PRASK
IV. ročník, 2017/18
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

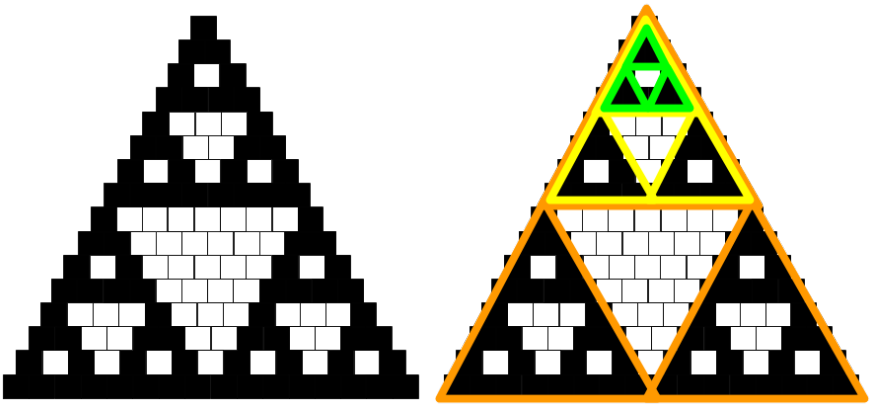
Vzorové riešenia 1. kola zimnej časti

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

1. Premiestnení faraóni

Prvá vec, ktorá by nám mohla pri riešení tejto úlohy pomôcť je, že si nakreslíme o niečo väčšiu pyramídu a pozrieme sa, ako vyzerá. Miesta, kde sa nachádzajú faraóni si budeme značiť čiernym štvorčekom a miesta kde nie sú bielym.

Zopakujme si najskôr pravidlá pre stavbu pyramídy. V najvyššej miestnosti sa musí nachádzať faraón. Následne, miestnosť pod dvoma obsadenými a tiež dvoma prázdnymi izbami musí byť prázdna. Sarkofág s faraónom teda budeme ukladať iba do izieb, ktoré majú nad sebou **jednu plnú** a **jednu prázdnu** izbu. Všimnite si tiež, že okolie pyramídy si môžeme predstaviť ako prázdne izby a napĺňanie miestností bude bez problémov fungovať.



Podúloha a.

Keď sa pozrieme na obrázok, môžeme si všimnúť niekoľko vecí. Jedno z najľahších pozorovaní je, že na n-tom poschodí sa nachádza presne n izieb. Ako oveľa zaujímavejšie sa však javia vzory, ktoré sa nám v našej pyramíde vytvorili. Presnejšie, pod každým poschodím, ktoré je vyplnené faraónmi, sa nachádzajú dve kópie doterajšej pyramídy – jedna naľavo a jedna napravo. A medzi týmito dvoma kópiami je prázdny priestor. Toto si môžete všimnúť jednak na riadku 4 a riadku 8.

Samozrejme, obrázok so 16 poschodiami nie je dôkaz toho, že to tak musí byť. Dáva nám to však najavo, že ideme dobrým smerom.

Začnime tým, že si zoberieme jedno poschodie, ktoré je vyplnené sarkofágmi. Ako vyzerá poschodie pod ním? Keďže pod dvoma susednými sarkofágmi nemôže ležať tretí, všetky miestnosti, okrem dvoch krajných budú prázdne. No a tieto dve krajné miestnosti by mali tvoriť vrch našich dvoch nových pyramíd.

A naozaj, keď si zoberieme jedno z nich, tak vedľa neho sú prázdne miestnosti a preto budú izby pod ním vyzerať ako v druhom riadku celej pyramídy. A opäť, na jednej strane tohto ďalšieho riadku je okraj pyramídy a na druhom prázdne izby (pretože pod dvoma prázdnymi izbami je opäť prázdna izba) a miestnosti pod ním sa vyplnia rovnako ako v treťom riadku pyramídy.

Keby nám takto vznikala len jedna pyramídy, tak jej nič nebráni, aby vyzerala rovnako ako tá nad ňou. Nám sa však takto tvoria súčasne dve pyramídy. Kým sú ďaleko od seba, oddelené prázdnymi izbami, tak sa nijak neovplyvňujú. Ak však medzi nimi prázdne izby nebudú, tieto dve menšie pyramídy sa začnú ovplyvňovať.

Už z obrázka je jasné, že tieto dve pyramídy sa skutočne k sebe približujú. Kedy sa však stretnú? Uvedomme si, že každá pyramída sa v ďalšom riadku zväčší o 1 políčko. A obe menšie pyramídy sa postupne (a symetricky) posúvajú ku stredu. Obe sa teda posunú o jedno políčko do stredu, ale zároveň, celá veľká pyramída narastie na danom riadku o jednu miestnosť. Vzájomná vzdialenosť sa preto zmenší o jednu miestnosť.

Čo sa teda stane v našom prípade na poschodí 9 a nižšie? Pyramída, ktorá je tvorená ôsmimi poschodiami má spodné poschodie vyplnené faraónmi. Preto sa na deviatom poschodí začnú tvoriť dve menšie pyramídy. Tieto dve pyramídy na ich prvom poschodí (deviate celkovo) delí 7 $(8-1)$ voľných izieb. Na ich druhom poschodí (desiate celkovo) ich delí o jednu izbu menej, teda 6 $(8-2)$ izieb. To znamená, že sa stretnú na ich ôsmom poschodí (šestnáste celkovo), kde ich vzájomná vzdialenosť bude 0 $(8-8)$ izieb.

A práve ôsme poschodie je predsa celé vyplnené faraónmi. To znamená, že celé šestnáste poschodie obsahuje iba faraónov – obe menšie pyramídy tu majú svoje ôsme poschodie a nie je medzi nimi žiadna prázdna izba.

Túto úvahu teraz vieme zopakovať aj pre pyramídu veľkosti 16. Na riadku 17 sa totiž začnú tvoriť dve pyramídy, ktoré budú oddelené 15-timi prázdnymi izbami. Dotknú sa preto na riadku 32, kde sa stretnú akurát dva riadky číslo 16 menších pyramíd, ktoré obsahujú iba faraónov, čo znamená, že aj riadok 32 obsahuje iba faraónov.

A takto to bude pokračovať aj ďalej, postupne s pyramídami veľkosti 64, 128, 256, 512... Ak si totiž zoberieme nejaký z týchto plných riadkov, tak dve menšie pyramídy, ktoré sa pod ním začnú vytvárať sa stretnú práve na riadku, ktorých je dvakrát väčší. Čísla úplne zaplnených poschodí budú teda v tvare 1, $2 \cdot 1 = 2$, $2 \cdot 2 \cdot 1 = 4$, $2 \cdot 2 \cdot 2 = 8$, $2 \cdot 2 \cdot 2 \cdot 2 = 16$, $2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 32$.

Tieto čísla sú takzvané mocniny dvojky a preto môžeme povedať, že čísla zaplnených poschodí majú tvar 2^n, pre $n \geq 0$.

Podúloha b.

Riešenie podúlohy b. nebolo vôbec ťažké. Napríklad jej prvú polovicu, kde ste mali zistiť zaplnenosť miestnosti na dvadsiatom poschodí ste mohli vyriešiť jednoducho tým, že ste si to nakreslili. Druhú časť by sa vám však už rukou kresliť nechcelo, predsa len vyše 200 riadkov je veľa. Kreslenie našej pyramídy má však vcelku jasné, algoritmické pravidlá. Preto ste si buď mohli napísať krátky program, ktorý vám to vypočítal, poprípade použiť Excel[1].

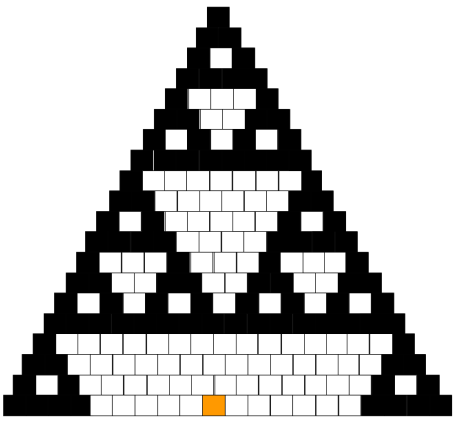
Ak to chceme riešiť pomocou tabuľkového editoru, tak jedinou otázkou ostáva, ako vypočítať číslo, ktoré uložíme do bunky. Začneme tým, že si celú tabuľku zarovnáme doľava. Vtedy bude platiť, že každé políčko vieme vypočítať pomocou políčka priamo nad ním a políčka o jedna nad ním a doľava. No a naviac môžeme použiť funkciu `MOD(a, 2)`, ktorá počíta zvyšok čísla a po delení 2. Ak totiž prítomnosť faraóna v miestnosti reprezentujeme číslom 1, tak keď sa pozrieme na súčet dvoch políčok nad bunkou, ktorú chceme vypočítať, chceme dostať súčet 1 a nechceme dostať súčet 0 a 2. Súčet 0 aj 2 majú však rovnaký zvyšok po delení 2 a to je 0. Naviac, tento zvyšok je presne číslo, ktoré chceme vložiť do našej bunky. Príklad takejto tabuľky je napríklad tu.

Vzorové riešenie tejto úlohy však nebolo kreslenie ani počítanie Excelom. Chcelo to len trochu sa zamyslieť a zároveň vás posunúť správnym smerom pred riešením podúlohy c.. Poďme si ešte raz preriešiť zadané dva príklady.

Poschodie 20, izba 10

Poschodia, ktoré sú tvaru 2^n vieme riešiť hneď, pretože obsahujú iba faraónov. Bohužiaľ 20 nie je mocninou čísla 2. Najbližšia menšia mocnina čísla 2 je 16. A z podúlohy a. vieme, že pod týmto poschodím sa tvoria dve nové pyramídy. Poschodie 20 teda vyzerá tak, že obsahuje dve kópie poschodia 4 oddelené prázdnymi miestnosťami. Prvá kópia poschodia 4 je úplne naľavo na políčkach 1 až 4 (štvrté poschodie má štyri políčka). Druhá kópia je úplne napravo na políčkach 17 až 20. A políčka medzi tým, teda políčka 5 až 16 sú prázdne.

Z toho vyplýva, že izba 10 na poschodí 20 neobsahuje faraóna.



[1] Alebo ľubovoľný iný tabuľkový editor.

poschodie 276, izba 266

Číslo 276 opäť nie je mocninou 2, najbližšia menšia mocnina je 256. Poschodie 256 preto obsahuje iba faraónov. Poschodie 276 je preto tvorené dvoma kópiami riadku 20 oddelenými prázdnymi izbami. Ľavá kópia je na políčkach 1 až 20, pravá na políčkach 257 až 276. Izba 266 teda neleží v prázdnych izbách medzi nimi, ale patrí pravej kópii.

Presnejšie, je to izba číslo 10 pravej kópie riadku číslo 20 na poschodí 276. To ale znamená, že zistiť, či je v izbe 266 na poschodí 276 faraón je rovnaké, ako zistiť, či je faraón v izba 10 na poschodí 20. A to sme predsa už zistili v prechádzajúcom zadaní. V izbe 266 na poschodí 276 sa teda faraón nenachádza.

Podúloha c.

Pre riešenie podúlohy c. si vyššie popísaný postup zovšeobecníme. Nech p označuje číslo poschodia, c_i číslo izby a m_2 najbližšiu mocninu čísla 2 menšiu ako p.

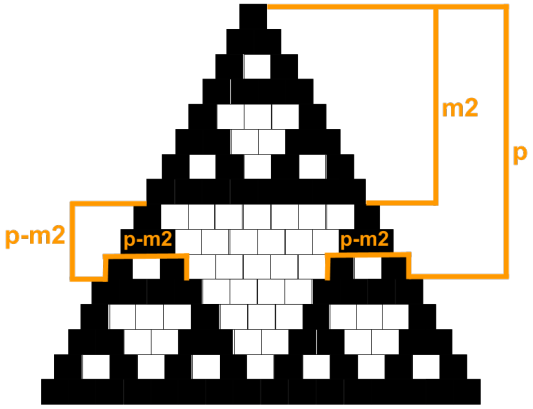
Začnime tým, že si popíšeme ako vypočítať hodnotu m_2. Začínajúc `m_2 = 1` budeme toto číslo násobiť až do momentu, keď $m_2 \geq p$. Týmto spôsobom narazíme na prvú mocninu 2, ktorá je väčšia alebo rovná číslu p. Ak m_2 následne vydelíme 2, ostane nám v tejto premennej hľadaná hodnota – najbližšia mocnina 2 menšia ako p.[2]

Ako prvé overme, či je číslo p mocnina 2. V prípade, že je, vieme, že bez ohľadu na to, na akú izbu sa pýtame, tak v nej bude faraón. Keďže m_2 je najbližšia mocnina menšia ako 2, tak p je mocnina 2 iba ak $2m_2 = p$.

V opačnom prípade vieme, že pod riadkom m_2 sa začali tvoriť dve menšie pyramídy a na riadku p sa nachádzajú ich v $p-m_2$-hé riadky. Tieto dve kópie sú dlhé $p-m_2$, jedna je úplne naľavo, druhá úplne napravo a medzi nimi sú prázdne izby. Presnejšie izby s číslami 1 až $p-m_2$ tvoria ľavú kópiu a izby s číslami $p-(p-m_2)+1 = m_2+1$ až p tvoria pravú kópiu.

Môžeme preto overiť, či platí $(p-m_2) < c_i < (m_2+1)$. V takom prípade sa totiž v hľadanej izbe faraón určite nenachádza.

V opačnom prípade sa naša izba nachádza v jednej z dvoch menších pyramíd. Ak je v ľavej pyramíde ($c_i \leq p-m_2$), tak vieme, že pýtať sa na poschodie p a izbu c_i je to isté ako sa pýtať na poschodie $p-m_2$ a izbu c_i. Naopak ak je v pravej pyramíde, tak otázka na poschodie p a izbu c_i je rovnaká ako otázka na poschodie $p-m_2$ a izbu c_i-m_2. V oboch prípadoch však dostaneme výrazne ľahší problém, pretože hodnota poschodia, ktoré hľadáme sa nám výrazne znížila.



A tento istý postup môžeme opakovať. Stačí si pre nové poschodie $p-m_2$ vypočítať novú hodnotu m_2 a dostať ešte menší problém. A tento postup opakovať, až kým sa nedostaneme na niektorý z najľahších problémov. Napríklad na ten, či je na vrchu pyramídy faraón.

Otázka však je, koľko opakovaní bude tento postup potrebovať. Uvedomme si však nasledovné. Keďže m_2 je najbližšia mocnina menšia ako p, tak platí $m_2 < p \leq 2m_2$. Nové poschodie, ktoré budeme musieť počítať je $p-m_2$, čo s použitím predchádzajúcej nerovnice znamená, že od p sme odčítali aspoň jeho polovicu. V každom kroku sa nám teda číslo poschodia zmenší na aspoň polovicu.

Počet krokov bude teda najviac počet delení číslom 2, ktoré vieme s číslom p robiť, kým nedostaneme číslo 1. Toto číslo sa volá logaritmus a zapisujeme ho ako $\log_2 p$. Táto hodnota navyše rastie veľmi pomaly. Ak by sme chceli napríklad vypočítať, či sa nachádza nejaký faraón v izbe 147 na poschodí 10^9 (miliarda), potrebovali by sme najviac 30 výpočtov. A porovnajte si to s tým, že by ste museli vypisovať celú pyramídu s miliardou riadkov.

Popísaný postup by sme mohli zapísať aj nasledovne. Za `//` sa nachádzajú komentáre pre lepšiu čitateľnosť.

[2] Všimnite si, že uvedený postup nefunguje pre $p = 1$, keďže od 1 neexistuje menšia mocnina 2. Treba si preto dať pozor na tento špeciálny prípad.

```
najdi_faraona (p , c):    // zadáme poschodie (p) a izbu (c)
    opakuj až kým nie je koniec:
        ak p = 1 -> koniec, v tejto izbe je faraón    // poschodie 1 je jednoduché
        // vypočítame hodnotu m2
        m2 = 1
        opakuj kým m2 < p:
            m2 = m2 * 2
        m2 = m2 / 2    // keď skončí cyklus, v m2 je mocnina 2 väčšia alebo rovná p

        ak 2*m2 = p -> koniec, v tejto izbe je faraón    // p je mocnina 2
        v opačnom prípade ->
            ak p-m2 < c < m2+1 -> koniec, v tejto izbe faraón nie je    // prázdny priestor
            inak ak c <= p-m2 ->    // sme v ľavej menšej pyramíde
                p = p-m2
                c = c
            inak ->    // sme v pravej menšej pyramíde
                p = p-m2
                c = c-m2
        pokračuj v cykle s novými hodnotami p a c
```

Veľmi podobne bude tiež vyzerať riešenie v Pythone. Od vás sme samozrejme program nechceli, môže však byť zaujímavé sa na to pozrieť.

Listing programu (Python)

```
def najdi_faraona(p, c):
    while True:
        if p == 1:
            return True
        m2 = 1
        while m2 < p:
            m2 = m2 * 2
        m2 = m2 / 2

        if 2*m2 == p:
            return True
        else:
            if p-m2 < c and c < m2+1:
                return False
            elif c <= p-m2:
                p = p-m2
                c = c
            else:
                p = p-m2
                c = c-m2

p = int(input())
c = int(input())
if najdi_faraona(p, c):
    print('V izbe {} na poschodí {} sa nachadza faraon'.format(p,c))
else:
    print('V izbe {} na poschodí {} sa nenachadza faraon'.format(c,p))
```

2. Pavúčí sklad

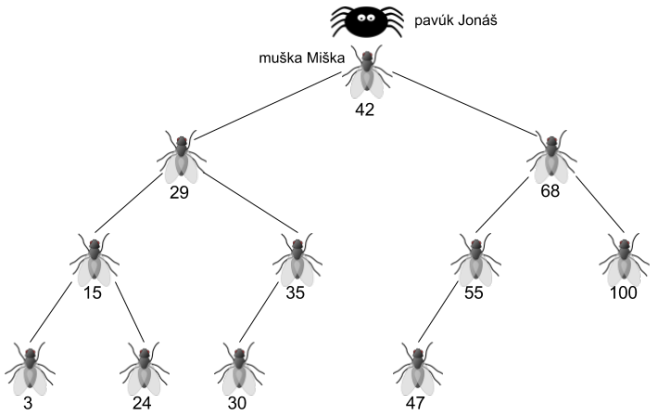
vzorák napísal(a) Žaba
(max. 15 b za riešenie)

Táto úloha v skutočnosti popisovala jeden veľmi známy a skúmaný problém informatiky – binárne vyhľadávacie stromy. Cieľom týchto stromov je skutočne ukladanie a vyhľadávanie čísel (alebo iných hodnôt) a sú často používané v rôznych aplikáciách. Ich rozšírenosť ilustruje aj to, že v množstve programovacích jazykov sú ich natívnou súčasťou (napr. `set` v `C++` alebo `dict` v `Pythone`).

Jednotlivé podúlohy vás navádzali na riešenie skutočných problémov, ktoré sa pri binárnych stromoch riešia – hľadanie, vkladanie a vymazávanie hodnôt. A riešenia, ktoré si prezentuje v tomto vzorovom riešení pomerne presne zodpovedajú riešeniam, ktoré sa aj v skutočnosti používajú. Ak ste teda riešili túto úlohu, môžete sa cítiť ako skutočný informatici :)

Začnime tým, že si zopakujeme, čo vieme o Jonášovej pavučine. V Jonášovej pavučine sa nachádza niekoľko mušiek, každá z nich má inú váhu. Naviac, z každej mušky vychádza práve jedno (okrem mušky úplne navrchu) vlákno dohora a najviac dve vlákna dodola – jedno doprava a jedno doľava. A pre každú mušku platí, že všetky mušky, ktoré sú zavesené pod ňou naľavo sú ľahšie ako táto muška a všetky mušky, ktoré sú zavesené pod ňou

napravo sú ťažšie ako táto muška. A keďže muška na vrchu siete je špeciálna (lebo z nej nevychádza vlákno dohora), voláme ju Miška.



Podúloha a.

Jonáš stojí na vrchu pavučiny a chce nájsť najľahšiu mušku, ktorú má v pavučine chytenú. Ako to spraví?

Jonáš začína pri muške Miške. Čo vieme o tejto muške? To čo o všetkých ostatných – naľavo od nej sú mušky ľahšie a napravo od nej mušky ťažšie. Ak sa chce teda Jonáš dostať k najľahšej muške, určite musí ísť doľava, vpravo sú totiž všetky mušky ťažšie.

Posuňme teda Jonáša po ľavom vlákne k najbližšej muške. Uvedomme si však, že pre túto mušku platí tá istá vlastnosť a Jonáš je preto v rovnakej situácii – ak pôjde doľava, pôjde k ľahším muškám, ak doprava, tak k muškám ťažším. Opäť sa teda musí posunúť po ľavom vlákne.

Táto situácia sa bude následne opakovať. Od každej mušky bude chcieť ísť Jonáš doľava, lebo vľavo sa nachádzajú ľahšie mušky. Zastaví sa až vtedy, keď doľava už nebude môcť ísť – z mušky pri ktorej stojí nevedie ľavé vlákno. Možno z nej aj vedie vlákno doprava, vpravo sú však iba ťažšie mušky a k tým sa Jonáš nechce dostať. Preto táto muška musí byť najľahšia v celej pavučine.

Všimnite si, že táto situácia mohla nastať kedykoľvek, kľudne aj pri muške Miške. Je ale jasné, že ak z Mišky nevedie vlákno doľava, Miška je najľahšie muška v pavučine.

Jonášov postup: začínajúc pri muške Miške sa vždy posuň dodola po ľavom vlákne k najbližšej muške. Ak v nejakom momente stojíš pri muške, z ktorej doľava už vlákno nevedie, táto muška je najľahšia v celej sieti.

Podúloha b.

V tejto podúlohe chce Jonáš nájsť najľahšiu mušku, ktorá je ťažšia ako muška Miška.

Jonáš opäť začína pri muške Miške. Ktorým smerom sa má vydať? Ak pôjde doľava, dostane sa iba k ľahším muškám, čo nie je dobre. Preto musí ísť vláknom doprava.

Po prvom presune je však Jonáš medzi muškami, ktoré sú všetky ťažšie ako Miška. A k ľahším sa už späť nedostane. Keďže išiel doprava od Mišky, všetky mušky, ktoré stretne budú ťažšie ako ona. Ktorú mušku teda teraz hľadá? No predsa tú najľahšiu. A to, ako nájsť najľahšiu mušku, sme si ukázali už v podúlohe a..

Jediné, čo musí Jonáš teraz robiť je posúvať sa ľavým vláknom dodola, až kým nenarazí na mušku, z ktorej už ľavé vlákno nevedie. Toto bude najľahšia muška ťažšia ako Miška.

Jonášov postup: v prvom kroku sa pohni doprava od Mišky. Následne choď už celý čas iba doľava, až kým nenarazíš na mušku, z ktorej vlákno doľava nevedie. To bude hľadaná muška.

Podúloha c.

Jonáš hľadá v pavučine mušku s váhou presne x. Poprípade má zistiť, že taká muška sa v pavučine nenachádza.

Ako prvé by Jonáš mohol overiť, či vlastne nehľadá mušku Mišku. Porovná teda váhu Mišky s číslom x. Ak sa rovnajú, tak je rád, lebo našiel mušku, ktorú hľadal. Čo však, ak sa nerovnajú?

V takom prípade sa bude musieť pohnúť buď doľava alebo doprava. No a o muškách naľavo vieme, že sú ľahšie ako Miška a o muškách napravo, že sú ťažšie ako Miška. Pozrieme sa teda, či hľadané x je väčšie alebo menšie ako Miškina váha. Ak je väčšie, pôjdeme doprava, kde sa nachádzajú väčšie čísla, ak je menšie, tak naopak doľava.

Po prvom posunutí, ktoré je navyše jednoznačné, sa ocitáme pri muške s váhou y. Porovnajme teraz toto číslo s číslom x. Ak $y = x$, tak Jonáš našiel mušku, ktorú hľadal a už sa nemusí viac hýbať. Ak $x < y$, tak hľadaná muška je ľahšia ako muška, pri ktorej aktuálne stojí a teda sa musí pohnúť doľava. A ak $x > y$, tak hľadaná muška je ťažšia a musí ísť doprava.

Po každom kroku je teda Jonášov postup pevne daný tým, koľko váži muška, pri ktorej stojí. Porovnanie jej váhy a čísla x mu totiž povie, ktorým smerom sa má vydať. A keďže nikdy nemá na výber, tak sa nemôže pomýliť. Ak sa taká muška v pavučine nachádza, tak ju určite nájde. Čo ale v prípade, že muška s váhou x v pavučine nie je?

V takom prípade bude Jonáš cestovať pavučinou, až kým sa niečo nepokazí. Presnejšie, až do momentu, keď mu jeho postup neoznámi, že sa má pohnúť po vlákne, ktoré neexistuje. Predstavte si, že stojí pri muške s váhou y a $x < y$. Z mušky y však vlákno doľava nevedie. Jonáš vie, že v takomto prípade muška s váhou x v sieti nie je. Ak by tam bola, musela by byť naľavo od y, naľavo od y však nič nie je.

Jonášov postup: nech stojíš pri muške s váhou y. Ak sa $y = x$, tak si našiel hľadanú mušku. Ak $x < y$, tak sa musíš pohnúť po vlákne doľava, ak $x > y$, tak sa musíš pohnúť po vlákne doprava. Ak vlákno, po ktorom sa máš pohnúť neexistuje, tak vieš, že muška s váhou x sa v sieti nenachádza.

Podúloha d.

V tejto podúlohe chytil Jonáš mušku s váhou x, stojí s ňou na vrchu pavučiny a chce ju niekam zavesiť. Ako to má spraviť?

Začnime tým, že zistíme, či má byť nová muška naľavo alebo napravo od Mišky. Ako to zistíme? Jednoducho porovnáme číslo x s váhou Mišky. Ak je x ľahšie, tak novú mušku musí zavesiť doľava, v opačnom prípade doprava.

Keď sa však daným smerom posunie, dostáva sa do rovnakej situácie. Podľa váhy mušky, pri ktorej stojí sa vie jednoznačne rozhodnúť, či má byť nová muška zavesená vľavo alebo vpravo. Uvedomme si naviac, že tento postup je úplne rovnaký ako postup v podúlohe c.. Tiež v podstate hľadáme umiestnenie mušky s váhou x v pavučine.

Muška s váhou x sa však v pavučine ešte nenachádza, však ju tam len chceme pridať. Ako sme si však vraveli, pri postupe z podúlohy c. bude Jonáš postupovať pavučinou, až kým nenarazí na situáciu, v ktorej zistí, že sa má pohnúť po vlákne, ktoré neexistuje. V našom prípade je to však presne miesto, kam má zavesiť jeho novú mušku.

Predstavme si, že stojí pri muške s váhou y a platí $y < x$, žiadne vlákno doprava však nevedie. Vidíme, že muška x by mala byť napravo od mušky y. No a keďže napravo nič nie je, môžeme vytvoriť nové vlákno, ktoré pôjde z y doprava a na jeho konci bude muška s váhou x.

Overme si už len, že celá sieť je po tomto pridaní v poriadku. Stále platí, že z každej mušky vedú dodola najviac dve vlákna. Pridali sme totiž iba jedno vlákno a aj to na miesto, kde chýbalo. Ostáva skontrolovať, že naľavo od každej mušky sú iba ľahšie mušky a napravo iba ťažšie. Pod muškou x zatiaľ žiadna ďalšia muška nie je zavesená, takže pre ňu to platí. A pre všetky zvyšné, pod ktorými leží muška x to platí tiež. V každom kroku sme sa totiž hýbali tak, ako keby tam muška x už bola. Preto muška x leží vždy na správnej strane.

Jonášov postup: nech stojíš pri muške s váhou y. Ak $x < y$, tak sa musíš pohnúť po vlákne doľava, ak $x > y$, tak sa musíš pohnúť po vlákne doprava. Ak vlákno, po ktorom sa máš pohnúť neexistuje, tak ho vytvor a na jeho koniec zaves mušku x.

Podúloha e.

Pavúk Jonáš zjedol jednu z mušiek a teraz stojí na jej mieste. Potrebuje previazať niektoré vlákna a presunúť nejaké mušky tak, aby jeho pavučina opäť spĺňala všetky požadované podmienky.

Ako už naznačovalo zadanie, zložitosť tejto operácie bude závisieť od toho, koľko vlákien vedie dodola z mušky, ktorú Jonáš zjedol. Ak pod muškou, ktorú zjedol už nie je žiadna ďalšia muška, situácia je jednoduchá. Odstránením tejto mušky sa pavučina vôbec nepokazí.

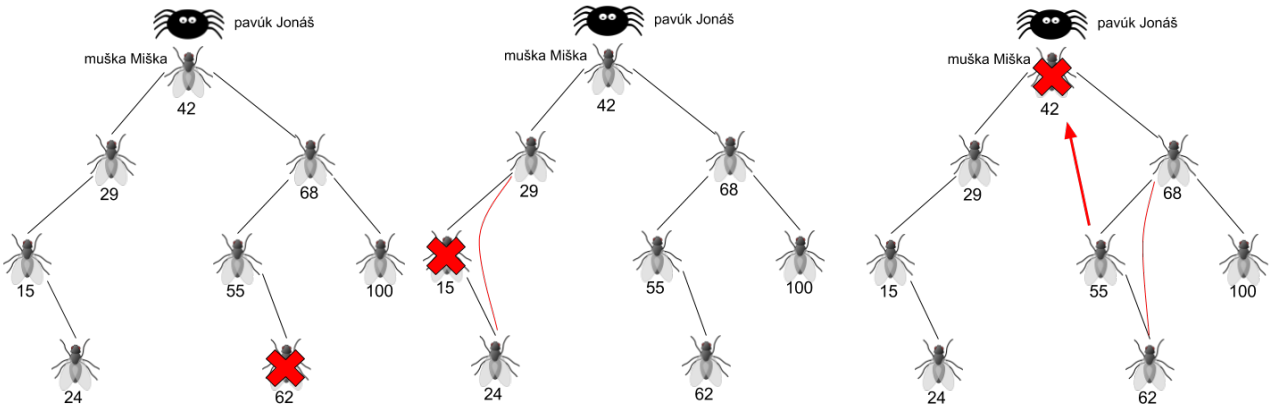
Poďme sa teda pozrieť na to, čo sa stane, keď z tejto mušky viedlo jedno vlákno, pričom je jedno, či viedlo doprava alebo doľava. Nech zjedená muška mala váhu x a muška nad ňou váhu y. Ďalej predpokladajme, že $x < y$, teda z mušky y viedlo doľava vlákno k muške x. Uvedomme si, že všetky mušky vľavo od y boli menšie ako y. Preto je nám jedno, či z x viedlo vlákno doľava alebo doprava. Dôležité je, že všetky mušky pod x sú aj tak menšie ako y. Keď sme odstránili mušku x, z y vedie doľava jedno prázdne vlákno, kam sa majú pripojiť mušky menšie ako y. A do x viedlo zdola jedno vlákno, na ktorom sú zavesené mušky ľahšie ako y. Tieto dve vlákna teda môžeme spojiť v jedno, čím dostaneme súvislú pavučinu, v ktorej stále platia všetky vlastnosti.

Ostáva už len najzložitejšia situácia, v ktorej z odstránenej mušky (nech má váhu x) vedie vlákno aj doľava aj doprava. Teraz už len tak ľahko nevieme vlákna previazať. Z vrchu totiž vedie iba jedno vlákno, ale dodola vedú dve. Bolo by teda fajn na miesto mušky x dať nejakú inú mušku zo siete. Ktorú?

Zdá sa logické, aby to bola niektorá z mušiek, ktoré ležia pod x. Mušky pod muškou x sú rozdelené na dve časti – ľahšie mušky, ktoré sú naľavo a ťažšie mušky, ktoré sú napravo. Predstavme si, že vyberieme niektorú z mušiek napravo a dáme ju na miesto mušky x. Keďže sme ju vybrali z pravej časti, vieme, že je ťažšia ako x. To znamená, že mušky naľavo sú ľahšie ako ona. Čo je super, lebo ľahšie mušky majú byť naľavo. Sú však všetky mušky napravo ťažšie ako táto muška?

Keďže sme vyberali ľubovoľnú mušku z pravej časti, tak nie nutne. Akú mušku musíme z pravej časti vybrať, aby všetky zvyšné mušky napravo boli ťažšie ako ona? No predsa tú najľahšiu. Takže najlepším kandidátom na nahradenie mušky x je najľahšia muška ťažšia ako x. Hľadať takúto mušku však vieme, ukázali sme si to predsa v podúlohe b..

Riešenie je teraz už jednoduché. Jonáš sa od zjedenej mušky x posunie doprava a potom ide celý čas doľava, až kým nenájde najľahšiu mušku ťažšiu ako x. Túto mušku musí vybrať z pavučiny a dať ju na miesto mušky x. Uvedomme si však, že z tejto novej mušky vedie dodola najviac jedno vlákno. Nemôže z nej totiž viesť vlákno doľava, lebo by nebola najľahšia. Takže ju vieme ľahko z pavučiny vybrať – to ako odstrániť mušku, z ktorej vedú menej ako dve vlákna sme si popísali vyššie.



Na obrázku sú znázornené všetky tri možné situácie. Červený krížik ukazuje, ktorá muška bola zjedená, červené čiary, ukazujú, ktoré nové vlákna budú pridané a červená šipka znamená presunutie mušky na novú pozíciu.

Jonášov postup: ak z mušky, ktorú zješ nevedie dodola žiadne vlákno, nemusíš nič robiť. Ak z nej vedie práve jedno vlákno, toto vlákno napoj na mušku priamo nad zjedenou muškou. V prípade, že zješ mušku, z ktorej vedú dodola obe vlákna, začni tým že nájdeš najľahšiu mušku, ťažšiu ako zjedená muška. To spravíš tak, že sa posunieš najskôr doprava a potom pôjdeš doľava až kým sa už ďalej doľava nebudeš vedieť pohnúť. Zober mušku, pri ktorej stojíš a uprav pavučinu, ako keby si túto mušku zjedol. Vedie z nej najviac jedno vlákno, takže situácia je ľahká. Túto vybranú mušku nakoniec polož na miesto zjedenej mušky.

3. Pohodlné Kreslo

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

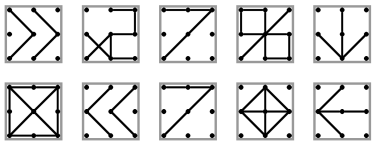
Vzorové programy nájdete aj na stránke s editorom ksp.sk/~prask/specialne/4/1/3 kde si môžete odsimulovať ich beh. Pri čítaní vzorového riešenia odporúčame si to aj skúšať. Hoci vám slovne načrtneme základné myšlienky, predsa len je to prehľadnejšie, keď sa to aj hýbe.

Podúloha a.

Na vstupe je zadané číslo a. Vypíšte číslo 7 ak je a deliteľné číslom 7 (je to násobok sedmičky), inak vypíšte číslo a nezmenené.

Na vyriešenie prvej podúlohy stačí pochopiť ako fungujú jednotlivé inštrukcie. Následne existuje niekoľko veľmi podobných riešení. Každé však začína tým, že načítame vstup. Načítanú hodnotu si ale potrebujeme niekam uložiť. Na to môžeme použiť buď niektorý z registrov alebo zásobník. My si ukážeme riešenie pomocou zásobníka. Číslo zo vstupu zduplikujeme (aby tam jedna kópia ostala aj po kontrole deliteľnosti), na zásobník pridáme číslo 7 a spravíme operáciu modulo. Následne pridáme podmienený skok, vďaka ktorému dáme číslo 7

na vrch zásobníka, iba ak vyšiel zvyšok nula. Ak zvyšok nie je 0, táto inštrukcia sa preskočí a na vrchu zásobníka bude pôvodné číslo zo vstupu. Ostáva vypísať číslo, ktoré je na vrchu zásobníka a ukončiť program.



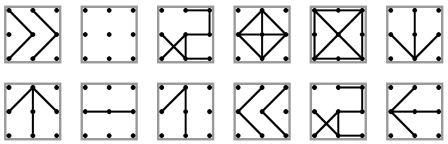
Podúloha b.

Na vstupe je zadané číslo n. Vypíšte čísla od n po 1 v klesajúcom poradí.

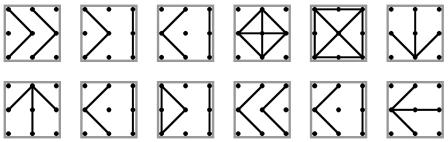
Pri riešení tejto podúlohy musíme časť nášho programu opakovať. To vieme dosiahnuť pomocou šipiek, vo vzorovom riešení sa však používa aj nasledovný trik – keďže na vstupe je len jedno číslo, tak keď sa náš program ocitne druhýkrát na príkaze na načítanie vstupu, tak nič nenačíta (lebo nemá čo) a iba sa otočí o 90° doprava. Túto inšrukciu budeme preto "recyklovať" – prvý krát ju použijeme na načítanie vstupu a následne ju budeme používať ako šipku doprava.

Vyriešiť úlohu je už potom ľahké. Na zásobníku (alebo v registri) si pamätáme aktuálne číslo, ktoré máme vypočítať. Podmieneným skokom overíme, či toto číslo nie je 0. Ak áno, program ukončíme. Inak číslo zduplikujeme a kópiu navrchu vypíšeme. Potom pridáme 1 a znamienko mínus zmenší číslo na zásobníku (v registri). V tomto momente môžeme celý proces začať odznovu.

So zásobníkom:



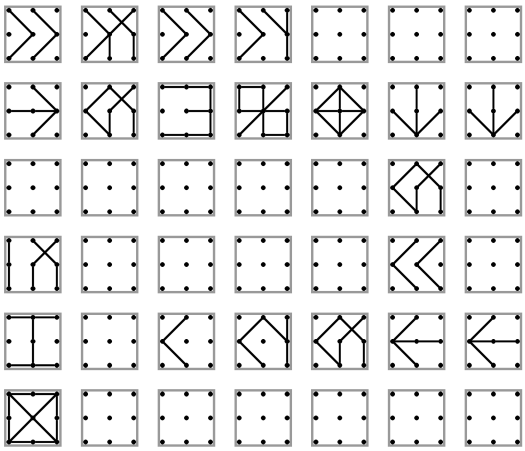
S registrom:



Podúloha c.

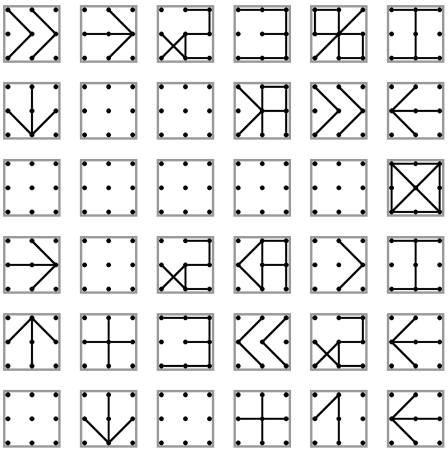
Na vstupe sú zadané čísla x a y. Vypíšte v rastúcom poradí všetky čísla od x do y (vrátane), ktoré sú násobkami čísla 3.

Táto úloha sa dá vyriešiť jednoduchou úpravou predošlého riešenia. Pri každom opakovaní iba navyše overíme, či je dané číslo deliteľné troma.



Ukážeme si však ešte jedno riešenie. Je síce o niečo zložitejšie, ale o to krajšie. Namiesto toho, aby sme prechádzali všetkými číslami, budeme prechádzať **iba násobkami čísla 3**.

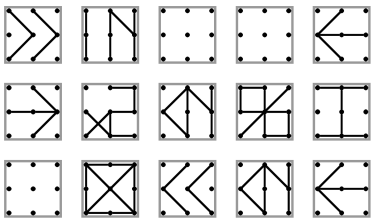
Samozrejme, je problém, ak číslo x nie je na začiatku násobok 3. Preto ho ešte pred začiatkom zväčšíme na najbližší väčší násobok 3 pomocou jednoduchého cyklu, ktorý sa bude opakovať, kým nebude zvyšok po delení čísla x nula. Po zväčšení čísla x načítame druhé číslo zo vstupu do registra `A` a začneme samotný cyklus vypisovania. Ten začneme tým, že overíme, že hranica v registry `A` ešte nebola prekročená. Ak nie, vypíšeme x a následne ho zväčšíme o 3, čím sa posunieme rovno na ďalší násobok 3.



Podúloha d.

Na vstupe je zadané číslo n. Vypíšte najmenšie číslo a, $1 < a$ ktoré delí n (zvyšok po delení n číslom a je 0).

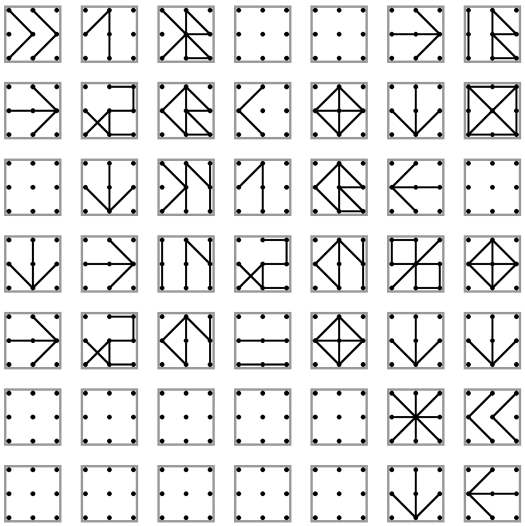
Opäť jednoduchšia úloha, riešením je jeden cyklus, v ktorom hľadáme najmenšieho deliteľa. Stačí postupne zväčšovať jedno číslo (pozor, aby sme začínali na dvojke a nie na jednotke) a v okamihu, keď toto číslo delí n vypísať ho a ukončiť program. No a overenie deliteľnosti nejakým číslom sme predsa už robili v podúlohe a. aj c..



Podúloha e.

Na vstupe je zadané číslo n. Vypíšte všetky prvočísla z rozsahu od 2 po n v rastúcom poradí.

Na vyriešenie tejto úlohy treba spojiť riešenia podúloh c. a d.. Budeme mať vonkajší cyklus, ktorý prechádza postupne všetkými číslami a vnútorný cyklus, ktorý pre dané číslo nájde najmenšieho deliteľa. Potom nám stačí skontrolovať, či sa tento deliteľ rovná nášmu číslu. Ak áno, dané číslo je prvočíslo a môžeme ho vypísať. V opačnom prípade pokračujeme na ďalšie číslo.



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

4. Pixelová výzva

Myšlienka

Máme obrázok o veľkosti $r \times s$, v ktorom chceme nájsť dĺžky súvislých úsekov čiernych políčok v riadkoch aj stĺpcoch. Ako prvé si musíme tento obrázok uložiť do pamäte počítača. To však nie je problém, práve na to sú určené dvojrozmerné polia. Dvojrozmerné pole si môžete predstaviť ako tabuľku, v ktorej je každé políčko označené číslom riadka a stĺpca. Jediné na čo si dajte pozor je, že v informatike číslujeme od 0.

Začnime tým, že si vysvetlíme, ako vypočítať výsledok pre jeden riadok. Riadok budeme prechádzať postupne zľava doprava. Naviac budeme mať jednu premennú `ans`, v ktorej si budeme pamätať, aký dlhý úsek čiernych políčok (jednotiek) máme aktuálne prečítaný. Predstavme si, že ideme spracovávať ďalšie políčko v riadku. Ak je na tomto políčku 1, znamená to, že aktuálny úsek jednotiek má byť o jedna dlhší. Preto zväčšíme hodnotu v našej premennej: `ans = ans + 1`.

Naopak, ak je na tomto políčku 0, úsek jednotiek na ňom končí. Pozrieme sa teda, aká hodnota je v premennej `ans`. Ak je táto hodnota nenulová, skutočne tu končí nejaký úsek jednotiek a jeho dĺžku (obsah premennej `ans`) by sme mali vypísať. Následne hodnotu `ans` vynulujeme, lebo aktuálne nemáme prečítané žiadne za sebou idúce jednotky, a pokračujeme v spracovávaní riadka.

Pozor si musíme dať v dvoch prípadoch. Ak riadok končí 1, tak posledný úsek si nikdy nepoznačíme, lebo niečo také by sme spravili až keby sme narazili na ďalšiu 0 v danom riadku, žiadna v ňom však už nie je. Po spracovaní celého riadku sa preto treba ešte raz pozrieť na to, či v premennej `ans` nie je nenulová hodnota – nejaký zabudnutý úsek. Druhý špeciálny prípad je, ak v riadku nie je žiadna 1, vtedy si musíme dať pozor, aby sme vypísali požadovanú 0.

No a nie je ťažké si uvedomiť, že spracovanie stĺpcov je veľmi podobné. Jediný rozdiel pri našom algoritme bude v tom, že načítaný obrázok neprechádzame po riadkoch, ale po stĺpcoch. Všetko ostatné ostane rovnaké.

Časová a pamäťová zložitosť

Aká bude časová zložitosť tohto riešenia? Celý obrázok musíme prejsť dvakrát – raz po riadkoch a raz po stĺpcoch. To nás stojí $O(r \cdot s)$ operácií, pretože na každom z $r \times s$ políčok strávime iba konštantne veľa času – upravíme premennú `ans`. Nemôžeme ešte zabudnúť do časovej zložitosti započítať načítavanie obrázka a výpis úsekov. Stačí si ale uvedomiť, že celkový počet úsekov je zhora obmedzený veľkosťou obrázka. Preto výsledná časová zložitosť ostane $O(r \cdot s)$.

Pamäťová zložitosť je tiež $O(r \cdot s)$, pretože si celý obrázok musíme uložiť niekam do pamäte. Ak by sme ale nemuseli počítať úseky v stĺpcoch, ale iba v riadkoch, vedeli by sme dosiahnuť pamäťovú zložitosť $O(1)$. Na riešenie by nám teda stačilo iba niekoľko premenných. Viete prísť na to, ako by sme to robili?

Listing programu (C++)

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

using namespace std;

inline void vypis(const vector<int> &useky) {
    if (useky.empty()) { // v riadku/stlpci sa nenachadzaju ziadne useky
```

```
        cout << "0\n";
        return;
    }

    for(int i = 0; i < useky.size(); ++i) {
        if (i > 0)
            cout << " ";

        cout << useky[i];
    }

    cout << "\n";
}

int main() {
    int r, s;
    cin >> r >> s;

    vector<vector<int> > obrazok(r, vector<int>(s));

    // nacitanie obrazku
    for(int i = 0; i < r; ++i) {
        for(int j = 0; j < s; ++j) {
            cin >> obrazok[i][j];
        }
    }

    vector<int> useky;
    int ans;

    // spracovanie riadkov
    for(int i = 0; i < r; ++i) {
        ans = 0;

        for(int j = 0; j < s; ++j) {
            if (obrazok[i][j] == 1) { // usek mozeme predlzit
                ans++;
            } else { // narazili sme na cierne policko, takze tu konci usek dlzky ans
                // musime osetrit pripad, kedy ma usek dlzku 0 (napr. ak nasleduju
                // dve cierne policka za sebou
                if (ans > 0) {
                    useky.push_back(ans);
                    ans = 0;
                }
            }
        }

        // riadok mohol skoncit 1 (bielym polickom), takze posledny usek
        // musime pridat zvlast
        if (ans > 0)
            useky.push_back(ans);

        vypis(useky);
        useky.clear();  // vymazanie pola useky
    }

    cout << "\n";

    // spracovanie stlpcov je velmi podobne spracovaniu riadkov,
    // rozdiel je v tom, ze obrazok prechadzame po stlpcoch, takze
    // sa nam zmenia medze for-cyklov a
    for(int j = 0; j < s; ++j) { // pre kazdy stlpec
        ans = 0;

        for(int i = 0; i < r; ++i) { // pre kazdy riadok v stlpci
            if(obrazok[i][j] == 1) {
                ans++;
            } else {
                if(ans > 0) {
                    useky.push_back(ans);
                    ans = 0;
                }
            }
        }

        if (ans > 0)
            useky.push_back(ans);

        vypis(useky);
        useky.clear();
    }

    return 0;
}
```

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

5. Problematický tréning

V tejto úlohe máme v kope údajov nájsť dve miesta, ktoré majú rovnakú výšku a sú od seba čo najvzdialnejšie. Tie by totiž mohli ukazovať na okruh, ktorý Roman beháva. Problémom však je, že nemáme k dispozícii postupnosť výšok, v ktorých sa Roman nachádzal, ale len zmeny nadmorskej výšky, ktoré pri behu robil.

Mohli by sme teda začať tým, že z tejto postupnosti zmien vypočítame, v akých nadmorských výškach sa

Roman kedy nachádzal. Samozrejme nevieme, na akej nadmorskej výške začínal. Budeme však predpokladať, že to bolo vo výške 0. Čo sa totiž zmení, ak by skutočná začiatočná výška bola napríklad 10? Len to, že všetky ďalšie výšky budú tiež o 10 vyššie. Ale naším cieľom je hľadať dvojicu rovnakých čísel. A tá bude rovnaká bez ohľadu na to, či bude zväčšená alebo zmenšená o 10.

Budeme si teda udržiavať jednu premennú `vyska`, v ktorej si budeme zaznačovať, v ktorej nadmorskej výške sa Roman práve nachádza. Na začiatku sa `vyska = 0`. Následne prechádzame postupnosťou zmien a každú zmenu jednoducho pričítame k aktuálnej výške. Ak Roman stúpol o 10, tak pričítame 10, ak o 10 klesol, tak pričítame -10.

Tento postup, v ktorom postupne nasčítavame vstupné hodnoty je pomerne štandardný a má názov prefixové súčty. Viac si o ňom môžete prečítať v našej kuchárke: ksp.sk/kucharka/prefixove_sumy. Následne vieme z poľa rozdielov spraviť pole výšok takýmto jednoduchým programom:

Listing programu (C++)

```
vector<long long> prefix(n+1);
prefix[0] = 0;
vyska = 0;
for(long long i=0; i<n; ++i) {
    vyska += rozdiely[i];
    prefix[i+1] = vyska;
}
```

Listing programu (Python)

```
vyska=0
prefix = [0]
for i in range(n):
    vyska += rozdiely[i]
    prefix.append(vyska)
```

V takto vytvorenom poli nadmorských výšok teraz chceme hľadať dve pozície, ktoré sú od seba čo najvzdialenejšie a majú rovnakú nadmorskú výšku. Najjednoduchší spôsob ako také niečo spraviť, je vyskúšať všetky možné dvojice.

A hoci je takéto riešenie správne, a na testovači zaň aj dostanete nejaké body, nie je úplne optimálne. Uvedomme si, že ak máme n výšok, tak všetkých dvojíc je n^2. Náš algoritmus bude mať preto zložitosť $O(n^2)$, čo je pre $n = 100\,000$ príliš veľa.

Listing programu (C++)

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

int main()
{
    //musime si pamatat cisla v long longoch lebo mozu byt az do 10^11
    long long   n,   //pocet usekov
                a, b,   //hranice aktualneho intervalu
                max_a, max_b, max_len = -1; //v troch premennych max_len, max_a a max_b si budeme
                                            //pamatat dlzku najdlhsieho doteraz videneho useku
    cin >> n; //nacitame pocet usekov

    vector<long long>rozdiely(n);
    for(long long i=0;i<n;++i)
    {
        cin >> rozdiely[i]; //ulozime vyskove rozdiely do pola (vector-u)
    }

    //vytvorime si prefixove pole prefix
    vector<long long>prefix(n+1);
    prefix[0] = 0;
    for(long long i=1;i<=n;++i)
    {
        prefix[i] = prefix[i-1] + rozdiely[i-1];
    }

    for(b=0;b<=n;++b) //prejdeme cez vsetky konce
    {
        for(a=0;a<b;++a)//a pre kazdy pre vsetky zaciatky; a<b
        {
            if((prefix[b]==prefix[a]) && (b-a+1 > max_len))
            //ak je vyhovujuci a dlhsi ako doteraz najdlhsi najdeny, tak si ho zapiseme
            {
                    max_a = a;
                    max_b = b-1;
                    max_len = max_b-max_a+1;
            }
        }
    }

    cout<<max_len<<endl; //vypiseme najdlhsi
    if(max_len != -1) //ak vobec existuje
```

```
        cout << max_a << " " << max_b << endl;
    return 0;
}
```

Listing programu (Python)

```
n = int(input()) #nacitame pocet usekov

rozdiely = [int(x) for x in input().split(' ')] #ulozime vyskove rozdiely do pola (list-u)

#vytvorime si prefixove pole prefix
vyska=0
prefix = [0]
for i in range(n):
    vyska+=rozdiely[i]
    prefix.append(vyska)

# v troch premennych max_len, max_a a max_b si budeme
# pamatat dlzku najdlhsieho doteraz videneho useku
max_len=-1
max_a=0
max_b=0

for b in range(n+1): #prejdeme cez vsetky konce
    for a in range(b): #a pre kazdy pre vsetky zaciatky; a<b
        #a, b su hranice aktualneho intervalu
        if prefix[b] == prefix[a] and b-a+1 > max_len:
            #ak je vyhovujuci a dlhsi ako doteraz najdlhsi najdeny, tak si ho zapiseme
            max_a = a;
            max_b = b-1;
            max_len = max_b-max_a+1;

print(max_len) #vypiseme najdlhsi
if max_len != -1: #ak vobec existuje
    print(max_a, max_b)
```

Vzorové riešenie

V predchádzajúcom riešení sme skúšali všetky možné dvojice čísel. Je to však nutné? Nerobili sme niečo zbytočne? Robili, napríklad sme porovnávali aj pozície, ktoré obsahovali rôzne čísla a teda okruh vôbec nemohli vytvoriť. Naše riešenie by sme preto vedeli zlepšiť, ak by sme výšku x porovnávali iba s výškami x. Môže sa nám však stať, že Roman bežal celý čas po rovine a zmeny nadmorskej výšky boli celý čas 0. V takom prípade by naše pole nadmorských výšok obsahovalo samé 0 a my by sme opäť skúšali všetky dvojice.

Ak by sme sa pozreli iba na pozície, na ktorých sa nachádza výška x, je nutné skúšať všetky dvojice týchto pozícií? Uvedomme si, že hľadáme dvojicu, ktorá je najvzdialenejšia. A ak hľadáme dva najvzdialenejšie výskyty čísla x, tak to znamená, že chceme zobrať prvý a posledný výskyt čísla x v poli. Ľubovoľná iná dvojica bude totiž k sebe bližšia.

Naše riešenie by preto mohlo vyzerať nasledovne. Po vypočítaní nadmorských výšok, v ktorých sa Roman nachádzal ich budeme prechádzať zľava doprava. Nech sa nachádzame vo výške x. Keďže nevieme, či toto x nie je posledné x v našom poli, musíme vyskúšať jeho vzdialenosť od prvého výskytu čísla x v našom poli. Ak je táto vzdialenosť najväčšia, akú sme zatiaľ videli, tak si ju zapamätáme.

Takéto riešenie znie sľubne, pretože vyskúšame iba n rôznych dvojíc – pre každú výšku sa pozrieme iba na najľavejšiu takú výšku v našom poli. Ešte sme však nevyriešili to, ako (čo najrýchlejšie) zistiť, na ktorej pozícii sa nachádza najľavšie číslo x v našom poli.

Najjednoduchšie riešenie by bolo zobrať pole `prve_vyskyty[]`, do ktorého by sme si tieto pozície značili. Na začiatku by boli v poli iba čísla -1. Vždy keď by sme spracovávali číslo x, najskôr by sme sa pozreli na pozíciu x v tomto poli. Ak by sa `prve_vyskyty[x]` rovnalo -1, tak by to znamenalo, že aktuálna pozícia je prvou pozíciou, na ktorej sa nachádza číslo x. Číslo tejto pozície by sme si preto zapísali do `prve_vyskyty[x]`. Naopak, ak by na tejto pozícii nebolo číslo -1, tak toto číslo by bola pozícia prvého výskytu čísla x a preto by sme rovno zistili ich vzájomnú vzdialenosť.

Takéto riešenie je síce veľmi pekné a jednoduché, má však malý háčik. Romanových záznamov je 100 000 a najväčšia zmena nadmorskej výšky, ktorú spravil môže byť až 1 000 000. To znamená, že keď počítame ako vysoko mohol vybehnúť, môžeme sa dostať až k číslam veľkým $10^5 \cdot 10^6 = 10^{11}$. Naše pole by teda muselo mať 10^{11} pozícií, na každej z nich by si pamätalo jedno 4 bajtové číslo, preto veľkosť takéhoto poľa by bola rádovo 400 GB. Takú veľkú operačnú pamäť (RAM) však na testovači nemáme (a asi ani vy doma).

Problémom takéhoto ukladania do poľa je, že v tomto poli je zbytočne veľa priestoru. Napriek tomu, že doň uložíme najviac 100 000 čísel, vytvoríme si 10^{11} políčok. Našťastie boli vymyslené dátové štruktúry, ktoré sa nazývajú asociatívne polia. Fungujú podobne ako klasické polia, akurát vytvárajú iba políčka, ktoré potrebujeme. Preto ak použijeme `asociativne_pole[1000]`, tak sa nevytvorí 1 000 políčok, ale iba jedno. Naviac v asociatívnych poliach vieme indexovať nielen celými číslami, ale takmer všetkým, čo sa dá navzájom porovnávať, napríkald stringom (text) alebo floatom (desatinné číslo).

V C++ sa asociatívne pole volá `map`. Základy jeho používania si môžete pozrieť v nasledujucom zdrojovom kóde:

Listing programu (C++)

```
#include <iostream>
#include <map>
#include <string>
using namespace std;
int main ()
{
  map<string,float> mapa; // deklaracia pod stringami(text) budeme mat ulozene rozne desatinne cislo

  mapa["cislo"]=11.5; //ulozenie cisla pod text
  mapa["dolezite cislo"]=42.47;
  mapa["nula"]=0;

  mapa["cislo"]=mapa["dolezite cislo"]; //zmena cisla za ine
  cout<<"mapa[\"cislo\"]="<<mapa["cislo"]<<endl; //pristup k  cislu pod textom
  cout << "v mape su " << mapa.size() << " prvky."<<endl; //pocet prvkov pola
  cout<<"nula je v mape "<<mapa.count("nula")<<" krat!"<<endl;
  //mapa.count vracia 0 alebo 1 podla toho, ci uz bolo k danemu klucu priradene desatinne cislo alebo nie

  return 0;
}
//vystup:
//  mapa["cislo"]=42.47
//  v mape su 3 prvky.
//  nula je v mape 1 krat!
```

Viac o `map` v C++ sa môžete dočítať online, napríklad http://www.cplusplus.com/reference/map/map/.

V Pythone sa asociatívne pole volá `dict`. Základy jeho používania si môžete pozrieť v nasledujucom zdrojovom kóde:

Listing programu (Python)

```
D=dict() #nazov pre asociativne pole v Phytone
D["cislo"]=11.5 #ulozenie desatinneho cisla pod text
D["dolezite cislo"]=42.47
D["nula"]=0

D["cislo"]=D["dolezite cislo"] #zmena cisla za ine
print("D[\"cislo\"]=", D["cislo"])
print("V D su",len(D), "prvky.") #pocet prvkov v dict
print("Je nula v D?", "nula" in D) #zistovanie ci uz prvok je v dict

# Vystup po spusteni:
#  D["cislo"]= 42.47
#  V D su 3 prvky.
#  Je nula v D? True
```

Viac o `dict` v Python sa môžete dočítať online, napríklad https://www.tutorialspoint.com/python/python_dictionary.htm.

Keďže v asociatívnom poli si pamätáme iba prvky, ktoré skutočne potrebujeme, tak ak doňho vložíme n prvkov, jeho pamäťová zložitosť bude $O(n)$. Samozrejme, má to aj nevýhodu a tou je časová zložitosť. Každá operácia s asociatívnym poľom nám trvá logaritmicky dlho od počtu jeho prvku. V našom prípade teda $O(\log n)$, čo vedie k celkovej časovej zložitosti $O(n \log n)$, pretože pre každý prvok našich prefixových súčtov sa raz pozrieme do asociatívneho poľa.

Napriek tomu je však asociátívne pole pomerne užitočné, ak však môžete použiť klasické pole, tak to spravte, predsa len to bude rýchlejšie.

Listing programu (C++)

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

int main()
{
    //musime si pamatat cisla v long longoch lebo mozu byt az do 10^11
    long long   n,   //pocet usekov
                a, b,   //hranice aktualneho intervalu
                max_a, max_b, max_len = -1; //v troch premennych max_len, max_a a max_b si budeme
                                            //pamatat dlzku najdlhsieho doteraz videneho useku
    cin >> n; //nacitame pocet usekov

    vector<long long>rozdiely(n);
    for(long long i=0;i<n;++i)
    {
        cin >> rozdiely[i]; //ulozime vyskove rozdiely do pola (vector-u)
    }
```

```
    //vytvorime si prefixove pole prefix
    vector<long long>prefix(n+1);
    prefix[0] = 0;
    for(long long i=1;i<=n;++i)
    {
        prefix[i] = prefix[i-1] + rozdiely[i-1];
    }

    //vytvorime si mapu mapa
    map<long long, long long> mapa;

    for(long long i=0;i<=n;++i)
    {
        if ( mapa.count(prefix[i]) == 0) //ak tu tento prefix este nebol
        {
            mapa[prefix[i]] = i; //tak potom je prvy a treba ho dat do mapy
        }
        else
        {//ak tu uz bol tak treba uvazovat dvojicu medzi nim a prvkom na ktory sa pozerame
            a = mapa[prefix[i]];
            b = i;  //pozrieme sa na zaciatok a koniec sucasneho intervalu

            if(b-a+1 > max_len) //ak je dlhsi ako doteraz najdlhsi najdeny, tak si ho zapiseme
            {
                    max_a = a;
                    max_b = b-1;
                    max_len = max_b-max_a+1;
            }
        }
    }

    cout<<max_len<<endl; //vypiseme najdlhsi
    if(max_len != -1) //ak vobec existuje
        cout << max_a << "␣" << max_b << endl;
    return 0;
}
```

Listing programu (Python)

```
n = int(input()) #nacitame pocet usekov

rozdiely = [int(x) for x in input().split('␣')] #ulozime vyskove rozdiely do pola (list-u)

#vytvorime si prefixove pole prefix
vyska=0
prefix = [0]
for i in range(n):
    vyska+=rozdiely[i]
    prefix.append(vyska)

mapa = dict() #vytvorime si mapu/dict

# v troch premennych max_len, max_a a max_b si budeme
# pamatat dlzku najdlhsieho doteraz videneho useku
max_len=-1
max_a=0
max_b=0

#hranice aktualneho intervalu
a=0
b=0

for i in range(n+1):
    if prefix[i] not in mapa: #ak tu tento prefix este nebol tak potom je prvy a treba ho dat do mapy
        mapa[prefix[i]]=i
    else:
        a=mapa[prefix[i]]
        b = i; #ak tu uz bol tak treba uvazovat dvojicu medzi nim a prvkom na ktory sa pozerame

        if b-a+1 > max_len: #ak je dlhsi ako doteraz najdlhsi najdeny, tak si ho zapiseme
            max_a = a;
            max_b = b-1;
            max_len = max_b-max_a+1;

print(max_len) #vypiseme najdlhsi
if max_len != -1: #ak vobec existuje
    print(max_a, max_b)
```