Korespondenční Seminář z Programování

SOUTĚŽ KASIOPEA

26. ročník | **Vzorová řešení** | **Duben 2014**

V tomto textu naleznete popis a zdůvodnění autorských řešení úloh online soutěže Kasiopea 2014, která proběhla o víkendu 22. – 23. března. Veškeré informace o soutěži včetně výsledků, zadání, vzorových zdrojových kódů a dalších informací naleznete na stránce soutěže: *http://ksp.mff.cuni.cz/akce/kasiopea/2014/*

V případě nejasností kolem těchto řešení či jiných podnětů neváhejte napsat na naši adresu *kasiopea@ksp.mff.cuni.cz*.

26-K-1 Néreidky — 10 bodů

Úkolem bylo ze seznamu trojúhelníků zadaných souřadnicemi vybrat ten, který nabývá největšího obsahu. Označme si vrcholy trojúhelníka jako $A = [x_1, y_1]$, $B = [x_2, y_2]$ a $C = [x_3, y_3]$.

Pravoúhlé trojúhelníky

Ve vstupech za polovinu bodů byly všechny trojúhelníky pravoúhlé s odvěsnami AB a AC. Obsah takového trojúhelníka lze vypočítat jako

$$S = \frac{|AB| \cdot |AC|}{2} = \frac{\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \cdot \sqrt{(x_1 - x_3)^2 + (y_1 - y_3)^2}}{2}.$$

Délky stran AB a AC jsme spočetli pomocí Pythagorovy věty. Protože nás nezajímají skutečné obsahy, ale pouze jejich vzájemné porovnání, můžeme si ušetřit práci a neodmocňovat ani nedělit dvěma. Ve skutečnosti tak budeme počítat hodnotu $2S^2$.

Dovolit si to můžeme z toho důvodu, že umocnění i vynásobení dvojkou jsou na nezáporných číslech rostoucími funkcemi. Pro nezáporná a, b je ekvivalentní $a < b$ s $2a^2 < 2b^2$. Díky tomu si při výpočtu vystačíme pouze s celými čísly.

Obecné trojúhelníky

Obsah by šlo spočítat třeba pomocí Heronova vzorce, zde si ale ukážeme postup, který vede k jednoduššímu vzorečku.

Podobně jako u pravoúhlých trojúhelníků se pokusíme najít způsob, jak si vystačit pouze s celými čísly. Místo počítání obsahu tohoto trojúhelníka budeme počítat obsah rovnoběžníku daného stranami AB a AC. Konkrétně myslíme rovnoběžník $ABCD$, kde bod D získáme jako průsečík rovnoběžky se stranou AB vedenou bodem C a rovnoběžky se stranou AC vedenou bodem B. Obsah rovnoběžníku $ABCD$ je přirozeně dvojnásobkem obsahu trojúhelníka ABC.

Abychom se dostali k rozumným výrazům, posuneme si rovnoběžník do nuly. Tedy zavedeme dvojici vektorů $u = B - A$ a $v = C - A$. Jejich složky označme jako $u = (x_u, y_u) = (x_2 - x_1, y_2 - y_1)$ a $v = (x_v, y_v) = (x_3 - x_1, y_3 - y_1)$. Vzoreček pro spočítání obsahu rovnoběžníku daného těmito vektory je $|x_u y_v - y_u x_v|$.

Vzoreček si můžete přímo odvodit, když si celou situaci nakreslíte. Také jej můžete odvodit jako determinant matice, jejímiž řádky budou souřadnice vektorů u a v, či jej odvodit pomocí vektorového součinu. (Tyto metody dávají obsah se znaménkem, je potřeba přidat ještě absolutní hodnotu.)

S využitím vzorečku získáváme obecný vzorec pro obsah trojúhelníka ABC:

$$S = \frac{|(x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1)|}{2}$$

Stejně jako u pravoúhlých trojúhelníků můžeme v programu počítat rovnou s dvojnásobkem a vystačíme si tak s celými čísly.

26-K-2 Permutace — 10 bodů

Tato úloha po vás vyžadovala generování všech permutací předepsané délky, které neobsahují zakázanou podpermutaci. Jako podpermutaci dané permutace jsme označovali libovolnou posloupnost, kterou z dané permutace získáme vyškrtáním některých prvků. Tedy pro permutaci 3214 jsou 32, 34 i 314 jejími podpermutacemi, kdežto 231 není její podpermutací.

Úlohu vyřešíme tak, že vygenerujeme všechny permutace a pak z nich vybereme ty, které neobsahují zakázanou podpermutaci.

Generování podpermutací

V devíti vstupech bylo $N \leq 6$, což je tak málo, že se permutace daly generovat skoro jakkoliv. Třeba generováním všech posloupností délky N a následného ověřování, které z nich jsou permutacemi. Pro získání posledního desátého bodu jste museli zvládnout $N = 9$, zde už bylo potřeba přijít s efektivním řešením.

Abychom vůbec měli přehled o tom, které permutace jsme již vygenerovali a které ne, hodí se zavést si nad permutacemi nějaké uspořádání. Použijeme lexikografické uspořádání (tedy přesně to, kterým jsou uspořádána hesla ve slovníku). Prvních pět permutací množiny $\{1, 2, 3, 4, 5\}$ v lexikografickém uspořádání tvoří permutace 12345, 12354, 12435, 12453 a 12534. Posledními pěti jsou pak permutace 54132, 54213, 54231, 54312 a 54321.

Vysvětlíme si, jak napsat funkci, která danou permutaci přemění na následující permutaci v lexikografickém pořadí. Navíc nás funkce u poslední permutace upozorní na to, že už máme skončit.

Označme si jako $p_0, \ldots, p_{N-1}$ permutaci, pro kterou chceme nalézt další permutaci. Onu další permutaci označme jako $q_0, \ldots, q_{N-1}$.

Nechť α je index prvního prvku, ve kterém se obě permutace budou lišit. Tedy pro všechna $i < \alpha$ bude platit $p_i = q_i$. Protože chceme další permutaci, musí být $p_\alpha < q_\alpha$. Z povahy lexikografického uspořádání přímo plyne, že index α musí být nejvyšší možný.

Uvažujme, jak α najít. Pokud by platilo $p_{N-1} < p_{N-2} < \cdots < p_\alpha$, není možné vybrat q_α tak, aby byla splněna nerovnost $p_\alpha < q_\alpha$. Jako q_α může totiž být zvolen

pouze některý z prvků p_i pro $i > \alpha$. Zvolíme tedy α jako největší index, pro který platí $p_\alpha < p_{\alpha+1}$. Pak už bude zaručeno, že existuje prvek, kterým půjde p_α nahradit.

Jak nyní vybrat q_α? Vybíráme jej z prvků $p_{\alpha+1}, \ldots, p_{N-1}$ a přirozeně vybereme nejmenší z těch, které jsou větší než p_α. Označme jej jako p_β. Všimněme si, že zbývající prvky splňují nerovnosti $p_{\alpha+1} > \cdots > p_{\beta-1} > p_\alpha > p_{\beta+1} > \cdots > p_{N-1}$.

Jak máme tyto zbývající prvky původní posloupnosti uspořádat? Protože generujeme nejbližší permutaci v lexikografickém pořadí, chceme je dostat do vzestupného pořadí. Díky předchozím nerovnostem získáme $q_{\alpha+1}, \ldots, q_{N-1}$ tak, že prvky $p_{\alpha+1}, \ldots, p_{\beta-1}, p_\alpha, p_{\beta+1}, \ldots, p_{N-1}$ zapíšeme v opačném pořadí.

Všimněme si, že pro novou permutaci nemusíme alokovat žádný prostor, místo toho můžeme rovnou přepisovat tu původní.

Celý postup si zrekapitulujme: Nalezneme α největší index takový, že $p_\alpha < p_{\alpha+1}$, pokud neexistuje, už jsme u poslední posloupnosti. Z prvků $p_{\alpha+1}, \ldots, p_{N-1}$ vybereme nejmenší p_β splňující $p_\beta > p_\alpha$. Prohodíme prvky na pozicích α a β, celou podposloupnot $p_{\alpha+1}, \ldots, p_{N-1}$ obrátíme.

Efektivita generování

Jaká je efektivita popsaného algoritmu? Hned vidíme, že přechod k další permutaci potřebuje pouze konstantně mnoho operací na každý prvek permutace, tedy můžeme odhadnout časovou složitost přechodu k další permutaci jako $\mathcal{O}(N)$. Tím bychom získali odhad $\mathcal{O}(N \cdot N!)$ na časovou složitost vygenerování všech permutací. To by pro účely úlohy stačilo. Náš odhad celkové složitosti totiž nebude záviset na délce zakázané podpermutace a pro $D = 0$ musíme vypsat všechny permutace, což samo o sobě stojí čas $\Theta(N \cdot N!)$.

Když už jsme si dali práci se samotným algoritmem, můžeme si dát ještě trochu více práce a zjistit, jak dobrý algoritmus jsme to vlastně popsali. Využijeme k tomu takzvanou penízkovou metodu.

Nejdříve si musíme zavést náš peněžní systém. Konkrétně aktuální počet korun, kterými disponujeme, vyjádříme jako $\Phi = \sum_{i=0}^{N-1} k_i$, kde

$$k_i = \begin{cases} 1, & \text{pokud } p_i > p_{i+1}, \\ 0, & \text{pokud } p_i < p_{i+1}. \end{cases}$$

Jinými slovy změnou nerovnosti $<$ na $>$ jednu korunu získáme, naopak za změnu $>$ na $<$ jednu korunu zaplatíme. Jak to souvisí s naším algoritmem? Při jednom kroku algoritmu otočíme posloupnost $p_{\alpha+1}, \ldots, p_{\beta-1}, p_\alpha, p_{\beta+1}, \ldots, p_{N-1}$, tedy zaplatíme $N - 2 - \alpha$ korun. To je ale přesně složitost jednoho kroku algoritmu! (To bychom měli možná formulovat o něco opatrněji. Z našeho postupu přímo vyplývá, že časová složitost jednoho kroku leží ve složitostní třídě $\Theta(N - \alpha)$. A funkce $N - 2 - \alpha$ v této třídě leží také, což nám pro tuto analýzu plně postačuje.)

Na začátku máme 0 korun a nemůžeme se zadlužit, časová složitost algoritmu je tedy nejvýše tak vysoká jako celkový počet korun, které vyděláme. Na každé permutaci

můžeme vydělat nejvýše jednu korunu, protože změna nerovnosti $<$ na nerovnost $>$ může nastat pouze mezi prvky na pozicích α a $\alpha + 1$.

Tak získáváme odhad na časovou složitost vygenerování všech permutací $\mathcal{O}(N!)$. Protože ale na každé permutaci provedeme alespoň jednu operaci, zjišťujeme, že tato složitost je přesně $\Theta(N!)$.

Pro praktické účely se hodí vědět, že třeba jazyk C++ ve své standardní knihovně nabízí funkci `std::next_permutation`, která dělá totéž jako naše funkce a pracuje stejně efektivně.

Hledání zakázané podpermutace

Potřebujeme zjistit, jestli permutace $p_0, \ldots, p_{N-1}$ neobsahuje jako svou podpermutaci $q_0, \ldots, q_{L-1}$. To ale znamená pouze zjistit, jestli pozice prvků $q_0, \ldots, q_{L-1}$ v naší permutaci tvoří rostoucí posloupnost.

V čase $\mathcal{O}(N)$ to můžeme udělat zhruba takto: Začneme s hodnotou $i_0 = 0$. A navyšujeme i_0, dokud neplatí $q_0 = p_{i_0}$. Pak položíme $i_1 = i_0 + 1$ a opět tuto hodnotu navyšujeme, dokud neplatí $q_1 = p_{i_1}$, obdobně hledáme $i_2, i_3, \ldots, i_{L-1}$.

Pokud najdeme všechny hodnoty $i_0, \ldots, i_{N-1}$, máme zakázanou podpermutaci. Pokud naopak libovolné i při hledání přesáhne $N - 1$, pak se zakázaná podpermutaci v naší permutaci neobjevuje.

Všech permutací je $N!$, na vygenerování, zpracování a případné vypsání permutace potřebujeme čas $\mathcal{O}(N)$, naše řešení má tedy celkovou časovou složitost $\mathcal{O}(N \cdot N!)$.

26-K-3 Bezpečné království — 10 bodů

Způsobů, jak se dalo úlohu řešit, bylo více, mohli jste si vyhrát s náhodnými algoritmy či zkusit použít techniku lokálního prohledávání. Zde si předvedeme řešení pomocí *hladového algoritmu*. Jako hladové označujeme takové algoritmy, které se ve svém rozhodování neohlíží příliš dopředu. Místo toho se rozhodnou pro řešení, které se aktuálně zdá býti nejlepším, i když ještě třeba nezpracovali všechna data. Nejslavnějším příkladem hladového algoritmu je Kruskalův algoritmus pro nalezení nejmenší kostry ohodnoceného grafu.

Hladový algoritmus

Na začátku si u všech měst poznačíme, že jsme jim ještě žádného strážného neurčili. Města nyní postupně zpracováváme (na pořadí nezáleží). Označme si aktuálně zpracovávané město jako m. Pro každý rod si spočteme, kolik sousedů města m má svého strážce z tohoto rodu. Vybereme rod, pro který je tento počet nejmenší (pokud jich je více se stejným počtem, volíme libovolně). Strážce z tohoto rodu přidělíme městu m.

Jak vhodně implementovat algoritmus? Kdybychom pro každý vrchol prošli všechny sousedské dvojice, dostali bychom se na složitost $\Theta(NM)$. Místo toho si pro každý vrchol na začátku uložíme seznam jeho sousedů, tak se v průběhu volby rodů na každou sousedskou dvojici podíváme pouze dvakrát. Počty měst z jednotlivých rodů si uchováme v poli délky K. Pokud budeme toto pole pro každé město pokaždé

nulovat, dosáhneme celkové složitosti $\Theta(NK + M)$. Takovéto řešení postačuje, můžeme však být pečlivější a nenulovat pole celé, ale pouze ty jeho pozice, kam jsme něco zapsali. Pak dosáhneme časové složitosti $\Theta(N + M + K)$. Protože je $K \leq N$, můžeme ji zapsat pouze jako $\Theta(N + M)$. To je asymptoticky optimální. $\Theta(M)$ je totiž velikost vstupu, který musíme přečíst, $\Theta(N)$ je zase velikost výstupu.

Prostorová složitost vychází stejně.

Důkaz správnosti

Označme si jako D_m množinu všech dvojic měst takových, že obsahují město m a že v době zpracování města m má druhé město již přiděleného strážného. Jako R_m označme počet dvojic měst z D_m takových, že obě města takovýchto dvojic mají strážné z různých měst. Jako R označme počet všech dvojic měst, které mají oba strážné z různých rodů. Potřebujeme dokázat $R \geq \frac{K-1}{K} \cdot M$.

Předpokládejme, že jsme městu m přidělili strážného z rodu k. Z volby rodu k vyplývá, že nejvýše $\frac{|D_m|}{K}$ dvojic měst z D_m má strážného z rodu k. Tak získáváme odhad $R_m \geq |D_m| - \frac{|D_m|}{K} = \frac{K-1}{K} \cdot |D_m|$.

Sečteme-li tyto nerovnice přes všechna města, získáme:

$$R = \sum_{m \in \{1,\ldots,N\}} R_m \geq \sum_{m \in \{1,\ldots,N\}} \frac{K-1}{K} \cdot |D_m| = \frac{K-1}{K} \cdot \sum_{m \in \{1,\ldots,N\}} |D_m| = \frac{K-1}{K} \cdot M$$

To je přesně odhad, který jsme potřebovali. Důkaz je tímto hotov.

26-K-4 Městský ruch — 10 bodů

Stejně jako v zadání budeme počet řádků a sloupců mapy města značit jako R a S. Jako K označíme počet druhů záležitostí, které musí poddaní vyřídit. Tato hodnota byla v úloze nastavena pevně na $K = 10$ a i ve vzorovém programu je tato konstanta pevně „zadrátována“. Pro analýzu algoritmů ale bude vhodnější si tuto konstantu označit obecně, abychom viděli, jaký vliv její velikost má.

Pomalé řešení

Úloha od pohledu spadá do kategorie hledání nejkratších cest. A skutečně můžeme z každého políčka spustit prohledávání do šířky, abychom pro každou záležitost nalezli nejbližší místo, kde ji lze vyřídit. Průchod do šířky má obecně časovou složitost $\mathcal{O}(|V| + |E|)$, kde $|V|$ značí počet vrcholů prohledávaného grafu a $|E|$ počet hran. V našem případě tedy průchod do šířky zabere čas $\mathcal{O}(RS)$.

Políček mapy je RS, výsledný algoritus tak dosahuje časové složitosti $\mathcal{O}(KR^2S^2)$ nebo $\mathcal{O}(R^2S^2)$ v závislosti na konkrétní implementaci.

Ukážeme si, že úlohu dokážeme vyřešit i mnohem rychleji. Jak bychom ale v tomto okamžiku mohli vytušit, že náš algoritmus je zbytečně pomalý? Všimněme si, že najít nejkratší cestu je v naší úloze triviální – délka nejkratší cesty mezi políčky $[a, b]$ a $[c, d]$ je vždy dána výrazem $|a - c| + |b - d|$ a snadno nějakou cestu takové

délky najdeme. Tedy asi budeme hledat něco trochu jiného než standardní algoritmy pro hledání nejkratších cest.

Podstatné pozorování je toto: Zafixujme si pevně aktivitu A a hledejme pro nějaké políčko $[r, s]$ nejbližší políčko s aktivitou A. Pokud je aktivita A přímo na políčku $[r, s]$, je to triviální, hledané políčko je rovnou $[r, s]$. Předpokládejme nyní, že jsme nejbližší vhodné políčko našli pro všechna políčka sousedící s $[r, s]$. Potřebujeme nyní spouštět znovu celý průchod do šířky?

Nepotřebujeme! K onomu hledému políčku určitě půjdeme přes některé ze sousedních políček, stačí se tedy podívat na sousední políčka a zjistit, která políčka s aktivitou A jsou jim nejblíž, z těchto políček si vybereme to, které je nejblíž políčku $[r, s]$.

Rychlé řešení

Myšlenka, ke které jsme se vás snažili navést v předchozím odstavci, patří mezi standardní techniky a označujeme ji jako *dynamické programování*. Je to velmi obecné označení pro různé techniky založené na tom, že si zkrátíme výpočet šikovným využitím již spočítaných hodnot. Pokud tuto techniku neznáte, vřele doporučujeme si někdy vyhradit chvilku na příslušnou kuchařku.[1]

Úlohu vyřešíme pro každou aktivitu zvlášť a jako výsledek pak vrátíme součet přes všechny aktivity. Zvolme si pevně aktivitu A, pro kterou budeme vzdálenosti počítat.

Pro začátek uvažujme zjednodušenou verzi, ve které se poddaní mohou pohybovat pouze směrem nahoru a vlevo a ve které mohou aktivitu vyřídit i na svém vlastním políčku. Jako $D_{r,s}$ označme vzdálenost políčka $[r, s]$ od nejbližšího políčka s aktivitou A.

Chceme spočítat $D_{r,s}$ pro všechna r a s. Pro políčko $[0, 0]$ je výpočet snadný. Pokud se nachází na tomto políčku aktivita A, pak $D_{0,0} = 0$, jinak je aktivita A nedostupná, to vyjádříme jako $D_{0,0} = \infty$.

Uvažme nyní políčko $[r, s]$. Pokud je na tomto políčku aktivita A, opět $D_{r,s} = 0$. V opačném případě $D_{r,s} = 1 + \min(D_{r-1,s}, D_{r,s-1})$, vzoreček přesně vyjadřuje to, že si můžeme vybrat krok nahoru nebo vlevo, pak už můžeme použít výsledek políčka, na které dojdeme. (Poznámka: Pro případ, kdy je $r = 0$, použijeme přirozeně $D_{0,s} = 1 + D_{0,s-1}$ a obdobně pro $s = 0$.)

Důležité je, že když budeme políčka procházet postupně po řádcích a v každém řádku začneme v nultém sloupci, budeme mít v okamžiku, kdy počítáme $D_{r,s}$ již dostupné hodnoty $D_{r-1,s}$ a $D_{r,s-1}$. V čase $\Theta(RS)$ tedy spočteme všechna $D_{r,s}$.

Uvědomme si, že obdobně můžeme vyřešit i varianty, kdy se pohybujeme pouze doprava a nahoru, nahoru a vlevo nebo vlevo a dolů. Akorát musíme trochu poupravit parametry, ukažme si to na případu, kdy se pohybujeme pouze doprava a nahoru. Jako počáteční políčko nyní zvolíme políčko v pravém horním rohu, tedy to na souřadnicích $[0, S - 1]$. „Dynamický výpočet“ nyní budeme provádět jako

[1] `http://ksp.mff.cuni.cz/viz/kucharky/dynamicke-programovani`

$D_{r,s} = 1 + \min(D_{r-1,s}, D_{r,s+1})$, políčka opět procházíme po řádcích, políčka jednoho řádku ale nyní musíme procházet odzadu.

Když už budeme mít spočteny všechny čtyři možnosti pohybu, budeme se moci dostat z každého políčka na libovolné jiné políčko. Tedy vzdálenost k nejbližšímu políčku s aktivitou A bude minimem ze vzdáleností pro tyto čtyři možnosti pohybu.

A jak se vypořádáme s tím, že poddaní nemohou použít své vlastní políčko? Krom $D_{r,s}$ si v každém směru budeme ještě počítat $D'_{r,s}$, které budeme rovnou počítat jako $1 + \min(D_{r-1,s}, D_{r,s-1})$, ať už je na poli $[r, s]$ aktivita A nebo ne. (Pro různé směry samozřejmě místo $[r-1, s]$ a $[r, s-1]$ použijeme jiné sousedy.)

Výsledná vzdálenost pak bude minimum ze všech $D'_{r,s}$ pro čtyři různé směry.

Protože tento postup opakujeme pro každou aktivitu, získali jsme řešení o časové složitosti $\mathcal{O}(KRS)$. Prostorová složitost je $\mathcal{O}(RS)$, protože nám stačí uchovávat pouze konstantně mnoho polí velikosti RS.

26-K-5 Mýtický strom — 10 bodů

Operaci magického navyšování hodnot vrcholu označujme prostě jako přičítací operaci. Jako syny vrcholu budeme označovat vrcholy, které jsou s ním spojeny s hranou a jsou „pod ním“. Jako potomky vrcholu budeme označovat jeho syny, syny jeho synů, syny synů jeho synů atd.

Úlohu lze řešit přímočaře. Přičítací operaci můžeme realizovat prostě tak, že všechny potomky navštívíme a upravíme jejich hodnotu, například průchodem do hloubky. Při tom si akorát musíme udržovat informaci o tom, zda jsme na sudé nebo liché hladině, abychom věděli, zda máme přičítat nebo odčítat.

Takto na přičítací operaci potřebujeme čas $\mathcal{O}(N)$. Vypisovací operace proti tomu stojí pouze $\mathcal{O}(1)$, prostě vypíšeme hodnotu, kterou už máme dávnou spočtenou.

V případech, kdy je přičítacích operací alespoň tolik co vypisovacích, se dostáváme na složitost $\mathcal{O}(NM)$. Za takové řešení se dalo získat pět i více bodů v závislosti na tom, jak jste jej naprogramovali. Efektivnější řešení, které zvládá obě operace v čase $\mathcal{O}(\log N)$, si nyní předvedeme.

Efektivní řešení

Pro přehlednost si řešení popišme pro začátek pro případ, kdy přičítací operace nestřídá znaménka, ale prostě ke všem potomkům (i vrcholu samotnému) přičte zadané x. Úprava pro přičítací operaci střídající znaménka už nebude obtížná.

Abychom mohli vrcholy rozumně zpracovávat hromadně, potřebujeme si je nejprve vhodně uspořádat. K tomu se nám bude velmi hodit průchod do hloubky. Představme si, že při tomto průchodu máme časomíru. Na začátku nastavme $T = 0$ a při každém vstupu do nového vrcholu ji navyšme o jedna. Po vstupu do vrcholu v si aktuální T uložme jako číslo vrcholu, budeme jej značit L_v. Při výstupu z vrcholu si aktuální T uložme jako R_v. Všimněme si, že všichni potomci vrcholu v mají čísla $L_v + 1$ až R_v.

Od tohoto okamžiku se už o původním stromu nemusíme vůbec bavit. Celá přičítací operace se redukuje na přičtení dané hodnoty ke všem prvkům určitého intervalu.

K tomu se, jak název napovídá, budou hodit intervalové stromy. Seznámit se s nimi můžete v intervalové kuchařce.[2] Konkrétně využijeme Fenwickův strom operající nad hodnotami $h_1, \ldots, h_N$. Ten umí v čase $\mathcal{O}(\log N)$ dvě operace. První operací je pro dané i navýšení h_i o předepsanou hodnotu. Druhou operací je pro dané i určení součtu $h_1 + \cdots + h_i$.

Hodnotu vrcholu v bude reprezentovat součet $h_1 + \cdots + h_{L_v}$. Jak teď realizovat přičítací operaci? Chceme-li přičíst x ke všem vrcholům s čísly p až q, navýšíme jednoduše h_p o x a naopak snížíme h_{q+1} o x. Hodnoty vrcholů před p se tak nijak nezmění. Hodnoty vrcholů za q se také nijak nezmění, protože se vyruší x a $-x$. Hodnoty vrcholů od p po q se zvýší o x. Operaci tedy provádíme korektně.

Tak jsme získali řešení, které obě operace (přičítací i vypisovací) provádí v čase $\mathcal{O}(\log N)$. Upravit toto řešení i pro magickou operaci, která střídá znaménka by už mělo být snadným cvičením. Opět bude potřeba vhodně rozlišovat sudé a liché hladiny jako u přímočarého řešení. Může se také hodit použít dva Fenwickovy stromy.

Pokud byste tuto úpravu nevymysleli, podívejte se do autorského zdrojového kódu.

26-K-6 Zajímavá posloupnost — 10 bodů

Způsobů, jak tuto úlohu vyřešit, je více. Pravděpodobně by šlo použít nějakou verzi dvourozměrného intervalového stromu a vyřešit tak úlohu v čase $\mathcal{O}(n \log^2 n)$. To my ale dělat nebudeme, my si ukážeme autorské řešení, které používá jen základní programátorské techniky.

Nejdříve si posloupnost v čase $\mathcal{O}(n \log n)$ přečíslujeme na hodnoty 0 až $n-1$ a označíme ji jako P. Dále v čase $\mathcal{O}(n)$ spočítáme indexy $\texttt{next}(i)$ a $\texttt{back}(i)$ – indexy dalšího a předcházejícího prvku stejné hodnoty. Tyto hodnoty použijeme v jádru našeho algoritmu. Budeme se snažit najít podposloupnost, která neobsahuje žádný unikátní prvek.

Pro podposloupnost $[a, b]$ je prvek na indexu c ($a \leq c \leq b$) unikátní, právě když $\texttt{back}(c) < a$ a zároveň $\texttt{next}(c) > b$. Dále si všimneme, že pokud je prvek na indexu c unikátní, tak všechny podposloupnosti posloupnosti $[a, b]$ jdoucí přes c již unikátní prvek obsahují. Tedy nám stačí se zaměřit jen na podposloupnosti $[a, c-1]$ a $[c+1, b]$.

Myšlenka algoritmu je jednoduchá: v posloupnosti $[0, n-1]$ zkusíme najít unikátní prvek. Pokud se nám to nepovede, posloupnost není zajímavá. Nechť se nám to podaří a unikátní prvek najdeme na indexu c. Pak stejný algoritmus znova zavoláme na příslušné dvě podposloupnosti.

Teď už nám jen zbývá vyřešit, jak hledat unikátní prvek. První nápad je, že budeme procházet jeden prvek po druhém a ověřovat, jestli náhodou není unikátní. Při

[2] `http://ksp.mff.cuni.cz/viz/kucharky/intervalove-stromy`

prvním nalezeném se rekurzivně zavoláme na levou a pravou podposloupnost. V nejhorším případě se nám stane, že nalezený prvek bude vždy až u konce posloupnosti a to nám způsobí v nejhorším případě časovou složitost $\mathcal{O}(n^2)$.

Kdybychom ale unikátní prvek vždy našli v první polovině, tak nás rozdělení posloupnosti bude vždy stát čas $\mathcal{O}(\textit{velikost menší části})$, což nás dostane na čas $\mathcal{O}(n \log n)$, protože v nejhorším případě za $\mathcal{O}(n)$ kroků rozdělíme posloupnost na poloviny.

My ale vždycky můžeme najít unikátní prvek v čase $\mathcal{O}(\textit{velikost menší části})$. Stačí jen nehledat prvek pouze z jedné strany, ale postupovat z obou stran zároveň. Pak se podíváme maximálně na dvakrát tolik prvků, kolik obsahuje menší část po rozdělení. A to je celý trik této úlohy!

Prvních pět úloh připravil a jejich řešení sepsal Lukáš Folwarczný, šestou úlohu zpracoval Karel Tesař.