B-II-1 Návrat do špajze

V tejto úlohe sme mali medzi zadanými číslami spočítať počet takých trojíc, ktoré majú tvar $c-k$, c a $c+k$ pre nejaké hodnoty c a k. Prvé čo nás napadne je vyskúšať všetky trojice. Ako však overíme, že nejaká trojica spĺňa zadané podmienky? Pripomeňme si totiž, že ani c ani k nie je zadané. Jedna možnosť je nájsť stredné z týchto troch čísel (stredné podľa veľkosti), to prehlásiť za hodnotu c a potom overiť, že rozdiel stredného a menšieho čísla je rovnaký ako rozdiel väčšieho a stredného čísla.
Pozrime sa ešte rýchlo na implementáciu. Všetky čísla si na začiatku usporiadame podľa veľkosti. Pomocou troch `for`-cyklov vieme potom postupne prechádzať všetkými trojicami zadaných čísel. A keďže sme si čísla usporiadali, tak keď máme trojicu indexov tvaru „prvé < druhé < tretie“, tak prostredné číslo podľa veľkosti je to na indexe „druhé“.

Listing programu (Python)

```
n = int(input())
cisla = [ int(x) for x in input().split() ]
cisla.sort()

vysledok = 0
for prve in range(0, n - 2):
    for druhe in range(prve+1, n - 1):
        for tretie in range(druhe+1, n):
            if cisla[druhe] - cisla[prve] == cisla[tretie] - cisla[druhe]:
                vysledok += 1
print(vysledok)
```

Toto riešenie má časovú zložitosť $O(n^3)$, čo je počet všetkých trojíc n čísel, ktoré musí náš algoritmus prezrieť. Naviac je vskutku jednoduché, takže zaň ľahko získame aspoň 3 body a pomôže nám pri ďalšom kroku.

Zlepšenie pomocou binárneho vyhľadávania

Vždy keď sa snažíme zrýchliť naše riešenie, je dobré sa zamyslieť nad tým, či nerobíme niečo zbytočne. Vidíme, že v našom riešení sa snažíme zistiť hodnoty c a k, ktoré nám pomôžu overiť, či je vybraná trojica správna. A používame na to všetky tri čísla. Je to však nutné? Koľko čísel skutočne potrebujeme na zistenie hodnôt c a k? Jedno čísla nám zjavne nestačí. Aj keby sme si povedali, že toto číslo je číslo c, netušíme, akú hodnotu môže mať číslo k. Ak si však zoberieme dve čísla, tento problém zmizne. Väčšie z týchto čísel prehlásime za c a menšie za $c-k$, tým pádom ich rozdiel je hodnota k. Pre túto dvojicu teda existuje iba jediné číslo $c+k$, ktoré vyhovuje podmienke zadania.
V predchádzajúcom riešení sme teda pre každú dvojicu čísel skúšali možno až n možností, pričom sme vedeli, že správna môže byť nanajvýš jedna. Čo je teda našou úlohou? Pre každú dvojicu čísel $c-k$ a c zistiť, či sa medzi zadanými číslami nachádza číslo $c+k$.
Hľadanie konkrétneho čísla v zadanej množine je pomerne známy problém, ktorý sa dá riešiť v zložitosti $O(\log n)$. Dva najčastejšie spôsoby sú použitie vopred naprogramovaných binárnych vyhľadávacích stromov (napr. `set` v C++) alebo binárneho vyhľadávania. V našom riešení sa zameriame na ten druhý spôsob, ak ste sa však rozhodli použiť prvý z nich, je to rovnako dobré.
Myšlienka binárneho vyhľadávania je jednoduchá. Predstavme si, že máme vzostupne usporiadanú postupnosť čísel a chceme zistiť, či sa medzi nimi nachádza číslo x. Pozrieme sa na číslo v strede tejto postupnosti, označme si ho y. Keďže postupnosť je usporiadaná, tak ak je $y < x$, tak ak sa x v tejto postupnosti nachádza, musí sa nachádzať v jej druhej polovici. Naopak, ak $y > x$, tak x sa môže nachádzať iba v prvej polovici tejto postupnosti. Tak či tak, môžeme vyškrtnúť aspoň polovicu čísel.
To čo nám ostane je opäť usporiadaná postupnosť, tentokrát však polovičnej veľkosti. A na ňu vieme aplikovať rovnaký postup. Pozrieme sa na číslo v strede (y) a porovnáme ho s hodnotou x. Podľa toho zistíme, v ktorej polovici tejto postupnosti sa x môže nachádzať a zvyšok zahodíme. Počas tohto procesu buď narazíme na číslo x, alebo postupne vyhodíme všetky čísla, vďaka čomu zistíme, že x sa medzi nimi nenachádzalo.
Keďže naša postupnosť sa v každom kroku skráti na polovicu, tak po najviac $\log n$ krokoch v nej ostane jediný prvok. Z toho vyplýva výsledná časová zložitosť $O(\log n)$. Binárne vyhľadávanie je známe tým, že je náročné na správnu implementáciu. Ak si však dáte pozor na indexy a čo znamenajú, nie je to také zlé. Dobrý trik je pamätať si interval, v ktorom sa x môže nachádzať ako polootvorený interval $\langle zac, kon)$. Následne sa pozriete na stredný prvok, ktorý je na indexe $(zac+kon)/2$ a ľahko zistíte, či je x napravo (vtedy zmeníme hodnotu zac)

alebo naľavo (vtedy zmeníme hodnotu kon). Túto implementáciu si môžete pozrieť aj v nasledujúcom programe. Celková časová zložitosť tohto riešenia je $O(n^2 \log n)$ – pre každú dvojicu čísel binárne vyhľadávame hodnotu $c + k$.

Listing programu (Python)

```
n = int(input())
cisla = [ int(x) for x in input().split() ]
cisla.sort()

vysledok = 0
for prve in range(0, n - 2):
    for druhe in range(prve+1, n - 1):
        c = cisla[druhe]
        k = cisla[druhe] - cisla[prve]
        zac, kon = druhe+1, n    # polootvoreny interval pozicii <zac, kon), v ktorom sa moze nachadzat c+k
        while kon - zac > 1:
            stred = (zac + kon) // 2
            if cisla[stred] > c + k:
                kon = stred
            else:
                zac = stred
        if cisla[zac] == c + k:  # ostala nam posledna moznost, kde sa moze nachdzat prvok c+k
            vysledok += 1
print(vysledok)
```

Rovnako bodov ste mohli získať aj za riešenie, ktoré namiesto vyvažovaných stromov (teda usporiadanej množiny) použilo hešovaciu tabuľku (teda neusporiadanú množinu, napr. `set` v Pythone alebo `unordered_set` v C++). Takéto riešenie bude v praxi zväčša nerozlíšiteľné od kvadratického, existujú však vstupy, ktoré ho vedia donútiť bežať výrazne dlhšie (keďže pri hešovaní nastane veľa kolízií).
Šikovnejšou implementáciou by sme vedeli dostať riešenie využívajúce hešovanie, ktorého časová zložitosť na *ľubovoľnom* vstupe bude *s veľkou pravdepodobnosťou* kvadratická. Aj od takéhoto riešenia však existujú ešte lepšie.

Vzorové riešenie

Predstavme si, že sme si už zvolili index j, na ktorom leží číslo c. Teraz hľadáme všetky dvojice indexov (i, k) také, že hodnota na indexe i a hodnota na indexe k sú presne rovnako ďaleko od c. Toto vieme ľahko spraviť v lineárnom čase. Začneme tým, že nastavíme $i = j - 1$ a $k = j + 1$. No a teraz už len dokola opakujeme: ak je jedna z hodnôt bližšie ku c ako druhá, posunieme len jej index (zmenšíme i alebo zväčšíme k), a ak sú obe hodnoty rovnako ďaleko od c, poznačíme si, že sme našli vyhovujúcu trojicu, a následne o jedno posunieme oba indexy.
Ľubovoľné konkrétne j spracujeme v lineárnom čase, keďže indexmi i a k dokopy raz prejdeme celú postupnosť hodnôt. Postupné vyskúšanie všetkých j teda dokopy zaberie čas $O(n^2)$.

Listing programu (Python)

```
n = int(input())
cisla = [ int(x) for x in input().split() ]
cisla.sort()

vysledok = 0
for j in range(1, n-1):
    i, k = j-1, j+1
    while i >= 0 and k < n:
        if cisla[j]-cisla[i] < cisla[k]-cisla[j]:
            i = i-1
        elif cisla[j]-cisla[i] > cisla[k]-cisla[j]:
            k = k+1
        else:
            vysledok, i, k = vysledok+1, i-1, k+1

print(vysledok)
```

Iné vzorové riešenie

V tomto vzorovom riešení zoberieme vyššie popísané riešenie, ktoré používalo binárne vyhľadávanie, a upravíme ho, aby malo len kvadratickú časovú zložitosť.

V tomto riešení sme si zvolili prvé dve čísla z trojice ($a_x = c - k$ a $a_y = c$), z nich sme vypočítali potrebnú hodnotu tretieho ($c + k = 2a_y - a_x$) a potom sme použili binárne vyhľadávanie na zistenie, či takéto číslo v poli máme.
Práve tento posledný krok však vieme spraviť aj šikovnejšie. Predstavme si, že ku danej dvojici indexov x a y sme už našli index z na ktorom je hľadané tretie číslo (resp. prvé od neho väčšie, ak hľadanú hodnotu nemáme). V ďalšom kroku bude náš algoritmus skúšať novú dvojicu indexov: zväčša to bude to isté x a nové $y' = y + 1$. Kde teraz hľadať nový index z'?
Keďže sa nezmenilo x a zväčšilo y, ľahko si zdôvodníme, že sa zväčšila aj hodnota c, aj hodnota k. Zväčšiť sa preto musí aj hodnota $c + k$. To ale znamená, že pre nový index z' musí platiť $z' \geq z$.
Namiesto binárneho vyhľadávania preto stačí zobrať starý index z a posúvať ho doprava, kým nenájdeme správne nové miesto z'.
Zhrňme si náš algoritmus. Postupne si budeme vyberať prvé číslo z našej trojice – index x. Pre každé takéto číslo budeme postupne skúšať všetky možnosti pre index y. Vždy, keď zväčšíme y, posunieme doprava aj index z o toľko, o koľko práve treba. No a vždy, keď indexom z naozaj nájdeme hľadanú tretiu hodnotu do trojice, zväčšíme si počet nájdených riešení.
Ostáva určiť časovú zložitosť tohto riešenia. Ak si zoberieme jedno konkrétne x, tak obe hodnoty y a z sa celý čas iba zväčšujú. Každá z nich sa pritom môže zväčšiť najviac n-krát. No a máme n rôznych hodnôt čísla x, ktoré musíme vyskúšať. Výsledná časová zložitosť nášho riešenia je preto $O(n^2)$.

Listing programu (Python)

```
n = int(input())
cisla = [ int(x) for x in input().split() ]
cisla.sort()

vysledok = 0
for x in range(0, n - 2):
    z = x + 2
    for y in range(x+1, n-1):
        c = cisla[y]
        k = cisla[y] - cisla[x]
        hladam = c+k
        while z < n and cisla[z] < hladam:
            z += 1
        if z < n and cisla[z] == hladam:
            vysledok += 1
print(vysledok)
```

B-II-2 Zoznamovačka

Súťažnú úlohu si môžeme preformulovať nasledovne. Na vstupe máme n intervalov určených začiatkom a koncom. Pre každý interval chceme vypočítať, s koľkými inými intervalmi sa aspoň čiastočne prekrýva. Tak ako aj pri zvyšných úlohách, môžeme začať najľahším riešením – skúšaním všetkých možností. V našom prípade sa pre každý interval pozrieme na všetky ostatné a zistíme, ktoré sa s ním prekrývajú.
(Dva intervaly sa prekrývajú vtedy a len vtedy, ak platí, že najskôr oba začnú a až potom oba skončia. Formálne, $\langle z_1, k_1 \rangle$ a $\langle z_2, k_2 \rangle$ sa prekrývajú práve vtedy, ak $\max(z_1, z_2) \leq \min(k_1, k_2)$. Iná, ekvivalentná podmienka je, že začiatok niektorého z nich leží v druhom z nich.)
Toto jednoduché riešenie má časovú zložitosť $O(n^2)$ a mohli sme zaň získať až 5 bodov.

Listing programu (Python)

```
n = int(input())
intervaly = []
for i in range(n):
    z, k = map(int, input().split())
    intervaly.append((z, k))

for interval1 in intervaly:
    pocet = 0
    for interval2 in intervaly:
        if interval1[0] < interval2[0] < interval1[1] or interval2[0] < interval1[0] < interval2[1]:
            pocet += 1
    print(pocet)
```

Obrátená úloha

Mohli by sme sa ďalej snažiť riešiť úlohu zo zadania. Pravdpodobne by sme však k ničomu lepšiemu neprišli. Možno je preto treba riešiť niečo iné. Ale čo? V konečnom dôsledku musíme aj tak vypočítať pre každý interval s koľkými inými intervalmi sa prekrýva. Nemusíme to však spraviť priamo. Túto hodnotu totiž vieme ľahko zistiť aj z toho, **s koľkými intervalmi sa neprekrýva**. A prekvapivo, vyriešiť túto obrátenú úlohu je oveľa ľahšie.
Poďme sa teda chvíľu venovať tejto opačnej úlohe – pre každý interval chceme vypočítať, s koľkými intervalmi sa neprekrýva. Ak potom toto číslo odčítame od $n-1$, tak zistíme aj výsledok. Zoberme si interval $\langle z, k\rangle$. S týmto intervalom sa neprekrývajú tie intervaly, ktoré skončili skôr ako on začal alebo začali neskôr ako on skončil.
Rozoberme si bližšie prvú časť – ukážeme si, ako pre každý začiatok vypočítame počet skorších koncov.
Spravíme si pole dĺžky $2n$ do ktorého si uložíme všetky začiatky a konce intervalov. Toto pole usporiadame. (Pripomíname, že všetky začiatky a konce sú navzájom rôzne, takže nemusíme ošetrovať situáciu, kedy by sme mali viacero začiatkov a koncov na tej istej súradnici.)
No a teraz už len stačí raz prejsť celé pole zľava doprava. Keď stretneme koniec nejakého intervalu, zväčšíme si počítadlo intervalov, ktoré už skončili. A keď stretneme začiatok, zaznačíme si pre tento interval, koľko intervalov skončilo pred ním.
Druhú časť riešenia spravíme presne rovnako, len pôjdeme cez naše pole sprava doľava a budeme si počítať, koľko začiatkov sme už videli.
Časová zložitosť tohto riešenia je $O(n \log n)$ kvôli triedeniu. Následné dva prechody poľom zjavne bežia v lineárnom čase.

Listing programu (Python)

```
ZACIATOK, KONIEC = +1, -1

body = []  # pole obsahujúce trojice (súradnica, číslo intervalu, typ bodu: začiatok/koniec)

N = int(input())

for n in range(N):
    z, k = [ int(_) for _ in input().split() ]
    body.append( (z, n, ZACIATOK) )
    body.append( (k, n, KONIEC) )

body.sort()

vysledky = [ N-1 for n in range(N) ]

# pre každý interval zistíme počet tých čo skončia skôr
skoncilo = 0
for kde, kto, typ in body:
    if typ == ZACIATOK:
        vysledky[kto] -= skoncilo
    else:
        skoncilo += 1

# pre každý interval zistíme počet tých čo začnú neskôr
zacalo = 0
for kde, kto, typ in body[::-1]:
    if typ == ZACIATOK:
        zacalo += 1
    else:
        vysledky[kto] -= zacalo

print(*vysledky)
```

B-II-3 Stolná hra

Konkrétnemu rozmiestneniu figúrok na hracom pláne budeme hovoriť *stav*.
Dokázať, že sa Jonášova hra časom musí zacykliť, je v skutočnosti až prekvapivo jednoduché. Stačí si uvedomiť dve skutočnosti:

- **Stavov je len konečne veľa.** Teoreticky je ich nanajvýš n^n, keďže každá z n očíslovaných figúrok môže byť na ľubovoľnom z n políčok. Hra však beží do nekonečna, a tak sa časom (nanajvýš v čase n^n, možno

skôr) **musí nejaký stav zopakovať**.

- **Každý ďalší stav je jednoznačne určený predchádzajúcim stavom.** A teda akonáhle sa nám nejaký stav zopakuje, bude sa opakovať aj postupnosť stavov, ktoré po ňom budú nasledovať. (Takže ak napr. stav v čase 7 je rovnaký ako stav v čase 3, tak aj stav v čase 8 bude rovnaký ako stav v čase 4, a tak ďalej. Stavy sa nám teda začnú opakovať s periódou dĺžky 4.)

Hľadanie periódy pomaly

Pre ľubovoľný proces, v ktorom je každý ďalší stav jednoznačne určený predchádzajúcim stavom, vieme periódu nájsť hrubou silou: jednoducho postupne generujeme stavy a ukladáme si ich do množiny, až kým sa nám prvýkrát nestane, že vygenerujeme stav, ktorý sme už niekedy predtým videli. Časová zložitosť takéhoto algoritmu je priamo úmerná počtu krokov, ktoré proces spraví kým sa zacyklí – a teda ju vieme zhora odhadnúť počtom stavov. Presnejšie, časová zložitosť bude súčinom dvoch zložiek: počtu krokov a času potrebného na výpočet jedného kroku procesu.
Existujú rôzne algoritmy, ktoré tento istý problém riešia s rovnakou časovou zložitosťou ale vystačia si s maličkým množstvom pamäte – stačí im dokonca len konštantne veľa stavov. Dočítate sa o nich napríklad tu: https://en.wikipedia.org/wiki/Cycle_detection. My nebudeme zachádzať do detailov, keďže v našom prípade máme aj omnoho efektívnejšie riešenie.

Hľadanie periódy rýchlo

Pozrime sa na jednu konkrétnu figúrku. Ako sa bude hýbať? Postupne bude skákať po nejakých políčkach, až kým sa prvýkrát nejaké políčko nezopakuje. Od tejto chvíle už bude táto figúrka dokola skákať po nejakom cykle z políčok.
Každá figúrka má teda nejakú predperiódu (kroky kým príde na cyklus) a potom nejakú periódu. Pre konkrétnu figúrku vieme priamočiarou simuláciou v čase $O(n)$ zistiť dĺžku jej predperiódy a jej periódy.
Teraz sa opäť pozrime na celú hru naraz. Kedy prvýkrát nastane stav, ktorý už bude patriť do periódy celej hry? Pomerne rýchlo: v čase, ktorý je rovný maximu všetkých dĺžok predperiód. Inými slovami, stane sa tak v okamihu, kedy už každá figúrka prišla na svoj cyklus. Predchádzajúce stavy sa zjavne zopakovať nemôžu.
No a po koľkých krokoch sa tento stav prvýkrát zopakuje? To je tiež jednoduché: každá figúrka sa musí vrátiť na tú pozíciu na svojom cykle, kde sa práve nachádza. Inými slovami, hľadaný počet krokov je najmenším spoločným násobkom všetkých dĺžok cyklov.

Hľadanie periódy veľmi rýchlo

Vyššie popísané riešenie by malo časovú zložitosť $O(n^2)$. Jeho najpomalšou časťou je hľadanie jednotlivých predperiód a periód, keďže ho robíme pre každú figúrku zvlášť. Ak však budeme o čosi šikovnejší, vieme celý tento výpočet spraviť v lineárnom čase. Ako na to?
Pre každé *políčko* chceme spočítať dĺžku predperiódy a periódy pre figúrku, ktorá na ňom začína. Spracovanie konkrétneho políčka P teraz bude vyzerať nasledovne: začneme od neho simulovať pohyb figúrky, kým sa nám buď nezacyklí (toto je rovnaké ako v minulom riešení) *alebo kým nenarazíme na políčko, ktoré sme už spracovali skôr*.
Ak pri simulácii narazíme na políčko Q, ktoré sme už spracovali skôr, stačí sa pozrieť na to, aká dĺžka predperiódy a periódy zodpovedá jemu. Políčku P zjavne zodpovedá presne tá istá perióda. No a predperióda pre P je tvorená krokmi, ktoré nás z P doviedli do Q a za nimi nasleduje predperióda políčka Q.
No a najdôležitejšie je si teraz uvedomiť, že sme práve úspešne našli dĺžku predperiódy a periódy nie len pre samotné políčko P, ale taktiež pre všetky políčka, cez ktoré sme prechádzali cestou z P na Q. Všetky tieto políčka si teraz označíme ako spracované a ku každému z nich si zapíšeme ich dĺžku periódy (tú majú všetky rovnakú) a ich dĺžku predperiódy (tá postupne rastie smerom od Q ku P).
Podobný záver môžeme spraviť aj v prípade, že objavíme nový cyklus – len si musíme dať pozor na to, že všetkým políčkam na cykle chceme nastaviť dĺžku predperiódy rovnú nule.
Celé riešenie teraz vyzerá nasledovne:

1. Postupne pre každé i od 1 po n: ak políčko i ešte nebolo spracované, spustíme z neho vyššie popísaný proces. Tým spracujeme nejaké políčka vrátane políčka i.

2. Nájdeme maximum všetkých predperiód a najmenší spoločný násobok všetkých periód.

Prvá časť riešenia má dokopy lineárnu časovú zložitosť, keďže každé políčko spracujeme len raz.
Druhá časť riešenia má (za predpokladu, že sa nám výsledok zmestí do bežnej číselnej premennej) časovú zložitosť $O(n \log n)$, lebo potrebujeme vypočítať najmenší spoločný násobok n čísel, ktoré sú všetky veľkosti do n. Tento výpočet zahŕňa $n-1$ volaní funkcie na výpočet najväčšieho spoločného deliteľa, ktorá beží v čase $O(\log n)$. Toto je teda aj celková časová zložitosť nášho riešenia.

Listing programu (Python)

```
# Euklidov algoritmus na výpočet najväčšieho spoločného deliteľa

def nsd(x,y):
    while y: x,y = y,x%y
    return x

# načítame vstup a prečíslujeme si políčka od 0 po N-1

N = int( input() )
P = [ int(_)-1 for _ in input().split() ]

# postupne preskúmame všetky políčka

predperioda = [ None for n in range(N) ]
perioda     = [ None for n in range(N) ]
poradie     = [ None for n in range(N) ]

for n in range(N):
    if perioda[n] is not None: continue
    navstivil  = [n]
    poradie[n] = 0
    for krok in range(1,n+47):
        kde = P[ navstivil[-1] ]
        if perioda[kde] is not None:
            # prišli sme na skôr preskúmané políčko
            for i in range(krok):
                predperioda[ navstivil[i] ] = predperioda[kde] + krok - i
                perioda[ navstivil[i] ]     = perioda[kde]
            break
        if poradie[kde] is not None:
            # našli sme nový cyklus
            dlzka_cyklu = krok - poradie[kde]
            for i in range(krok):
                predperioda[ navstivil[i] ] = max(0, krok-i-dlzka_cyklu)
                perioda[ navstivil[i] ]     = dlzka_cyklu
            break
        navstivil.append(kde)
        poradie[kde] = krok

# vypočítame, kedy sa stav figúrok prvýkrát zopakuje

nsn = 1
for p in perioda: nsn = nsn * p // nsd(nsn,p)
print( max(predperioda) + nsn )
```

B-II-4 Reálne čísla

Jedna sedmina

V dvojkovej sústave môžeme deliť presne rovnako ako v desiatkovej: postupne získavame cifry výsledku a priebežne si udržiavame zvyšok po delení.

```
 1 : 111 = 0.001...
-0
--
 10
 -0
 --
 100
  -0
 ---
 1000
 -111
 ----
    1
```

Ak vydržíme dostatočne dlho, všimneme si, že sa nám zvyšky opakujú: dokola sa strieda zvyšok 1, 2 a 4 (teda 1_2, 10_2 a 100_2). A teda vieme, že číslo $1/7$ má v dvojkovej sústave periodický zápis s periódou dĺžky 3: jedna sedmina zapísaná v dvojkovej sústave je $0.\overline{001}_2$.
Iné možné riešenie je založené na pozorovaní, že sa vlastne snažíme zapísať jednu sedminu ako súčet rôznych zlomkov tvaru $1/2^k$. Postupne môžeme počítať:

$$\begin{aligned}\frac{1}{7} &= \frac{1}{8} + \frac{1}{7\cdot 8} \\ &= \frac{1}{8} + \frac{1}{8\cdot 8} + \frac{1}{7\cdot 8\cdot 8} \\ &= \frac{1}{8} + \frac{1}{8\cdot 8} + \frac{1}{8\cdot 8\cdot 8} + \frac{1}{7\cdot 8\cdot 8\cdot 8} \\ &= \cdots\end{aligned}$$

Dostávame teda, že $1/7 = 1/2^3 + 1/2^6 + 1/2^9 + \cdots$, čo opäť vieme v dvojkovej sústave zapísať ako $0.001001001\ldots$

Potešíme sa teda z toho, že nám aj druhým postupom vyšiel ten istý výsledok, a pôjdeme sa pozrieť na druhú časť tejto podúlohy: čo sa teraz stane, keď túto hodnotu uložíme do premennej typu single?
Nasledujme príklad zo študijného textu, kde sme si do premennej ukladali číslo 1.23456789_{10}. Začneme tým, že si jednu sedminu zapíšeme v tvare s exponentom a mantisou:

$$\frac{1}{7} = 2^{-3} \times 1.00100100100100100100100100\ldots_2$$

Do premennej typu single sa nám zmestí prvých 23 bitov za binárnou bodkou. A keďže nasledujúci bit jednej sedminy je 1, pri ukladaní číslo zaokrúhlime nahor. Presná hodnota uložená v pamäti bude teda

$$x = 2^{-3} \times 1.\underbrace{00100100100100100100101}_{23 \text{ bitov}}{}_2 = 0.00100100100100100100100101_2$$

Pre zaujímavosť, v desiatkovej sústave má x presnú hodnotu 0.14285714924335479736328125. (Toto samozrejme nebolo potrebné dopočítať.)

„Nekonečný“ cyklus

V riešeniach domáceho kola sme sa zaoberali otázkou, či musí vždy platiť $(a+b)+c = a+(b+c)$. Ukázali sme si, že to nemusí byť pravda. A presnejšie, ukázali sme si to tak, že sme si ukázali, že pre dostatočne veľké y platí $y = y+1$, lebo keď vypočítame presnú hodnotu $y+1$ a uložíme ju späť do premennej, zaokrúhli sa nám táto naspäť na y.
Presne toto bude aj dôvodom, kvôli ktorému cyklus zo zadania nebude nekonečný. Akonáhle y narastie na dostatočne veľkú hodnotu, už sa ho nepodarí zväčšiť o 1, a tým cyklus skončí. Kedy presne sa tak stane?
Všetky čísla po $2^{24}-1$ vrátane si vieme v premennej typu single reprezentovať presne, keďže nám ešte mantisa stačí na všetky ich bity. Posledné z týchto čísel má v pamäti tvar $2^{23} \times 1.11111111111111111111111$.
Číslo 2^{24} si tiež vieme reprezentovať presne: je to $2^{24} \times 1.000\ldots$.
No a všetko sa pokazí v okamihu, keď k tomuto číslu skúsime pripočítať ďalšiu jednotku. Tá už je teraz v ráde, ktorý je tesne za tými, čo sa nám zmestia do mantisy. Presná hodnota čísla $2^{24}+1$ je teda presne uprostred medzi dvomi hodnotami, ktoré v premennej typu single vieme reprezentovať: hodnotou 2^{24} a hodnotou $2^{24}+2$.
No a ako sme si v študijnom texte povedali, v takejto situácii zaokrúhlime výsledok na okrúhlejšiu z oboch možností, a tou je práve číslo 2^{24}. Cyklus sa teda zastaví na hodnote $y = 2^{24} = 16\,777\,216$.

TRIDSIATY TRETÍ ROČNÍK OLYMPIÁDY V INFORMATIKE

Príprava úloh: Michal Anderle, Michal Forišek
Recenzia: Michal Forišek
Slovenská komisia Olympiády v informatike
Vydal: IUVENTA – Slovenský inštitút mládeže, Bratislava 2018