zistime si cisla flias ktore ideme pouzit
    kde_vela = privela.pop()
    kde_malo = primalo.pop()

    # pozrieme sa kolko vody mame navyse / kolko jej chyba
    navyse = V[kde_vela] - ciel
    chyba  = ciel - V[kde_malo]

    # zistime kolko preliat a spravime to
    prelej = min( navyse, chyba )
    V[kde_vela] -= prelej
    V[kde_malo] += prelej

    # zapamatame si toto preliatie ako sucast riesenia
    # robime +1 lebo my cislujeme od 0 ale v zadani sa cisluje od 1
    riesenie.append( ( prelej, kde_vela+1, kde_malo+1 ) )

    # aspon jednu flasu sme takto vyriesili
    # ak ostala druha flasa nevyriesena, vratime ju do zasobnika
    if V[kde_vela] > ciel: privela.append( kde_vela )
    if V[kde_malo] < ciel: primalo.append( kde_malo )

# vypiseme vyssie zostrojene riesenie

print( len(riesenie) )
for x,y,z in riesenie: print(x,y,z)
```

B-I-2 Disc golf

Ak by sme implementovali algoritmus priamo podľa zadania, dostali by sme program, ktorého čas výpočtu by (v najhoršom možnom prípade) závisel kvadraticky od počtu jamiek. Ak by sme totiž mali hráčov, ktorí každú jamku zahrajú všetci rovnako dobre, tak by sme sa postupne na každej jamke museli pozrieť na výsledky všetkých predchádzajúcich, a nakoniec by aj tak rozhodlo až abecedné poradie.

Šikovnejšie riešenie úlohy založíme na tom, že si všimneme, čo vlastne robíme, keď zisťujeme, v akom poradí pôjdu naši hráči hádzať jamku $j+1$: pozrieme sa na výsledky jamky j, a ak sa podľa tých nerozhodlo, tak sa postupne pozeráme na výsledky jamiek $j-1, j-2, \ldots, 1$. Presne na tie isté jamky a v tom istom poradí sme sa však len nedávno pozerali – keď sme zisťovali, v akom poradí majú ísť hráči hádzať na jamke j. Ako teda naše riešenie zrýchliť? Na jamky od $j-1$ skôr sa vôbec pozerať nemusíme! My už totiž vieme, ako porovnanie na nich dopadne – výsledok tohto porovnania nám hovorí poradie, v ktorom hráči odohrali jamku j.

Formálne si teda vieme naše pravidlo preformulovať do nasledovnej ekvivalentnej podoby: *Jamku $j+1$ pôjdu hráči hádzať usporiadaní podľa toho, ako dobre zahrali jamku j. Ak nejakí hráči zahrali jamku j rovnako dobre, pôjdu jamku $j+1$ hádzať v rovnakom poradí v akom hádzali jamku j.*

No a po tejto úprave už ľahko naprogramujeme riešenie, ktorého časová zložitosť bude od počtu jamiek závisieť len lineárne. Postupne pre každú jamku hráčov preusporiadame a vypíšeme, pričom pri usporadúvaní sa pozeráme primárne na výsledok predchádzajúcej jamky a sekundárne na poradie, v ktorom ju začínali hrať.

Listing programu (Python)

```
# nacitame vstup

N, J = [ int(x) for x in input().split() ]
vysledky = [ input() for n in range(N) ]

# zostrojime si aktualne poradie -- podla abecedy

poradie = ''
for n in range(N): poradie += chr(65+n)

# postupne pre kazdu jamku:

for j in range(J):

    # najskor vypiseme aktualne poradie hracov

    print(poradie)

    # a potom hracov preusporiadame podla vysledkov tejto jamky
    # pri nasej implementacii namiesto volania sort() vyuzivame, ze vysledky jamky su 1-9

    nove_poradie = ''
    for h in '123456789':
        # do poradia pridame vsetkych hracov, ktori jamku "j" zahrali na "h" hodov
        # pritom zachovame ich poradie z predchadzajucej jamky
        for x in poradie:
            if vysledky[ord(x)-65][j] == h:
                nove_poradie += x

    poradie = nove_poradie
```

B-I-3 Pokazené kalkulačky

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 vyrába číslo 13

Pripomeňme si, že Jankinej kalkulačke fungujú už len tlačidlá „`46+-*/=`“. Vie teda písať cifry 4 a 6, sčítať, odčítať, násobiť a deliť.

Aj na takejto kalkulačke vieme ľahko vyrobiť číslo 13. Jedna z najjednoduchších možností vyzerá nasledovne: „`4/4+6+6`“. (Najskôr si pomocou „`4/4`“ vyrobíme hodnotu 1 a k tej potom pripočítame 12.)

Podúloha B: Janka vyrába ľubovoľné prirodzené číslo

Podobným trikom ako sme vyššie vyrobili jednotku vieme vlastne vyrobiť aj každé iné prirodzené číslo. Napríklad 2 vyrobíme postupnosťou stlačení „`4+4/4`“ a 3 postupnosťou stlačení „`4+4+4/4`“. Analogicky ďalej.

(Dalo by sa samozrejme trochu optimalizovať: 4 zapísať ako „`4`“, 5 ako 1+4, čiže ako „`4/4+4`“, a tak ďalej. To však nebolo potrebné: aj zápis „`4+4+4+4+4/4`“ je platným spôsobom ako vyrobiť číslo 5.)

Podúloha C: Zuzka vyrába číslo mínus 13

Na Zuzkinej kalkulačke už fungujú len tlačidlá „`68-/=`“. Oproti riešeniu podúlohy A teda nemáme sčítanie. Keďže ale aj tak chceme vyrobiť záporné číslo, príliš nám to nebude prekážať. Zafunguje podobný trik ako v podúlohe A: -13 je $1 - 14$, čiže napríklad „`6/6-6-8`“.

Podúloha D: Zuzka vyrába ľubovoľné prirodzené číslo

Teraz nám už to sčítanie na prvý pohľad môže chýbať. Ale len na prvý pohľad – totiž napríklad namiesto toho, aby sme použili sčítanie a išli od nuly dohora môžeme použiť odčítanie a ísť od viacciferného čísla dodola.

Napríklad vieme priamo vyrobiť číslo 1 111 111 nasledovne: „`6666666/6`“. Jeho úpravou vieme vyrobiť číslo 1 111 110: „`6666666-6/6`“. A analogicky ďalej. Číslo n z rozsahu od 1 do 1 000 000 teda vyrobíme tak, že začneme s „`6666666`“, potom $(1\,111\,111 - n)$-krát zadáme „`-6`“ a na konci spravíme „`/6`“ a máme želaný výsledok.

Podúloha E: Janka vyrába prirodzené čísla efektívne

Postupy, ktoré sme si ukázali v podúlohách B a D nie sú zrovna šikovné – napríklad na vyrobenie čísla 1 000 000 by Janka potrebovala postupne stlačiť až 2 000 001 tlačidiel. Ako teda vieme vyrobiť hľadané čísla efektívnejšie?

Existuje veľa podobne efektívnych postupov. My si ukážeme dva trochu odlišné.

Prvý postup založíme na tom, že číslo n, ktoré chceme vyrobiť, si zapíšeme v štvorkovej sústave. Ak napríklad chceme vyrobiť číslo $n = 4774$, zistíme si, že v štvorkovej sústave je jeho zápis $(1022212)_4$. Inými slovami, platí:

$$n = 1 \cdot 4^6 + 0 \cdot 4^5 + 2 \cdot 4^4 + 2 \cdot 4^3 + 2 \cdot 4^2 + 1 \cdot 4^1 + 2 \cdot 4^0$$

Jednotlivé cifry v štvorkovej sústave teraz použijeme na vytvorenie čísla n na našej kalkulačke. Dokola budeme striedať dve operácie: najskôr budeme robiť „`+4`“ toľkokrát, akú ďalšiu cifru chceme pridať, a potom urobíme „`*4`“, čím prejdeme na ďalší rád.

Pre vyššie uvedené $n = 4774 = (1022212)_4$ by to celé vyzeralo nasledovne:

$$\underbrace{4}_{1}\ *4\ \underbrace{}_{0}\ *4\ \underbrace{+4+4}_{2}\ *4\ \underbrace{+4+4}_{2}\ *4\ \underbrace{+4+4}_{2}\ *4\ \underbrace{+4}_{1}\ *4\ \underbrace{+4+4}_{2}\ /4$$

(Všimnite si, že na konci potrebujeme špeciálne raz urobiť „`/4`“, aby sme sa zbavili posledného rádu. Keby sme vedeli robiť operáciu „`+1`“ namiesto „`+4`“, nebol by tento krok potrebný.)

Počet stlačení tlačidiel, ktoré pri takomto postupe potrebujeme na vyrobenie čísla n, je priamo úmerný počtu cifier čísla n v štvorkovej sústave – a teda priamo úmerný logaritmu čísla n.

Pre zaujímavosť uvádzame, že pri tomto postupe sú spomedzi čísel do milióna najhoršie čísla 786 431 a 983 039. Ich zápisy v štvorkovej sústave sú $2\,333\,333\,333_4$ a $3\,233\,333\,333_4$. Na výrobu každého z nich potrebujeme 77 stlačení tlačidla. Každé iné číslo vyrobíme rýchlejšie.

Iné riešenie:

Ľubovoľné prirodzené číslo vieme naskladať ako súčet čísel, z ktorých každé je zapísané len pomocou samých jednotiek. Navyše to budeme robiť tak, že jednotlivé sčítance budeme určovať postupne, a to tak, aby každý sčítanec bol čo najväčší. Napríklad pre $n = 580$ dostaneme nasledovný zápis:

$$580 = \underbrace{111+111+111+111+111}_{555} + \underbrace{11+11}_{+22} + \underbrace{1+1+1}_{+3}$$

Takýto postup by sme mohli na našej kalkulačke použiť, keby sme mali cifru „1“. Keďže ju nemáme, zadáme do kalkulačky ten istý výraz, len namiesto jednotiek všade použijeme cifru „4“. Tým zjavne namiesto čísla n vyrobíme číslo $4n$. To nám ale vôbec neprekáža, lebo nie je nič ľahšie ako toto číslo na konci vydeliť štyrmi. Pre $n = 580$ bude teda výroba na Jankinej kalkulačke vyzerať takto: „`444+444+444+444+444+44+44+4+4+4/4`“.

Tento postup je o niečo menej efektívny ako ten, ktorý sme si ukázali ako prvý. Bez dôkazu uvedieme, že u tohto postupu je počet stlačení tlačidiel približne priamo úmerný štvorcu počtu cifier čísla n v desiatkovej sústave. Napovieme len myšlienku dôkazu: sčítanec 111 použijeme pri zápise ľubovoľného čísla n najviac desaťkrát, lebo inak by sme radšej použili viackrát sčítanec 1111.

Opäť pre zaujímavosť uvádzame, že pri tomto postupe je spomedzi čísel do milióna najhoršie číslo 999 994. Potrebujeme naň až 239 stlačení tlačidla.

B-I-4 Regulárne výrazy

Začneme tromi ľahkými podúlohami.

Vzorku, ktorej budú zodpovedať reťazce „`zaba`“ a „`zabka`“, vieme zostrojiť podľa príkladov v zadaní. Jedna možnosť je úplne priamočiara: „`zaba|zabka`“. To isté vieme zapísať stručnejšie napríklad ako „`zab(a|ka)`“. Najstručnejším riešením je vzorka „`zabk?a`“.

Ako druhú sme mali napísať vzorku, ktorej zodpovedajú iba reťazce, v ktorých každý znak je „`a`“ a ktorých dĺžka je aspoň 5. Takúto vzorku si môžeme slovne popísať nasledovne: najskôr je presne 5 „`a`“, potom nasleduje ľubovoľne veľa ďalších „`a`“. A toto už vieme zapísať pomocou známych symbolov: „`aaaaaa*`“.

V tretej podúlohe sme mali naopak zostrojiť vzorku, ktorej zodpovedali len reťazce v ktorých bolo nanajvýš 15 znakov a všetky boli „`a`“. Ako na to?

Jedna možnosť je uvedomiť si, že týchto reťazcov je len konečne veľa, a teda ich môžeme všetky vymenovať: „`|a|aa|aaa|aaaa|...`“. To je ale vcelku bolestivé.

Šikovnejším riešením je napr. vzorka „`a?a?a?a?a?a?a?a?a?a?a?a?a?a?a?`“. V tejto vzorke je 15 výskytov „`a`“, pričom každé má za sebou kvantifikátor „`?`“ hovoriaci „toto `a` vo výslednom reťazci môže ale nemusí byť“.

Trikom by sme túto konštrukciu vedeli ešte o čosi skrátiť, napr. nasledovne: „`(aaaaaaaa)?(aaaa)?a?a?a?`“. Rozmyslite si, prečo aj táto vzorka funguje. Na záver ešte podotkneme, že moderné regulárne výrazy obsahujú syntax na spríjemnenie riešenia takýchto úloh. V praxi napr. funguje aj stručná a čitateľná vzorka „`a{0,15}`“.

Prirodzené čísla deliteľné štyrmi

Využijeme známu skutočnosť: prirodzené číslo je deliteľné 4 práve vtedy, keď je jeho posledné dvojčíslie deliteľné 4. (Toto platí preto, že každý násobok 100 je zjavne deliteľný 4.)

Hľadáme teda čísla, ktorých skoro všetky cifry môžu byť ľubovoľné, len posledné dve musia byť „správne“.

Pri písaní vzorky rozoberieme tri navzájom disjunktné prípady:

- Hľadané číslo môže byť jednociferné. Takéto sú len dve: 4 a 8. Tým zodpovedá vzorka „`4|8`“.
- Hľadané číslo má aspoň tri cifry. Ako vyzerá takéto číslo? Prvá cifra je nenulová. Potom nasleduje ľubovoľne veľa ľubovoľných cifier, no a na záver je dvojčíslie ktoré je deliteľné 4.

  Tieto dvojčíslia by sme opäť mohli jednoducho všetky vymenovať, my ich ale vieme zapísať aj v kompaktnejšej podobe. Posledná cifra musí byť párna. Ak je to 0, 4, alebo 8, musí pred ňou byť párna cifra, inak pred ňou musí byť cifra nepárna. Správne posledné dvojčíslia teda zodpovedajú nasledujúcej vzorke: „`[02468][048]|[13579][26]`“.

  No a teda všetky aspoň-trojciferné násobky štyroch vieme popísať nasledovnou vzorkou: „`[1-9][0-9]*([02468][048]|[13579][26])`“.

- Hľadané číslo má práve dve cifry. Platí to isté, čo sme si vyššie povedali o poslednom dvojčíslí, ale navyše vieme, že prvá cifra nemôže byť nulová. Tentoraz teda vyhovuje vzorka „`[2468][048]|[13579][26]`“.

Celkové riešenie dostaneme ako logický or troch vyššie uvedených vzoriek. Pre názornosť sme do výslednej vzorky pridali medzery ktoré nie sú jej súčasťou.

„`4 | 8 | [2468][048] | [13579][26] | [1-9][0-9]*([02468][048]|[13579][26])`“

Zakázaný podreťazec aaa

Vo všeobecnosti je ťažké pomocou regulárnych výrazov popísať, že sa niečo nikde v reťazci nevyskytuje. Regulárne výrazy sú skôr stavané na opačnú úlohu. Ľahko napríklad popíšeme všetky reťazce, ktoré podreťazec „`aaa`“ *obsahujú*. Na to nám stačí vzorka „`.*aaa.*`“ – čiže „ľubovoľne veľa ľubovoľných znakov, potom `aaa`, a za tým ďalšie ľubovoľné znaky“.

Ak ale chceme podreťazec „`aaa`“ zakázať, musíme akoby skontrolovať úplne celý reťazec. Nikde v ňom nesmú byť tri áčka po sebe, nanajvýš jedno alebo dve. Inými slovami, každá skupina áčok má nanajvýš dve áčka, a medzi nimi sa vždy vyskytuje aspoň jedno nie-áčko. Toto už vyzerá, že sme na dobrej stope: podmienky ako „najviac dve áčka“ a „ľubovoľne veľa nie-áčok“ už formulovať vieme.

Tak ešte raz a poriadnejšie. Ako môžu vyzerať hľadané reťazce?

Jedna možnosť je, že v hľadanom reťazci vôbec žiadne áčka nie sú. Tomu zodpovedá vzorka „`[^a]*`“.

Druhá možnosť je, že tam nejaké áčka sú. V tom prípade hľadaný reťazec vyzerá nasledovne:

1. Na začiatku je nula alebo viac nie-áčok.
2. Nasleduje prvá skupina áčok, sú v nej najviac dve.
3. Za tou sa nula alebo viackrát zopakuje nasledovné: najskôr sa objaví nenulový počet nie-áčok a za nimi ďalšia skupina nanajvýš dvoch áčok.
4. Za posledným áčkom ešte opäť môže nasledovať ľubovoľne veľa nie-áčok.

Jednotlivým bodom postupne zodpovedajú nasledujúce vzorky:

1. „`[^a]*`“
2. „`aa?`“
3. „`([^a][^a]*aa?)*`“
4. „`[^a]*`“

Podrobnejšie si popíšeme vzorku z bodu 3. Začneme tým, že zapíšeme vzorku „`[^a][^a]*`“ predstavujúcu aspoň jedno nie-áčko. Za ňu pridáme vzorku „`aa?`“ predstavujúcu jednu alebo dve áčka. No a celé toto následne zabalíme do zátvoriek a použijeme kvantifikátor `*`, čím povieme, že takýchto reťazcov môže nasledovať postupne ľubovoľne veľa.

Jedna možná výsledná vzorka teda vyzerá nasledovne (opäť, s medzerami pre názornosť):

„`[^a]*  |  [^a]* aa? ([^a][^a]*aa?)* [^a]*`“

Iné, trikovejšie riešenie tejto podúlohy: Náš výraz vieme značne zjednodušiť, ak nebudeme explicitne kontrolovať prítomnosť áčok. Stačí sa sústrediť na nie-áčka: dobré reťazce sú práve tie, v ktorých vieme vyznačiť niektoré nie-áčka tak, aby medzi každými dvoma (a takisto pred prvým a za posledným) boli nanajvýš dva ľubovoľné znaky. Takto dostávame nasledovnú vzorku: „`(.?.?[^a])*.?.?`“

V praxi opäť vznikla nová syntax na riešenie takýchto nepríjemných úloh. Niektoré (avšak nie všetky) implementácie regulárnych výrazov obsahujú tzv. *negatívny lookahead*: syntax, pomocou ktorej môžeme zapísať výrok „od tohto miesta ďalej sa *nesmie* nachádzať táto vzorka“. Nám by teda stačilo pomocou tejto syntaxe zapísať výrok „na začiatku nášho reťazca sa nesmie nachádzať vzorka `.*aaa`“.

TRIDSIATY PRVÝ ROČNÍK OLYMPIÁDY V INFORMATIKE

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