Korespondenční Seminář z Programování

ZAČÁTEČNICKÁ KATEGORIE

32. ročník | **KSP-Z** | **Leden 2020**

Řešení druhé série začátečnické kategorie 32. ročníku KSP

32-Z2-1 Prší

Po přečtení zadání a prohlédnutí si ukázkového *obrázku* v něm (jestli se tak vůbec znázornění výšek jednotlivých částí dá nazývat) vás mohlo napadnout si pro vyřešení úlohy také takový zkonstruovat (třeba v nějakém dvourozměrném poli). To ale vůbec není potřeba a také by se mohlo stát, že si třeba pro vstup o třech číslech (například 54321, 10 a 65432) budete potřebovat vyrobit obrovské dvourozměrné pole o více jak 100 tisících prvků… a to je trochu zbytečné.

Počítač si zadání ale nemusí nijak vizualizovat, stačí navrhnout dobrý postup jak zjistit kapacitu prohlubní ve skalce jen ze samotných čísel. Zkusíme si v každém sloupci stanovit výšku, do které v něm bude sahat voda (aby se nám s tím dobře počítalo, tak to vždy bude minimálně výška kamenné části).

Představme si pro ukázku skalku tohoto tvaru (pro vstup `2 1 4 1 0 3`):

```
  X
  X  X
X X  X
XXXX X
```

Nejprve si zkusíme úlohu zjednodušit – co kdyby v našem světě platily speciální fyzikální zákony a voda mohla vytékat jenom doleva? V tom případě můžeme projít výšky sloupců zleva a budeme si postupně pro každý sloupec určovat výšku hladiny, která se v něm zadrží. Vždy to bude maximum z výšek sloupců, které jsme zatím potkali (když mám sloupec výšky dva a někde více vlevo jsem již potkal sloupec výšky deset, tak vím, že směrem doleva nevyteče hladina výšky deset).

Pro naši ukázkovou skalku tak spočítáme hladinu ve sloupcích při vodě tekoucí pouze doleva jako `2 2 4 4 4 4`. Stejně tak můžeme spočítat (při průchodu z druhé strany), jaká by byla výška hladiny kdyby v našem světe tekla voda pouze doprava. To bychom dostali výšky hladin naopak `4 4 4 3 3 3`.

Jenže v našem světě teče voda doleva i doprava současně, co s tím? Spojíme obě doposud získané informace do jedné, kde voda může vytéci doleva i doprava. To uděláme tak, že pro každý sloupec vezmeme to menší číslo z obou hladin (když se nám při vytékání doleva udrží hladiny pět a při vytékání doprava jenom tři, tak při vytékání na obě strany se nám udrží hladina pouze tři). V našem případě tak vyjde výsledná posloupnost `2 2 4 3 3 3`.

Teď už nám zbývá poslední krok a to odečíst od výšky hladiny kamenné části skalky (tím nám zůstane posloupnost hloubky vody v jednotlivých sloupcích) a nakonec jenom všechnu vodu sečíst. Pro náš příklad dostaneme posloupnost `0 1 0 2 3 0` a celkový objem vody 6 litrů.

Když si celý postup v krocích zopakujeme, tak:

- Spočítáme hladinu pro vodu tekoucí jen doleva
- Spočítáme hladinu pro vodu tekoucí jen doprava
- Spočítáme minimum z obou možností pro každý sloupec
- Odečteme výšku kamenné části skalky
- Sečteme zbylá čísla

Každý krok zvládneme spočítat na jeden průchod vstupními hodnotami (tedy *lineárně* k velikosti vstupu – pro vstup délky N to značíme $\mathcal{O}(N)$) a průchodů je v tomto případě maximálně pět, takže můžeme říci, že celou úlohu umíme vyřešit v lineárním čase k velikosti vstupu.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/32-Z2-1.py`

Jirka Setnička

32-Z2-2 Turnaj hada

Druhá úloha této série spočívala v tom šikovně odsimulovat pohyby jednotlivých hadů po hrací ploše. Ač to myšlenkově nebyla těžká úloha, tak implementačně naopak patřila mezi nejsložitější v sérii – pojďme si tedy postupně sestavit jednotlivé stavební bloky, díky kterým pak zvládneme celý turnaj odsimulovat.

Pohyb jednoho hada

První otázkou je, jak si pamatovat hada. První myšlenkou může být pamatovat si hada v poli (prvky budou vždy dvojice čísel představující souřadnice jednoho políčka) a při posunu jeho hlavy přepočítat pozice jednotlivých políček. Přesněji nová hlava bude mít pozici o jednu souřadnici jinde, než byla minulá hlava, druhé hadovo políčko bude tam, kde byla předtím hlava, … a obecně i-té hadovo políčko bude tam, kde bylo předtím $(i-1)$-té hadovo políčko.

To si můžeme lehce naprogramovat jako cyklus, který projde pole představující hada odzadu (rozmyslete si proč jdeme odzadu) a na současné políčko nakopíruje souřadnice z políčka s o jedna menším indexem. Takto nám jeden posun hada délky D bude trvat čas $\mathcal{O}(D)$. Neděláme ale zbytečnou práci? Nešlo by zpracovat jeden posun hada rychleji?

Pokud si namísto pole pořídíme spojový seznam, tak nám stačí jenom odebrat z konce spojového seznamu jedno políčko (ocas) a naopak na začátek spojového seznamu jedno políčko přidat (pozice nové hlavy), což jsou obě dvě konstantní operace bez ohledu na délku hada. Takové datové struktuře se říká fronta. Například v Pythonu pro frontu existuje šikovná kolekce `deque`. Další alternativou by mohlo být pořídit si cyklické pole a vždy jen pozicí nové hlavy přepsat pozici starého ocasu a posunout ukazatel na hlavu hada o jedno dál; detaily si rozmyslete.

Detekování kolizí hadů

Dobře, nyní tedy umíme simulovat jeden krok hada v čase $\mathcal{O}(1)$ a K kroků v čase $\mathcal{O}(K)$ – jak ale poznávat, jestli had náhodou nenaboural hlavou do nějakého jiného (nebo do svého vlastního těla)?

Budeme si chtít nějak pamatovat všechna použitá políčka – můžeme si pořídit třeba dostatečně velké dvourozměrné pole a na jednotlivých políčkách si pamatovat, kolik se na nich nachází hadů. Na začátku zvedneme počítadlo za každého hada na nějakém políčku o plus jedna a pak při přesunu hada odečteme jedničku od políčka, ze kterého mizí ocas, a přičteme ji k políčku, na které se přesouvá hlava. Poté již můžeme pro každého hada zkontrolovat, zdali se jeho hlava nenachází na políčku s vyšším číslem než jedna – pokud ano, tak se jeho hlava střetla s jiným hadem (nebo jinou částí jeho vlastního těla).

Drobná poznámka k tomu, proč používáme počítadlo s přičítáním a odečítáním jedniček namísto prostého `True` a `False`: v rámci zpracování jednoho kroku mohou na stejné políčko přesunout hlavy dvou různých hadů a my chceme tento stav poznat (abychom oba hady označili za mrtvé).

Hadi se ale mohou pohybovat na skutečně velké ploše a tak velké dvourozměrné pole se nám do paměti nevejde. Co s tím? Namísto prostého pole, jež indexujeme čísly, si pořídíme asociativní pole aneb hashovací tabulku (Python i spousta jiných vysokoúrovnovějších jazyků nějakou vestavěnou má, v Pythonu se tento datový typ nazývá *slovník*). Toto asociativní pole budeme indexovat pomocí souřadnic políček – to nám nezaručí rychlé použití v nejhorším případě, ale v průměrném případě budeme mít jednu operaci v konstantním čase.

Pojďme to tedy celé složit dohromady. Po načtení hadů do paměti budeme postupně simulovat jeden za druhým jednotlivé kroky – v každém kroku se všichni hadi posunou o jedno políčko, z asociativního pole odebereme pozice jejich ocasů a naopak přidáme pozice jejich posunutých hlav, a nakonec zkontrolujeme pozice hlav hadů, jestli náhodou nedošlo ke kolizi (pokud by došlo, odstraníme hada a zapamatujeme si číslo kroku). Nakonec už jen vypíšeme stav jednotlivých hadů na výstup.

Celkově zpracováváme K kroků a v každém posunujeme H hadů. Posunutí jednoho hada nám zabere v průměrném případě konstantní čas (dvě operace se spojovým seznamem daného hada a dvě operace s asociativním polem obsazených políček). Celkově tak odsimulujeme celý turnaj hada v čase $\mathcal{O}(KH)$.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/32-Z2-2.py`

Jirka Setnička

32-Z2-3 Panika v chodbě

Představme si, že budeme sledovat studenty vyděšeně pobíhající po chodbě a aby nám nic neušlo, pořádně zpomalíme čas – dokud jde jenom o myšlenkový experiment, klidně můžeme předpokládat, že jsme si čas rozkrájeli na nekonečně mnoho nekonečně krátkých „momentů“.

Většina momentů je hrozně nudná: prostě se každý student o maličko posune. Někdy se přeci jenom stane něco zajímavého: *srážka* dvou studentů, případně nějaký student *doběhne* do své třídy. Těmto zajímavým momentům budeme říkat *události.*

V programu si samozřejmě nemůžeme dovolit simulovat všech nekonečně mnoho momentů. Místo toho budeme sledovat pouze události a všechen čas mezi nimi přeskakovat.

To je dobrý plán, ale jak na to? Začneme s málem: zkusíme zjistit, jaká událost nastane jako první. Seřadíme si studenty podle polohy na chodbě. Pro každého studenta se podíváme, kterým směrem běží, jaký je jeho soused tímto směrem a kdy do souseda narazí. Podle toho naplánujeme, jakou nejbližší událost tento student způsobí: buďto srážku, nebo pokud už za ním žádný další student neleží, tak to bude doběhnutí do třídy.

(Detaily výpočtů: Nechť student s polohou p_1 má rychlost v_1, řekněme kladnou. Vpravo od něj je tedy nějaký student s polohou $p_2 > p_1$ a rychlostí v_2. Z pohledu prvního studenta má druhý polohu $p_2 - p_1$ a rychlost $v_2 - v_1$, takže pokud $v_2 - v_1 < 0$, dojde ke srážce za čas $(p_2 - p_1)/(v_1 - v_2)$, jinak k ní nedojde vůbec. Doběhnutí do třídy si představíme jako srážku se studentem o nulové rychlosti stojícím na kraji chodby.)

Jenže ouha, některé z naplánovaných událostí vůbec nenastanou! Můžeme si třeba spočítat, že student A za 10 sekund narazí do studenta B, ale ve skutečnosti ho už za 5 sekund zezadu bezostyšně srazí student C. Srážka A s B tedy doopravdy nenastane – takové události budeme říkat *virtuální.*

Nám to ale nevadí: my přeci hledáme jen úplně první z naplánovaných událostí. A ta nikdy nemůže být virtuální – před ní totiž nenastane žádné další událost, která by té první zabránila.

Jakmile najdeme první událost, můžeme zpracovat její následky: započítat a odstranit z chodby studenta, který doběhl do třídy, nebo odstranit pomalejšího ze srazivších se studentů. Pak můžeme spustit celý výpočet znovu a tím zjistit druhou skutečnou událost. A tak dále.

Jakou časovou složitost tento algoritmus má? Na začátku si studenty jednou uspořádáme podle polohy na chodbě, což pro N studentů trvá $\mathcal{O}(N \log N)$. Pak opakovaně počítáme následující událost: projdeme N studentů, u každého strávíme konstantní čas plánováním události, a pak najdeme minimum z časů naplánovaných událostí. To trvá $\mathcal{O}(N)$. A jelikož při každé skutečné události zmizí jeden student, událostí nastane právě N. Proto celý algoritmus běží v čase $\mathcal{O}(N^2)$.

Při programování ještě můžeme narazit na problémy s přesností výpočtů. Při výpočtu časů srážek dělíme a jelikož běžná desetinná čísla mají omezenou přesnost, výsledek vyjde nějak zaokrouhlený. To do časů zanese chybu, která je sice maličká, ale mohla by způsobit, že se prohodí pořadí nějakých dvou událostí. Tím by se dokonce mohlo prohodit, která z nich bude virtuální. Můžete si rozmyslet, že celkový počet zachráněných studentů se tím nezmění. Lepší je ale napsat program tak, aby místo s desetinnými čísly počítal se zlomky – například v Pythonu k tomu můžete použít typ `Fraction`.

Poznámka: Kdyby nám kvadratická složitost přišla příliš, mohli bychom algoritmus zrychlit na $\mathcal{O}(N \log N)$ tím, že bychom si naplánované události ukládali do nějaké chytré datové struktury, a pokaždé přeplánovávali jen sousedy studenta, který právě zmizel. Kdyby vás zajímalo, jak se takové věci dělají, přečtěte si kapitolu o geometrických algoritmech v Průvodci labyrintem algoritmů.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/32-Z2-3.py`

Martin „Medvěd“ Mareš

32-Z2-4 Opisování v testu

Připomeňme si značení ze zadání: T značí počet testů, N je počet otázek. Vstup si zapamatujme jako T-prvkové pole N-znakových řetězců. Ukážeme si řešení pracující v čase $\mathcal{O}(TN^2)$.

Nejprve si všimneme, že úlohu velice zjednodušuje, že hledáme úsek, na kterém se shoduje více než půlka odevzdaných řešení. Znamená to totiž, že pro každou otázku v testu existuje maximálně jedna odpověď (znak), kterou potřebujeme uvažovat (zkrátka proto, že žáci nemohli více odpovědí použít víc než v polovině případů). Té budeme říkat *dominantní* odpověď.

Dominantní odpovědi spočítáme snadno. Pro každou otázku projdeme všechny testy, zjistíme, kolikrát se která odpověď vyskytla, a podíváme se, zda některý z počtů je větší než $T/2$. K tomu se dá použít, že abeceda je konstantně dlouhá, tedy můžeme si vyrobit 26-prvkové pole čísel, které nejprve naplníme nulami a s každým znakem o jedna zvýšíme tolikátý prvek, kolikátý znak v abecedě jsme potkali. Kódy znaků v abecedě používané programovacími jazyky (takzvaný kód ASCII) jdou dokonce popořadě, takže spousta programovacích jazyků např. `'C'-'A'` vyhodnotí jako 2; v Pythonu je potřeba převést znaky na jejich kódy, takže napíšeme `ord('C')-ord('A')`. Výpočet dominantních odpovědí stihneme v čase $\mathcal{O}(TN)$.

Teď vyzkoušíme všechny úseky a najdeme nejdelší takový, v němž se nadpoloviční většina testů shodne na dominantní odpovědi. Označme i index začátku úseku a j index jeho konce. Budeme zkoušet všechna i od 0 do $N-1$. Pro každé i budeme postupně zvyšovat j od i do $N-1$ a udržovat si čísla všech testů, které se ještě shodují na dominantní odpovědi. Začínáme s prázdným úsekem, kde se shodnou všechny testy. A pro každé další j projdeme všechny testy v seznamu, podíváme se, které z nich odpověděly dominantně, a ty, které ne, ze seznamu vyřadíme. Jakmile velikost seznamu klesne pod $T/2$, přestaneme zvyšovat j a zkusíme další začátek, tedy zvýšíme i o 1.

Průběžně udržujeme, jaký nejdelší interval jsme potkali, a nakonec ho vypíšeme.

Jak dlouho tohle potrvá? Zkoušíme N začátků, pro každý z nich nejvýše N konců. A pro každou dvojici začátku a konce procházíme až T testů. Celkem tedy $\mathcal{O}(TN^2)$.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/32-Z2-4.py`

Lineární řešení

Předchozí řešení bylo dost rychlé na to, aby v rozumném čase zpracovalo naše testovací vstupy. Ale přesto ho ještě zkusíme zrychlit. Dosáhneme dokonce lineární časové složitosti (vůči velikosti vstupu), tedy $\mathcal{O}(TN)$. Pozor, bude to trochu náročnější.

Označíme D_k množinu všech testů, které na k-tou otázku odpovídají dominantně. Náš předchozí algoritmus můžeme popsat tak, že pro aktuální levý okraj i a postupně se zvětšující pravý okraj j počítá množiny X_j těch testů, které se od i-té otázky po j-tou shodnou na dominantní odpovědi. Tedy $X_j = D_i \cap D_{i+1} \cap \ldots \cap D_j$.

Pro daný levý okraj i tedy začneme s $X_i = D_i$ a $j = i$. Pak postupně zvyšujeme j a volíme X_{j+1} jako množinu všech prvků z X_j, které padly i do D_{j+1}. Zastavíme se, jakmile velikost X_j klesne pod $T/2$.

Pak nás čeká zkusit další levý okraj, tedy zvýšit i o 1. Novým množinám budeme říkat X'_j. Každá X'_j je zjevně nadmnožinou X_j (předtím nás omezovalo D_i až D_j, teď už jenom D_{i+1} až D_j). Proto všechny pravé okraje, které fungovaly předtím, fungují i teď. Tudíž můžeme ve zvyšování j pokračovat od toho místa, kde jsme předtím skončili, a nemusíme se vracet na $j = i+1$. Takhle se okraje i i j posouvají jenom doprava, takže každý z nich udělá nejvýše N kroků. A jelikož v každém krok trávíme čas $\mathcal{O}(T)$ zkoumáním testů, celkem to trvá $\mathcal{O}(TN)$.

Jen pozor na to, že každá X'_j může kromě prvků z X_j obsahovat ještě nějaké navíc: předtím jsme z D_{i+1} vynechali ty prvky, které chyběly v D_i, ale teď už je můžeme použít. Při každém přechodu od i k $i+1$ tedy musíme do množin X_j doplnit chybějící prvky. Uděláme to následovně.

Množinu X_{i+1} rozšíříme na všechny prvky z D_{i+1} a podíváme se, které prvky jsme tím přidali. U těch prozkoumáme, jestli je nemůžeme přidat i do X_{i+2} (ověříme, zda příslušné testy dávají v $(i+2)$-ní otázce dominantní odpověď). Prvky přidané do X_{i+2} zkusíme dále přidat do X_{i+3} a tak dále.

Nahlédneme, že všechna tato rozšiřování množin dohromady trvají jen $\mathcal{O}(TN)$. Všimneme si totiž, že většina práce se týká přidávání prvků do množin a jednou přidané prvky už z množin neubývají. Celkem máme N množin, v každé nejvýše T prvků, takže počet všech přidání prvků nepřekročí TN. Při každém zvyšování i vezmeme každý test z D_{i+1} a přidáváme ho do množin. Testem strávíme konstantní čas na jeho objevení, pak nějaký čas na přidávání (ten už jsme ale započítali do TN) a nakonec konstantní čas na zjištění, že už dál přidávat nejde. To dá dohromady TN plus konstantní čas na dvojici (*začátek*, *test*). Jelikož těchto dvojic je opět TN, doplňování trvá celkově také $\mathcal{O}(TN)$. Celkovou časovou složitost nám to tedy nepokazí.

Martin Koreček & Martin „Medvěd“ Mareš

32-Z2-5 Asfaltérský problém

Prvním krokem v řešení úlohy jako je tato je pochopit, co je vlastně naším úkolem. Máme náměstí plné děr, přes které chceme jednou přejet asfaltovacím autem širokým D a přitom vyasfaltovat co nejvíce děr. Takže vlastně v obdélníku plném bodů hledáme pruh široký D, ve kterém se nachází největší počet bodů.

Druhým krokem v tomto případě bude zjednodušení: Máme povoleno asfaltovat pouze rovně ve směru z východu na západ, takže v podstatě posouváme naším pruhem (asfaltovacím vozem) přes obdélník v severo-jižním směru a vyasfaltujeme všechny body, které mají severo-jižní souřadnici v našem pruhu, na druhé souřadnici vůbec nezáleží. Můžeme si tedy všechny body „splácnout“ na úsečku podle jejich severo-jižní souřadnice a na této úsečce hledáme interval velký D, ve kterém bude bodů nejvíce.

Tím už jsme úlohu dostatečně zjednodušili, abychom se mohli vrhnout přímo na její řešení. Ale jak na to? Můžeme zkusit začít s intervalem co nejvíce na jihu a postupně ho posouvat na sever – ale jak takové posouvání dělat? Tím, že body nemají celočíselné souřadnice, tak si nemůžeme udělat cyklus od nuly po velikost náměstí, ale musíme na to jinak. Můžeme například posouvat jižní začátek intervalu postupně po seřazených bodech od nejjižnějšího po nejsevernější a přitom si pro každý takový začátek spočítat, kolik bodů

leží v daném intervalu. To pro N bodů může v každém kroku zabrat čas až $\mathcal{O}(N)$ (musíme zkontrolovat každý bod) a těchto kroků bude N.

Toto řešení sice funguje (najde interval široký D, ve kterém se nachází největší možný počet bodů, protože projde přes všechny jeho možné jižní konce), ale tak, jak jsme ho popsali, zabere algoritmus čas $\mathcal{O}(N^2)$. A to je příliš pomalé.

Co nám na postupu výše trvá tak dlouho? Pro každý jižní konec intervalu procházíme všechny body stále znovu – nešlo by si pamatovat body, které jsou obsažené v posledním intervalu, a po jeho posunutí je pouze aktualizovat? Asi již tušíte, že to samozřejmě lze. Na začátku si stanovíme jižní konec intervalu jako nejjižnější bod a postupně do intervalu započítáme všechny body vzdálené maximálně o D a zapamatujeme si poslední takový bod jako koncový bod intervalu.

Při každém dalším kroku (při každém posunutí jižního konce intervalu na další bod) nejprve odečteme od počtu bodů v intervalu jedničku za právě odstraněný bod na jižním konci. Poté se podíváme na koncový bod intervalu a zkontrolujeme bod za ním, jestli se již náhodou do intervalu nevejde (a pokud ano, tak ho přidáme a zkontrolujeme i body za ním). Průběžně si přitom držíme maximum.

Výše popsaný postup sice v jednom kroku můžeme do intervalu přidat až $\mathcal{O}(N)$ bodů, ale každý bod přidáme do intervalu právě jednou a každý bod z intervalu odebereme právě jednou. Celkově provedeme dvakrát N operací, což nám dá časovou složitost $\mathcal{O}(N)$, se kterou jsme již spokojeni. Podle toho, jestli body dostaneme na vstupu správně seřazené, tak ještě musíme započítat čas na jejich setřídění podle severojižní souřadnice, v tom případě by časová složitost byla $\mathcal{O}(N \log N)$.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/32-Z2-5.py`

Jirka Setnička

32-Z2-6 Černobíločervená hra

Nejprve se zamyslíme nad jednodušší verzí úlohy, ve které nebude potřeba, aby se na nalezené cestě střídaly barvy. Stačí tedy rozlišovat červené hrany (ty nesmíme používat, takže jako by neexistovaly) od všech ostatních. Tehdy můžeme cestu najít prohledáváním do šířky, které už jsme potkali v úloze 32-Z1-4. Připomeňme si, jak se to dělá.

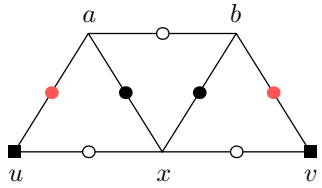
U každého vrcholu si budeme pamatovat, jestli jsme ho už objevili. Vrcholy, které jsme objevili, ale ještě jsme neprozkoumali jejich okolí, si budeme ukládat do fronty (to je seznam, kde se přidává na konec, ale odebírá ze začátku, tedy třeba pythoní `deque`).

Začneme prohledávat z počátečního vrcholu. Ten označíme za objevený a přidáme do fronty. Pak opakujeme následující: Odebereme první vrchol z fronty. Podíváme se na jeho sousedy (po hranách, jež nejsou červené). Kdykoliv není soused označený, označíme ho a přidáme do fronty. Takto pokračujeme, až dojdeme do cílového vrcholu.

Na prohledávání do šířky je šikovné, že vrcholy objevuje v pořadí od nejbližšího k nejvzdálenějšímu. Vskutku: nejdřív prohledá počáteční políčko, pak všechny jeho sousedy, pak ještě neoznačené sousedy těchto sousedů a tak dále. Takže stačí zapamatovat si u každého políčka, odkud jsme na něj přišli (tomu se říká *předchůdce* políčka). Až prohledávání skončí, podíváme se na předchůdce cílového políčka, pak na předchůdce tohoto předchůdce a tak dále, až dojdeme do počátečního políčka. Tím jsme pozpátku sestrojili nejkratší cestu mezi počátkem a koncem.

Jak dlouho to celé potrvá? Nejprve samotné prohledání do šířky: každý vrchol se do fronty dostane nejvýš jednou (jakmile je označený, podruhé ho nepřidáme). Jeho zpracování obnáší vložení do fronty, vyndání z fronty a podívání se na nejvýše 3 hrany do sousedních vrcholů. To jistě stihneme v konstantním čase na vrchol. Sestrojení cesty podle předchůdců vstoupí do každého vrcholu nejvýš jednou a opět tím stráví konstantní čas. Celkem tedy můžeme říci, že algoritmus běží v čase lineárním s celkovým počtem vrcholů.

Dobrá, ale ještě nám zbývá vyřešit původní úlohu :-) Mohli bychom si pro každý vrchol zapamatovat, jestli jsme na něj přišli černou nebo bílou hranou, a pokračovat jen hranami opačné barvy. To ale selže už na příkladu v zadání, zopakujme si ho:



Vyjdeme z vrcholu u. Pak hned objevíme x a zapamatujeme si, že jsme do něj došli bílou hranou. To nám ovšem nedovoluje pokračovat další bílou hranou do v. Ale ani obejití prostředního trojúhelníku přes a a b nám nepomůže, protože x už je označené a prohledávání do něj podruhé nevstoupí.

Napravíme to tak, že dovolíme vstoupit do jednoho vrcholu dvakrát: jednou po bílé hraně, jednou po černé. Místo vrcholů si tedy budeme značit dvojice (*vrchol*, *barva hrany*), přičemž počáteční vrchol navštívíme oběma barvami hran (to odpovídá tomu, že počáteční barvu můžeme volit libovolně).

Tím algoritmus začne fungovat a časová složitost se nepokazí: do každého vrcholu vstoupíme nejvýše dvakrát, takže pořád jedním vrcholem strávíme konstantní čas.

Martin „Medvěd“ Mareš

Výsledková listina druhé série začátečnické kategorie 32. ročníku KSP

	řešitel	*škola*	*ročník*	*sérií*	*Z2-1*	*Z2-2*	*Z2-3*	*Z2-4*	*Z2-5*	*Z2-6*	*série*	*celkem*
0.					8	10	10	12	12	14	66,0	132,0
1.	Eliška Macáková	CENADA BA	0	0	8	10	10	12	12	14	66,0	133,0
2.	Robert Gemrot	GKomHavíř	3	0	8	10	10	12	12	12	64,0	130,0
3.	Jakub Ondroušek	GTomkovaOL	0	0	8	10	10	12	11	9	60,0	126,0
4.	Kristýna Petrlíková	VOŠJičín	2	0	8	10	10	12	12	5	57,0	121,0
5.	Michal Bravanský	GBílovec	2	0	8	3	6	12	11	14	54,0	118,0
6.	Vladimír Chudý	G Chrudim	3	0	8	4	10	12	12	7	53,0	117,0
7.	Ondřej Skácel	GTomkovaOL	1	0	8	10	10	12	5	10	55,0	114,0
8.	Kateřina Vokálová	G Kolín	4	0	8	6,7	8	4	11,5	5	43,2	109,2
9.	Patrik Herman	GTomkovaOL	1	0	8	3	6	0	12	6	35,0	90,0
10.–11.	Martina Daňková	GBystrc	3	0	8	10	10	12			40,0	80,0
	Albert Kučera	GNadŠtolPH	3	0	8	10	10	12			40,0	80,0
12.	Matej Štencel	GPošKošice	3	0	8	10	10	9			37,0	77,0
13.	Jan Kotovský	GPísnickáPH	1	0	8		2	0			10,0	76,0
14.	Darian Poljak	GJŠkodyPŘ	3	0	6						6,0	72,0
15.	Sebastián Svoboda	MendelGOP	3	0	8	0	2				10,0	67,0
16.	Martin Koňařík	GJŠkodyPŘ	3	0	6						6,0	65,0
17.	Vít Ulehla	GJŠkodyPŘ	3	0							0,0	61,0
18.	Adam Húšťava	EupSchoolLux	2	0	8	0		2	2	5	17,0	57,0
19.–20.	Petr Filip	GLovosice	1	0							0,0	54,0
	Thomas Riedle	BRG APP	1	0	6				5		11,0	54,0
21.	Tomáš Chabada	SPŠEMasLI	4	0							0,0	52,0
22.	Ondřej Chlubna	GOrlová	3	0	8	2	0				10,0	50,0
23.	Jiří Bartošík	SUHr	3	0	8	1		0			9,0	49,0
24.–25.	Janek Hlavatý	GJirsíkaČB	1	0	8						8,0	48,0
	Vojtěch Žák	GŠpitálsPH	4	0	8						8,0	48,0
26.	Jiří Bleha	SPSEPard	3	0	4		0				4,0	44,0
27.	Robert Jaworski	GÚstavníPH	2	0							0,0	40,0
28.	Štěpán Řihák	G UherBrod	4	0	8						8,0	36,0
29.–30.	Jakub Nevařil	G UherBrod	2	0	3						3,0	32,0
	Daniel Šoltýs	GTřeKošice	2	0	4	0					4,0	32,0
31.	Veronika Jůzková	MensaG	2	0			0				0,0	31,0
32.–33.	Klára Hloušková	G Kolín	4	0						2	2,0	30,0
	Vojtěch Káně	G Brandýs	4	0							0,0	30,0
34.–40.	Martin Boček	MendelGOP	1	0							0,0	28,0
	Jan Hartman	GChodoviPH	4	0							0,0	28,0
	Vojtěch Máčala	G UherBrod	4	0							0,0	28,0
	Alexander Mateides	GJirsíkaČB	1	0			0				0,0	28,0
	Jan Najman	SPSEPard	3	0							0,0	28,0
	Jan Picek	GJírovcČB	3	0							0,0	28,0
	Ondřej Polanecký	SPŠPísek	3	0							0,0	28,0
41.	Jan Hlaváč	GNAlejíPH	4	0							0,0	24,0
42.	Martin Bencko	GOhradníPH	3	0							0,0	23,0
43.	Adam Krška	GMikulov	2	0							0,0	22,0
44.	Martin Klimeš	GZábřeh	4	0	2,7						2,7	20,7
45.–46.	Kryštof Maxera	GJírovcČB	−1	0							0,0	20,0
	Patrik Baláš	SPSEPard	2	0							0,0	20,0
47.	Erik Sabol	GČeskoliPH	0	0	6						6,0	19,0
48.–49.	Šimon Andrš	GKepleraPH	1	0							0,0	18,0
	Kristýna Umlaufová	SPŠOstrov	3	0							0,0	18,0
50.	David Maňásek	SUHr	3	0							0,0	17,0
51.	Stanislav Müller	GČajOlom	1	0							0,0	13,0
52.	Dominik Farhan	GMikulášPL	3	0	8			4			12,0	12,0
53.–55.	Matěj Frič	GMatOS	4	0							0,0	10,0
	Filip Chytil	SPŠ Přerov	3	0							0,0	10,0
	Antonín Musil	PORGPha	3	0							0,0	10,0

	řešitel	*škola*	*ročník*	*sérií*	*Z2-1*	*Z2-2*	*Z2-3*	*Z2-4*	*Z2-5*	*Z2-6*	*série*	*celkem*
56.–62.	Alexandr Čelakovský	G UherBrod	4	0							0,0	8,0
	Adam Ďubašík	G UherBrod	4	0							0,0	8,0
	Jiří Gallo	SPŠERožnov	3	0							0,0	8,0
	Petr Hýbl	G UherBrod	4	0							0,0	8,0
	Tadeáš Kozub	G UherBrod	4	0							0,0	8,0
	Arnošt Polák	PORG Krč	2	0							0,0	8,0
	Michal Zacek	MensaG	3	0							0,0	8,0
63.	Bohdan Kopčák	GNAlejíPH	4	0	6						6,0	6,0
64.–67.	Marek Chwistek	MendelGOP	1	0							0,0	5,0
	Jan Machourek	GBBr	−1	0							0,0	5,0
	Kristýna Prokopová	GJosBožČT	4	0	0						0,0	5,0
	Tomáš Vesecký	SSŠVTPraha	3	0							0,0	5,0
68.	Petr Hladík	GMikulášPL	2	0	4						4,0	4,0
69.	Jakub Sukdol	ZŠKubČB	−1	0							0,0	2,3
70.–71.	Michal Koudela	G UherBrod	4	0							0,0	1,0
	Petr Kroča	G UherBrod	−1	0	1						1,0	1,0