B-I-1 Babičkina špajza

Nejaké body sa dali ľahko získať za riešenie „hrubou silou“: jednoducho vyskúšame všetky trojice fliaš. Toto sa dalo robiť dvoma spôsobmi: buď vygenerujeme úplne všetky trojice a pre každú skontrolujeme, či nie sú dve fľaše príliš blízko seba...

```
for a in range(N):
    for b in range(N):
        for c in range(N):
            if abs(a-b) <= K: continue
            if abs(a-c) <= K: continue
            if abs(b-c) <= K: continue
            riesenie = max( riesenie, V[a]+V[b]+V[c] )
```

... alebo rovno vygenerujeme a skontrolujeme len správne trojice indexov a, b, c – také, v ktorých je a dostatočne menšie od b a b dostatočne menšie od c.

```
for a in range(N):
    for b in range(a+K+1, N):
        for c in range(b+K+1, N):
            riesenie = max( riesenie, V[a]+V[b]+V[c] )
```

Obe tieto riešenia majú kubickú časovú zložitosť – čas behu programu rastie priamo úmerne tretej mocnine počtu fliaš. Formálne píšeme, že časová zložitosť je $O(n^3)$.

Kvadratické riešenie

Lepšie riešenie vieme založiť napríklad na skúšaní fliaš v inom poradí. Môžeme si totiž všimnúť, že keď si v predchádzajúcom riešení zvolíme index b (teda číslo prostrednej fľaše, ktorú chceme zobrať), môžeme a a c zvoliť nezávisle od seba: a bude vždy číslo najlepšej fľaše s dostatočne menším a c číslo najlepšej fľaše s dostatočne väčším indexom.
Pre každé b teda dokopy spravíme pri hľadaní optimálneho a a c len lineárne veľa krokov výpočtu, keďže každú fľašu inú od b skontrolujeme nanajvýš raz.

Lineárne riešenie

Predchádzajúce riešenie vieme ešte ďalej vylepšiť, a to tak, že vhodné časti z neho predpočítame.
Predstavme si napríklad, že máme $k = 10$ a práve sme vyskúšali $b = 101$. Počas tohto skúšania sme prezreli fľaše s číslami 0 až 90 a zistili sme, ktorá z nich je najlepšou fľašou a. Keď sa teraz posunieme na $b = 102$, opäť budeme hľadať najlepšiu fľašu a. Treba znova prezerať všetky fľaše s číslami 0 až 91? Zjavne nie. Stačí porovnať najlepšiu fľašu v úseku 0-90 s jedinou novou fľašou číslo 91. Takto sa nové optimálne a dozvieme v konštantnom čase.
Celé riešenie si teda môžeme rozdeliť do troch krokov:

1. V lineárnom čase prejdeme pole objemov zľava doprava. Počas tohto prechodu si pre každé i zistíme najlepšiu fľašu s číslom $\leq i$.
2. V lineárnom čase znova prejdeme pole objemov, ale tentokrát sprava doľava. Počas tohto prechodu si pre každé i zistíme najlepšiu fľašu s číslom $\geq i$.
3. Tretíkrát prejdeme celé pole. Pri tomto prechode postupne vyskúšame všetky možnosti pre index b. Pre každú možnosť použijeme predpočítané informácie na to, aby sme v konštantnom čase našli najlepšie a a c zodpovedajúce tomuto b.

Listing programu (Python)

```
# načítame vstup

N, K = [ int(_) for _ in input().split() ]
V    = [ int(_) for _ in input().split() ]

# predpočítame si pre každé i:
# - aká najlepšia fľaša sa nachádza medzi fľašami s číslom <= i
# - aká najlepšia fľaša sa nachádza medzi fľašami s číslom >= i

nalavo  = [ None for n in range(N) ]
napravo = [ None for n in range(N) ]
```

```
nalavo[0] = V[0]
for n in range(1,N): nalavo[n] = max( nalavo[n-1], V[n] )

napravo[N-1] = V[N-1]
for n in range(N-2,-1,-1): napravo[n] = max( napravo[n+1], V[n] )

# postupne vyskúšame všetky možnosti, ktorú strednú fľašu vyberieme
# ku každej zoberieme najlepšiu dostatočne naľavo a najlepšiu dostatočne napravo

riesenie = 0
for stred in range(K+1,N-K-1):
    riesenie = max( riesenie, nalavo[stred-K-1] + V[stred] + napravo[stred+K+1] )

print(riesenie)
```

Iné lineárne riešenie

Na záver si ukážeme ešte jedno lineárne riešenie, ktoré je síce myšlienkovo trochu zložitejšie, ale na druhej strane je všeobecnejšie – jeho myšlienka by sa dala bez väčších zmien použiť aj ak by Peťka napríklad chcela sedem fliaš namiesto troch.

Základná myšlienka je, že postupne vyriešime zadanú úlohu najskôr pre len jednu fľašu, potom pre dve a na záver pre tri. Presnejšie, postupne pre každý počet fliaš p a každý index i spočítame, koľko najviac džemu vieme mať, ak vyberieme p spomedzi fliaš s indexmi 0 až i. Tento počet si označme napríklad $m_{p,i}$.

Pre $p = 1$ je táto úloha ľahko riešiteľná – urobíme presne to isté, čo v prvom kroku predchádzajúceho riešenia. (Odborne tomu hovoríme, že sme si vypočítali *prefixové maximá* pre zadané pole.)

Ako to bude vyzerať pre väčšie p? Pozrime sa napríklad na výpočet konkrétnej hodnoty $m_{2,i}$. Chceme najlepším možným spôsobom vybrať dve fľaše džemu spomedzi fliaš s číslami 0 až i. Toto riešenie vieme šikovne nájsť tak, že rozoberieme dve možnosti: buď fľašu i zoberieme alebo nie.

Prvá možnosť vedie k tomu, že okrem fľaše i chceme najlepšiu jednu fľašu s číslom $\leq i - k - 1$. A toto už sme spočítali v predchádzajúcom kroku – najlepšia taká fľaša má objem $m_{1,i-k-1}$. No a druhá možnosť vedie k číslu $m_{2,i-1}$, keďže rozhodnutie nepoužiť fľašu i znamená, že chceme najlepším spôsobom vybrať dve fľaše spomedzi tých s číslami 0 až $i - 1$. Ak teda budeme hodnoty $m_{2,i}$ počítať s rastúcim i, pri výpočte konkrétnej z nich už poznáme všetky údaje potrebné na to, aby sme ju zistili v konštantnom čase.

Zvyšné detaily tohto riešenia je najlepšie vidieť priamo na konkrétnej implementácii:

Listing programu (Python)

```
# načítame vstup

N, K = [ int(_) for _ in input().split() ]
V    = [ int(_) for _ in input().split() ]

# pre každé i spočítame, koľko najviac džemu vieme dostať ak zoberieme JEDNU fľašu spomedzi fliaš s číslom <= i

jedna = [ None for n in range(N) ]
jedna[0] = V[0]
for n in range(1,N):
    jedna[n] = max( jedna[n-1], V[n] )

# najlepší spôsob, ako vybrať DVE fľaše spomedzi fliaš s číslom <= i vypočítame ako lepšiu z dvoch možností:
# - fľašu i nepoužijeme, vyberieme dve naľavo od nej
# - fľašu i použijeme a vyberieme k nej najlepšiu jednu naľavo od nej

dve = [ None for n in range(N) ]
dve[K+1] = V[K+1] + jedna[0]
for n in range(K+2,N):
    dve[n] = max( dve[n-1], V[n] + jedna[n-K-1] )

# a teraz to isté pre TRI fľaše

tri = [ None for n in range(N) ]
tri[2*K+2] = V[2*K+2] + dve[K+1]
for n in range(2*K+3,N):
    tri[n] = max( tri[n-1], V[n] + dve[n-K-1] )

print( tri[N-1] )
```

B-I-2 Bicyklový výlet

Na obrázku v zadaní to vyzerá, že máme „dvojrozmernú“ úlohu, keďže v grafe vystupuje aj rýchlosť Jankovho bicykla aj čas. Keď sa však trochu zamyslíme, zistíme, že na čase vlastne vôbec nezáleží. Voľne povedané, keby sme graf ľubovoľne natiahli alebo zúžili vo vodorovnom smere, nič sa tým nezmení, každú rýchlosť Janko nadobudne rovnako veľakrát ako v pôvodnom grafe. Čas teda môžeme úplne zanedbať.
V skutočnosti teda ide o „jednorozmernú“ úlohu. Máme daných n intervalov rýchlosti (jeden pre každú minútu Jankovho výletu) a hľadáme takú rýchlosť v, ktorá leží v čo najviac spomedzi zadaných intervalov.
V našej úlohe ešte vieme, že jednotlivé intervaly na seba nadväzujú (každé dva po sebe idúce majú spoločný koncový bod). Toto pozorovanie však v riešení nijak nevyužijeme – dokážeme totiž efektívne vyriešiť všeobecnejšiu úlohu, v ktorej o intervaloch vôbec nič nepredpokladáme.

Pomalšie riešenia

Označme v_{max} najväčšiu rýchlosť, ktorou Janko išiel niekedy počas výletu. Jedno priamočiare riešenie súťažnej úlohy teraz vyzerá nasledovne: pre každú nepárnu rýchlosť medzi 0 a v_{max} prejdi všetky intervaly a spočítaj, koľko z nich ju obsahuje.
Takéto riešenie má časovú zložitosť $O(nv_{max})$ a nie je veľmi efektívne. To je ale spoločná vlastnosť všetkých podobných riešení, ktorých časová zložitosť závisí od *veľkosti čísel* na vstupe. Veľmi ľahko vieme vyrobiť maličké vstupy na ktorých budú takéto riešenia veľmi pomalé. Tu by napríklad stačilo zvoliť $n = 2$ a $v_1 = 10^{18}$. Napriek tomu, že ide o výlet tvorený len dvomi úsekmi, toto riešenie by optimálnu rýchlosť hľadalo niekoľko tisícročí.

Radšej by sme teda našli nejaké riešenie, ktorého časová zložitosť bude závisieť len od *počtu* čísel na vstupe – v našom prípade teda od počtu intervalov.
Ako na to? Jedno šikovné pozorovanie je, že ak žiaden interval nezačína ani nekončí číslom $2v$, tak nemá zmysel skúšať aj rýchlosť $2v - 1$ aj rýchlosť $2v + 1$, lebo odpoveď pre obe bude rovnaká.
Z toho teraz napríklad vyplýva, že *najmenšia* optimálna rýchlosť je určite o 1 väčšia ako hranica niektorého intervalu rýchlostí. Stačí teda vyskúšať všetky takéto rýchlosti (ktorých je len lineárne veľa) a vybrať najlepšiu z nich. Toto riešenie má časovú zložitosť $O(n^2)$, čiže kvadratickú od počtu intervalov.

Listing programu (Python)

```
def spocitaj(v,N,V): # zistí, koľkokrát mal Janko nepárnu rýchlosť v
    odpoved = 0
    for n in range(N):
        if min( V[n], V[n+1] ) < v < max( V[n], V[n+1] ):
            odpoved += 1
    return odpoved

# načítame vstup

N = int(input())
V = [ int(_) for _ in input().split() ]

# vyskúšame rýchlosti o 1 väčšie ako hranice intervalov

bestv, bestcnt = 0, 0
for v in V:
    cnt = spocitaj(v+1,N,V)
    if cnt > bestcnt: bestv, bestcnt = v+1, cnt

print(bestv)
```

Vzorové riešenie

Ako v mnohých iných úlohách, aj tu platí, že tú istú informáciu sa dá spočítať rôzne efektívne. Aj vo vzorovom riešení v podstate vyskúšame tie isté možnosti ako v práve popísanom kvadratickom riešení, len to budeme robiť šikovnejšie.
Toto riešenie bude založené na technike nazývanej *zametanie*. Zadané intervaly rýchlostí si predstavíme ako úsečky na x-ovej osi, a následne túto os prejdeme zľava doprava, pričom si priebežne budeme udržiavať informáciu o tom, v koľkých intervaloch sa práve nachádzame.

Keď pôjdeme po x-ovej osi zľava doprava, postupne nastane presne $2n$ *udalostí*: n-krát narazíme na začiatok nejakého intervalu a n-krát na koniec nejakého intervalu. No a každá udalosť zodpovedá tomu, že sa o 1 zmení počet intervalov, v ktorých sa nachádzame: ak nový začal, tak o 1 stúpne, ak nejaký v ktorom sme boli skončil, tak počet intervalov, v ktorých sa nachádzame, o 1 klesne.
Riešenie teda začneme tým, že si vygenerujeme zoznam všetkých $2n$ udalostí. Tie si následne usporiadame (od najmenšej rýchlosti po najväčšiu, čiže na x-ovej osi zľava doprava) a v tomto poradí ich všetky postupne spracujeme.
Ostáva už len jediné: Ako nájsť rýchlosť, ktorú Janko nadobudol najviackrát? Jednoducho: vždy, keď nová udalosť zodpovedá inej rýchlosti ako tá predchádzajúca, našli sme jeden interval, v ktorom Janko každú rýchlosť nadobudol rovnako veľakrát. A keďže si tento počet priebežne udržiavame, vieme presne, koľkokrát to je. A ak sme práve zlepšili najlepšie známe riešenie, vyberieme si ako odpoveď ľubovoľnú rýchlosť z práve spracúvaného intervalu.
Najpomalšou časťou tohto riešenia je usporiadanie zoznamu udalostí. To vieme spraviť v čase $O(n \log n)$. Celý zvyšok riešenia potom už beží v lineárnom čase.
V programe (uvedenom nižšie) ešte používame jeden drobný trik: ak jednej rýchlosti zodpovedá viacero udalostí, usporiadame ich tak, aby sme najskôr spracovali všetky konce intervalov a až potom všetky začiatky. Rozmyslite si, ako nám to zjednodušilo implementáciu – kontrolu ktorej podmienky sme potom mohli vynechať a prečo?

Listing programu (Python)

```
def spocitaj(v,N,V): # zistí, koľkokrát mal Janko nepárnu rýchlosť v
    odpoved = 0
    for n in range(N):
        if min( V[n], V[n+1] ) < v < max( V[n], V[n+1] ):
            odpoved += 1
    return odpoved

# načítame vstup

N = int(input())
V = [ int(_) for _ in input().split() ]

# vygenerujeme si a usporiadame zoznam udalostí

udalosti = []
for n in range(N):
    v1, v2 = V[n], V[n+1]
    if v1 > v2: v1, v2 = v2, v1
    udalosti.append( (v1,+1) )
    udalosti.append( (v2,-1) )

udalosti.sort()

# postupne prejdeme zoznam udalostí a nájdeme najlepšiu rýchlosť

bestv, bestcnt, cnt = 0, 0, 0

for v, zmena in udalosti:
    cnt += zmena
    if cnt > bestcnt: bestv, bestcnt = v+1, cnt

print(bestv)
```

B-I-3 Dvanásťminútovka

Toto vzorové riešenie začneme tým, že sa zamyslíme nad súvisom medzi najmenším spoločným násobkom (least common multiple, lcm) a najväčším spoločným deliteľom (greatest common divisor, gcd) dvoch čísel. Až potom si povieme, ako vlastne riešiť zadané úlohy.

Výpočet najmenšieho spoločného násobku

Majme ľubovoľné dve prirodzené čísla a, b. Nech d je ich najväčší spoločný deliteľ. Označme teraz $a' = a/d$ a $b' = b/d$. Keďže d je najväčší, musia čísla a' a b' byť nesúdeliteľné.
Nech teraz nejaké n je násobkom a aj b. Z toho zjavne vyplýva, že n je násobkom d. Označme $n' = n/d$. Keďže n je násobkom $a = a'd$, musí n' byť násobkom a'. A z rovnakého dôvodu musí n' byť aj násobkom b'. No a

keďže a' a b' sú nesúdeliteľné, musí tým pádom n' byť násobkom $a'b'$.
Tým sme dokázali, že n musí byť násobkom čísla $a'b'd$. Ale na druhej strane ľahko nahliadneme, že už samotné číslo $a'b'd$ je násobkom aj čísla a, aj čísla b. Preto pre najmenší spoločný násobok čísel a a b platí $n = a'b'd$.
No a už ostáva len jedno záverečné pozorovanie. Pozrime sa na súčin a a b. Keď si a a b rozpíšeme ako $a'd$ a $b'd$, dostávame, že $ab = a'b'd^2$. Inými slovami, $ab = (a'b'd) \cdot d = nd$.

Túto istú rovnosť si môžeme zapísať aj v podobe $n = ab/d$. Inými slovami, najmenší spoločný násobok čísel a a b vieme nájsť tak, že pomocou Euklidovho algoritmu nájdeme ich najväčšieho spoločného deliteľa d a potom ním vydelíme súčin oboch čísel.

Riešenie podúlohy A

Aby bol v čase t konkrétny žiak na štarte, musí t byť násobkom času t_i, za ktorý tento žiak obehne jedenkrát okruh. Aby boli v čase t na štarte všetci žiaci, musí t byť násobkom všetkých t_i. A keďže hľadáme prvý takýto čas, hľadáme najmenší spoločný násobok všetkých t_i.
Tento vieme počítať postupne: najskôr vypočítame najmenší spoločný násobok t_1 a t_2, z neho potom najmenší spoločný násobok tejto hodnoty a t_3, a tak ďalej.
Keď už spočítame hodnotu t, hodnotu k (celkový počet zabehnutých kôl) zistíme ľahko: žiak i zabehol t/t_i kôl, takže len sčítame tieto hodnoty cez všetky i.
Pre zaujímavosť uvádzame kompletnú implementáciu tohto riešenia, vrátane vlastnej implementácie funkcií *gcd* a *lcm*.

Listing programu (Python)

```
def gcd(x,y): # Euklidov algoritmus pre najväčšieho spoločného deliteľa dvoch čísel
    while y > 0:
        x, y = y, x % y
    return x

def lcm(x,y): # najmenší spoločný násobok dvoch čísel
    return (x // gcd(x,y)) * y

N = int( input() )
T = [ int(_) for _ in input().split() ]

cas = 1
for t in T: cas = lcm( cas, t )
kola = sum( cas//t for t in T )
print( cas, kola )
```

Riešenie podúlohy B

Ak chceme našu úlohu vyriešiť v konštantnej pamäti, potrebujeme vyriešiť len jeden problém: nemôžeme si počet zabehnutých kôl spočítať až na konci, lebo na to by sme potrebovali znova poznať každú z hodnôt t_i. Riešenie je však veľmi jednoduché: budeme si postupne udržiavať aj hodnotu t, aj hodnotu k počas toho, ako zo vstupu po jednom čítame a spracúvame hodnoty t_i.
Na začiatku je to ľahké: ak máme len jedného žiaka, je $t = t_1$ a $k = 1$.
Predstavme si teraz, že už sme spracovali nejakú skupinku žiakov a vieme, že na štarte sa prvýkrát všetci naraz stretnú v čase t a dovtedy zabehnú k kôl. Teraz nám pribudne ďalší žiak, ktorý zabehne okruh v čase t_i. Čo sa stane?
Už vieme, že nová hodnota t (označme ju t') bude najmenším spoločným násobkom t a t_i – na štarte musí byť v čase t' aj nový žiak, aj všetci ostatní žiaci.
Ako sa zmení k? Nové k (označme ho k') vypočítame nasledovne: Vieme, že ostatní žiaci všetci dokopy za čas t zabehli k kôl. Za čas t' (ktorý je násobkom času t) teda dokopy zabehnú $(t'/t)k$ kôl. No a nový žiak do okamihu stretnutia zabehne t'/t_i kôl. Dostávame teda, že $k' = (t'/t)k + (t'/t_i)$.
Takto vieme každého žiaka spracovať v konštantnom čase. Dokopy teda dostávame riešenie v lineárnom čase a konštantnej pamäti. Nižšie uvádzame jeho implementáciu v C++.

Listing programu (C++)

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

typedef unsigned long long cislo;

cislo gcd(cislo x, cislo y) { return (y == 0) ? x : gcd(y,x%y); }
cislo lcm(cislo x, cislo y) { return (x/gcd(x,y))*y; }

int main() {
    int N; cin >> N;
    cislo T = 1, K = 0;
    for (int n=0; n<N; ++n) {
        cislo t; cin >> t;
        cislo newT = lcm(T,t);
        cislo newK = (newT/T) * K + (newT/t);
        T = newT;
        K = newK;
    }
    cout << T << " " << K << endl;
}
```

B-I-4 Reálne čísla

Podúloha A

Sčítanie je skutočne komutatívne – teda pre ľubovoľné a a b skutočne platí, že $a + b = b + a$.
Dôvod je očividný: v oboch prípadoch totiž zoberieme tú istú presnú hodnotu súčtu a toto číslo následne uložíme do premennej. V poslednom kroku síce môže nastať zaokrúhľovacia chyba, no keďže v oboch prípadoch ukladáme presne to isté číslo, nastane nutne presne tá istá chyba, a teda výsledok tiež bude presne ten istý.

Podúloha B

S asociativitou sčítania je to už horšie, presnejšie, vôbec nemusí platiť. Ak sčítavame viacero hodnôt, závisí na poradí, v akom to robíme.
Tu je jednoduchý príklad v Pythone:

```
>>> a = 1
>>> b = float(2**60)
>>> c = -b
>>> print( a + (b + c) )
1.0
>>> print( (a + b) + c )
0.0
```

Kde nastal problém? V tom, že rôzne postupy môžu viesť k rôznemu zaokrúhľovaniu.
Všimnime si vo vyššie uvedenom príklade najskôr možnosť, kde počítame $a + (b + c)$. Keďže $c = -b$, výsledkom operácie $b + c$ je nula. Tú si počítač uloží do pomocnej premennej a následne ju sčíta s premennou a, čím dostane správny výsledok: $1+0 = 1$.
A čo možnosť $(a + b) + c$? Tam začneme tým, že vypočítame $a + b = 2^{60} + 1$. Lenže hodnota 1 je oproti hodnote 2^{60} zanedbateľne malá. Presná hodnota výsledku by bola

$$1.\underbrace{00\ldots00}_{59\text{ núl}}1 \times 2^{60}$$

Lenže, ako vieme, pri ukladaní reálnych čísel v double precision formáte máme 52-bitovú mantisu. Uložiť si teda vieme len 52 cifier nasledujúcich za „1.“. Pripočítanie jednotky sa teda na takto veľkom výsledku vôbec neprejaví, uložením do premennej sa hodnota zaokrúhli späť na pôvodných 2^{60}. Výsledok sčítania $a + b$ bude teda v premennej typu double vyzerať nasledovne:

$$1.\underbrace{00\ldots00}_{52\text{ núl}} \times 2^{60}$$

No a z toho už je zjavné, že výsledkom sčítania $(a + b) + c$ bude nula.

Akonáhle sčítujeme viac „reálnych“ čísel, chyby sa nám rýchlo môžu nazbierať. Predstavte si napríklad, že máme pole single precision čísel, v ktorom je najskôr 10^8 jednotiek a potom jedna 10^8. Ak ho sčítame od najmenších hodnôt po najväčšie, dostaneme správny výsledok 2×10^8. Ak ho ale sčítame v opačnom poradí, dostaneme veľmi výrazne nesprávny výsledok – len 10^8.

Podúloha C

Laik by očakával, že sa program spustí, presne desaťkrát vypíše hviezdičku a skončí. Kto si ale program skúsil spustiť, zistil, že neskončí – spokojne si beží a vypisuje hviezdičky, až kým ho my nezastavíme. V čom je problém? Pes je samozrejme zakopaný v tom, že hodnotu 0.1 nevieme v dvojkovej sústave reprezentovať presne, keďže jej binárny rozvoj je nekonečný: $0.1_{10} = 0.0\overline{0011}_2$. A teda keď do premennej uložíme hodnotu 0.1, zaokrúhli sa tým o trochu. No a keď túto zaokrúhlenú hodnotu $10\times$ nasčítavame, táto zaokrúhľovacia chyba sa prejaví aj na výsledku. Po desiatich iteráciách cyklu teda v premennej x nemáme presnú hodnotu 1, ale hodnotu, ktorá sa od 1 v posledných pár bitoch líši.
(Ak sa chcete dozvedieť viac, dajte si vypísať presnú hodnotu premennej `x` po každej iterácii cyklu a pozrite sa na to, ako presne sa mení. Pozor, nestačí použiť štandardný spôsob vypisovania, treba explicitne povedať, že chcete veľa desatinných miest – napríklad 30.)
A hlavné poučenie, ktoré z tejto podúlohy pre nás vyplýva? Výpočty s reálnymi číslami sú nepresné, preto pri písaní našich programov nikdy nemôžeme používať obyčajné `==` (operátor porovnania) na testovanie, či sa dve reálne čísla rovnajú. Namiesto toho býva zvykom pri testovaní rovnosti povoliť určitú toleranciu, teda za rovnaké považovať reálne čísla vtedy, keď sa líšia len o zanedbateľne malú hodnotu.

Poznámky na záver

Realita je ešte o čosi komplikovanejšia ako to, čo sme si popísali v zadaní a v tomto vzorovom riešení.
Problémy s asociativitou spôsobuje napríklad aj pretečenie – teda sčítanie prestane byť asociatívne aj vtedy, keď sa priblížime k hraniciam rozsahu čísel reprezentovateľných v danom type premennej. Ak napríklad používame typ double, problémy nám spôsobia aj premenné s hodnotou $a = b = 2^{1023}$ a $c = -a$. Pre ne korektne dostaneme, že $a+(b+c)$ je rovné a. Lenže keď počítame $(a+b)+c$, tak hodnota $a+b = 2^{1024}$ je priveľká – v premennej typu double už nemáme dosť miesta na zápis jej exponentu. Výsledkom tohto sčítania je preto špeciálna hodnota `+inf` predstavujúca kladné nekonečno. No a pre túto hodnotu už akékoľvek ďalšie sčítanie s konečným číslom vráti opäť `+inf`.
Ďalším zdrojom problémov je skutočnosť, že štandard IEEE 754, ktorý popisuje, ako sa majú správať reálne čísla v programoch, v skutočnosti napríklad dovoľuje, že program si medzivýsledky môže počítať s väčšou presnosťou ako programátor špecifikoval. A toto sa v praxi aj naozaj deje: napríklad mnohé bežné procesory majú až 10-bajtové registre na prácu s reálnymi číslami. A naozaj sa niektoré výpočty v programoch počítajú až s takouto presnosťou – ale to, ktoré to budú, záleží na rozhodnutí kompilátora. A tak sa vám pokojne môže stať, že na dvoch rôznych počítačoch program beží rôzne, lebo sa na výpočty používa iná presnosť a hodnoty sa pri nej iným smerom zaokrúhlia. Rôzne zaokrúhlené výsledky môžete dokonca dostať aj vtedy, ak na dvoch rôznych miestach svojho programu použijete úplne presne identickú postupnosť príkazov. Pri práci s reálnymi číslami *naozaj vždy* treba rátať s drobnými zaokrúhľovacími chybami.

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 2017