Korespondenční Seminář z Programování

30. ročník **KSP** **Červenec 2018**

Milí řešitelé a řešitelky!

Zpožděný mezistátní motorový osobní vlak KSP-30 právě přijíždí na první kolej. Ve vlaku je řazen vůz na přepravu hrochů, vzorová řešení 3. až 5. série včetně seriálu a závěrečná výsledková listina. Všem účastníkům se omlouváme za zpoždění vlaku, ale i celého letošního ročníku KSP. Doufáme, že nám zachováte přízeň i napřesrok (a že KSP bude opět fungovat obvyklým způsobem). Zatím přejeme krásné prázdniny a těšíme se na ty z vás, kdož přijedou na podzimní soustředění. Vaši organizátoři.

Vzorová řešení třetí série třicátého ročníku KSP

30-3-1 Vlnění

Pokud nevíte, jak začít, tak je vždy dobré si vzít jednoduchý případ a ten si vyřešit „ručně“. Pomocí papíru a tužky šlo dokonce vyřešit jeden vstup, takže byste ani neodcházeli s prázdnou!

Při řešení tímto způsobem jste si pravděpodobně nakreslili dostatečný počet vln, zvýraznili obdélník a spočítali černé čtverečky. Není příliš velký problém toto uvažování převést do řeči nějakého programovacího jazyka.

Jednou z jednodušších variant, jak tento přístup naprogramovat, bylo postupovat po černých vlnách. Vždy projdeme celou vlnu a připočteme jedničku za každý čtvereček, který je v obdélníku. Jen musíme někdy skončit. Přestat můžeme v okamžiku, kdy pořadové číslo vlny přesáhlo minimální souřadnici (takové vlny se nacházejí již celé „za“ obdélníkem). Takto dostaneme řešení kvadratické vzhledem k minimální souřadnici.

Toto řešení lze snadno zrychlit. Při jeho programování si totiž můžete všimnout, že při procházení každé vlny nejprve vždy zahodíte všechny čtverečky, než se dostanete do obdélníka. A jakmile se dopočítáme na konec obdélníka, zase vyhodíme všechny zbylé čtverečky. Stačí tedy při procházení každé vlny skočit na první čtvereček v obdélníku a po vystoupení z obdélníka rovnou procházení vlny ukončit. Detaily jsou spíše technického rázu, a tak odkážeme na krátký vzorový program. Takto dostaneme řešení, které je lineární vzhledem k obsahu obdélníka.

Program (C):
`http://ksp.mff.cuni.cz/viz/30-3-1-quad.c`

Rychlejší řešení

Další zrychlení je velmi podobné. Všimneme si, že při procházení každé vlny vždy přičítáme jedničku, dokud se nedostaneme na konec. Tyto jedničky bychom měli být schopní přičíst najednou, tedy spočítat, kolik černých čtverečků z dané vlny je v obdélníku.

Abychom se do řešení nezamotali, započítáme zvlášť horní a pravou stranu vlny (nezapomeneme, že čtverečky na diagonále chceme započítat jen do jedné z těchto stran).

Budeme procházet nejprve horní strany. Nejprve si zajistíme, že budeme procházet jen ty vlny, jejichž horní strana je na úrovni obdélníku (tedy ani ne pod, ani nad). Toho lze snadno dosáhnout tak, že rovnou skočíme na vlnu s pořadovým číslem odpovídajícím y-ové souřadnici levého dolního rohu a přestaneme procházet vlny, až se dostaneme na y-ovou souřadnici pravého horního rohu.

Dále si u každé strany vyjádříme počet čtverečků uvnitř obdélníka. Pro to si nejprve vyjádříme počet černých čtverečků uvnitř a před obdélníkem (to je minimum z pořadového čísla vlny a x-ové souřadnice pravého horního rohu) a počet čtverečků před obdélníkem (to je minimum z pořadového čísla vlny a x-ové souřadnice levého dolního rohu). Pak stačí tato dvě čísla odečíst.

Abychom dostali konečný výsledek, posčítáme čtverečky z každé strany. Pro pravé strany je postup obdobný jako pro horní. Takto dostaneme řešení lineární ve větším rozměru obdélníka. Pro technické detaily opět doporučujeme pročíst vzorový kód.

Program (C):
`http://ksp.mff.cuni.cz/viz/30-3-1-lin.c`

Optimální řešení

Když si budeme procházet vlny dle předchozího řešení, uvědomíme si, že z každé vlny přičítáme podle určitého vzoru: Nejprve nic, dokud vlna nedosáhne obdélníku. Poté přičteme postupně p, $p+4$, $\ldots$ $p+q$, když je „roh“ vlny v obdélníku. Dále přičítáme nějaké r za jednu stranu a poté opět nic. To bychom mohli vyjádřit přímo, ale je to velmi náchylné na chybu (zvlášť když nějaká z těchto fází pro danou vlnu chybí). Proto na to půjdeme určitým trikem.

Nejprve si představíme, že náš obdélník je ve skutečnosti čtverec a navíc má levý dolní roh o souřadnicích $(0, 0)$. V takovém čtverci jsou vždy celé vlny, dokud vlna „nepřeteče“, a pak už do čtverce nezasahují vůbec. Z první černé vlny máme 3 černé čtverce, z druhé 7, z n-té $4n - 1$. Tedy ze součtu všech vln dostaneme:

$$\sum_{i=1}^{n} 4i - 1 = 2n^2 + n.$$

V předchozím vzorci chápeme n jako počet černých vln, tedy dolní celou část ze souřadnice pravého horního rohu. Tento vzoreček je poměrně známý, ale lze k němu třeba dojít tak, že spolu sečtete první a poslední člen řady, druhý a předposlední atd.

Teď si představme, že nedostaneme čtverec, ale obdélník stále ukotvený rohem v $(0, 0)$. Budeme předpokládat, že šířka obdélníka je větší než jeho výška, ale řešení se v podstatě neliší. Souřadnice pravého horního rohu označíme (x, y). Obdélník si rozdělíme na čtverec $(0, 0)$ až (y, y) a obdélník $(x + 1, 0)$ až (x, y). Černé čtverečky ve čtverci spočítáme podle předchozího odstavce.

Pro zbylý obdélník je řešení ještě jednodušší. Žádná vlna totiž v tomto obdélníku nemá roh, skládá se tedy ze sloupečků, které jsou celé bílé nebo černé. Sloupečků je $x - y$,

polovina z nich černých (zda máme zaokrouhlovat dolů nebo nahoru, záleží na tom, jestli jsme začali černým sloupcem, tedy zda je y liché). Počet černých čtverečků pak už dostaneme jako násobek počtu černých sloupců jejich výškou (číslem y).

Nyní konečně případ, kdy nemáme levý dolní roh v $(0,0)$. Zde použijeme trik, který někteří možná znáte z tzv. prefixových součtů. Máme totiž nástroj, jak rychle spočítat počet černých čtverečků v obdélníku ukotveném v počátku. Označme (x_1, y_1) souřadnice levého dolního rohu našeho obdélníka a (x_2, y_2) pravého horního. Začneme s obsahem obdélníka $(0,0)$ až (x_2, y_2) (spočítaný podle předchozího odstavce). Jenže v tomto obdélníku jsou některé černé čtverečky navíc. Můžeme snadno odečíst černé čtverečky z obdélníku $(0,0)$ až $(x_2, y_1 - 1)$ a ještě odečteme čtverečky z obdélníku $(0,0)$ až $(x_1 - 1, y)$. Teď jsme však odečetli čtverečky z obdélníka $(0,0)$ až $(x_1 - 1, y_1 - 1)$ dvakrát. Jenže počet černých čtverečků v tomto obdélníku umíme spočítat, není tedy nic jednoduššího, než je zpátky přičíst. Tím dostaneme požadovanou odpověď.

Počet černých čtverečků ve všech čtyřech obdélnících dokážeme spočítat v konstantním čase, stejně rychle tedy běží i celé řešení. Při psaní kódu je třeba si dát pozor na to, kde chceme přičíst a nebo odečíst jedničku a také kdy správně počítat zbytky po dělení prvočíslem ze zadání. Skutečně tedy doporučujeme přečíst si vzorový kód.

Program (C):
`http://ksp.mff.cuni.cz/viz/30-3-1-const.c`

Dominik Smrž

30-3-2 Zmrazovač

Připomeňme si značení ze zadání: K je délka cesty, X je dosah zmrazovače. Jeden zmrazovač zasáhne celkem $2X + 1$ políček (své vlastní, X nalevo a X napravo). Body na cestě číslujeme 0 až $K - 1$. Dále si označíme N celkový počet nepřátel. Chceme umístit dva zmrazovače tak, aby dohromady zasáhly co nejvíce nepřátel.

Nejprve se zbavíme jednoho okrajového případu. Pokud $K \leq 4X + 2$, můžeme postavit agenty tak, aby pokryli přesně celou cestu (konkrétně na pozice X a $K - 1 - X$). Zmrazí tedy všechny nepřátele a není co řešit.

Nadále budeme předpokládat $K \geq 4X + 3$.

Též můžeme předpokládat, že v optimálním řešení se dosahy zmrazovačů nepřekrývají. Pokud bychom měli řešení, kde se překrývají, můžeme pravý z nich posouvat doprava (případně levý doleva, podle toho, na kterou stranu máme volné místo), dokud se překrývat nepřestanou. Žádné políčko tím neuvolníme z dosahu zmrazovače, takže se počet zasažených nepřátel nemůže snížit.

Zdánlivě lineární řešení

Nejprve si ukážeme řešení, které napadlo většinu řešitelů, ale není tak úplně optimální.

Pořídíme si pole P délky X, kde na pozici $P[i]$ napíšeme počet nepřátel stojících na souřadnici i (typicky to bude 0 nebo 1, ale zadání nezakazuje víc nepřátel na jednom místě).

Nyní si pro každou pozici spočítáme, kolik nepřátel by zmrazil agent stojící na daném místě, a tato čísla uložíme do pole Z. To uděláme snadno. Představíme si, že po cestě posouváme okénko délky $2X + 1$, které představuje dostřel zmrazovače. Na začátku jej postavíme středem na pozici 0 a spočítáme, kolik nepřátel okénko obsahuje.

Potom postupně posouváme okénko vždy o jednu pozici doprava. Po každém posunutí snadno v konstantním čase přepočítáme počet nepřátel uvnitř okénka: přičteme nepřátele na políčku, které se nově dostalo do okénka a odečteme nepřátele na políčku, které právě opustilo okénko. Ignorujeme části okénka, které přesahují mimo cestu, takže například prvních X posunutí nic neodečítáme. Po každém posunutí okénka si uložíme aktuální počet nepřátel pro daný středový bod i do $Z[i]$. Posunutí okénka trvá $\mathcal{O}(1)$, tedy celé pole naplníme v čase $\mathcal{O}(K)$.

Nyní by se nám líbilo postupně vyzkoušet všechna možná umístění levého zmrazovače a pro každé z nich vybrat nejlepší možné umístění pravého. Už víme, že by se dosahy neměly překrývat, tedy dáme-li levý zmrazovač na pozici i, pravý by měl být na pozici $j \geq i + 2X + 1$. Ale zároveň chceme z povolených pozic vybrat takovou, kde zmrazí nejvíce nepřátel. Tedy vybereme takové j z rozsahu $i + 2X + 1$ až $K - 1$, pro které je $Z[j]$ maximální.

Tento výběr bychom zvládli v čase $\mathcal{O}(K)$, ale to je zbytečně pomalé. Namísto toho si nejprve spočítáme *sufixová maxima* pole Z. Ta fungují podobně jako prefixové součty, jen se počítají odzadu a s maximem místo součtu. Přesněji pořídíme si pole M a do $M[i]$ uložíme pozici maxima na intervalu $Z[i]$ až $Z[K - 1]$. To zvládneme v čase $\mathcal{O}(K)$ jedním průchodem pole Z odzadu, při kterém si udržujeme průběžné maximum.

Nyní už pro danou pozici levého agenta i snadno v konstantním čase určíme pozici pravého – je to přesně $M[i + 2X + 1]$. Teď stačí vyzkoušet všechny možné pozice levého agenta, vybrat tu s nejlepším celkovým počtem zmražených $(Z[i] + M[i + 2X + 1])$ a v čase $\mathcal{O}(K)$ máme hotovo.

Hurá, lineární řešení!

Lineární?

Ehm... Kdykoli má úloha více parametrů, je třeba zamyslet se nad tím, lineární vůči čemu. Informatici obecně nemají rádi úlohy, jejichž složitost závisí na hodnotách čísel na vstupu a ne jen na jejich počtu.

Znamená to totiž, že existují maličké vstupy, na kterých program poběží v zásadě libovolně dlouho. Uvažte třeba vstup s pouhými dvěma nepřáteli, kteří stojí na pozicích 0 a 10^{18}. Takovýto vstupní soubor bude mít pár bajtů, ale výstupu se nedočkáte (a ještě před tím vám dojde paměť, protože paměťová složitost je také $\mathcal{O}(K)$).

Navíc: představte si, že vezmete nějaký existující vstup a souřadnice všech nepřátel (spolu s X a K) vynásobíte tisícem. Tím dostanete zcela ekvivalentní zadání (jen v jiném měřítku), ale váš program bude najednou tisíckrát pomalejší. To je obvykle špatné znamení.

Dá se na to dívat i formálně. Informatici, kteří se složitostí zabírají vážněji, ji obvykle měří v závislosti na celkové velikosti vstupu (v bitech). Zápis čísla K ve dvojkové soustavě je dlouhý $\log_2 K$ bitů. Ale my na vstupu délky $\log_2 K$ strávíme čas

$$\Theta(K) = \Theta(2^{\log_2 K}) = \Theta(2^{\text{velikost vstupu}}).$$

Tedy toto řešení má vůči velikosti vstupu exponenciální časovou složitost.[1]

[1] A kdo byl na letošním jarním soustředění, ví, jak to může dopadnout, když člověk píše exponenciální řešení ...

Skutečně lineární řešení

Rádi bychom našli řešení pracující v čase $\mathcal{O}(N)$. Bude vycházet ze stejných principů jako předchozí řešení, ale už si nemůžeme dovolit používat pole indexované souřadnicemi nepřátel (to by se nám ani nemuselo vejít do paměti). Místo toho budeme pracovat nad setříděným seznamem S souřadnic nepřátel (zadání nám slibuje, že už jej setříděný dostaneme), který je dlouhý $\mathcal{O}(N)$.

Tady nám pomůže ještě jeden předpoklad: totiž, že v optimálním řešení stojí na levém okraji dosahu každého zmrazovače nepřítel. Pokud ne, můžeme zmrazovač posouvat doprava, dokud se tak nestane, což nesníží počet zasažených nepřátel. A stále přitom můžeme zachovat podmínku, že se dosahy nepřekrývají.

Odteď nebudeme popisovat umístění zmrazovače jeho pozicí, ale tím, kolikátý nepřítel stojí na levém okraji zasažené oblasti.

Analogicky jako předtím si budeme chtít spočítat pole $Z[i]$ udávající, kolik nepřátel zasáhne zmrazovač, pokud na levém okraji jeho dosahu stojí i-tý nepřítel.

Použijeme opět posuvné okénko. Na začátku bude levým okrajem na prvním nepříteli a přičítáme do něj další nepřátele, dokud se do okénka vejdou (jejich vzdálenost od prvního je nejvýše $2X$). Tím dostaneme hodnotu $Z[0]$.

Nyní vždy okénko posuneme o jednoho nepřítele dál, toho odečteme a přičteme všechny, kteří se do okénka nově vejdou (průběžně si udržujeme pozici levého a pravého okraje okénka).

Jedno posunutí levého okraje může způsobit více posunutí pravého, ale protože se posouvá jen doprava, za celou dobu algoritmu se posune nejvýš N-krát. Celé pole Z tedy zvládneme naplnit v čase $\mathcal{O}(N)$.

Nyní bychom opět chtěli pro dané umístění levého zmrazovače vybrat nejlepší umístění pravého. Opět to bude takové j, aby $Z[j]$ bylo maximální a dosahy se nepřekrývaly (tato podmínka je důležitá, protože kdyby se překrývaly, započítali bychom nepřátele v jejich průniku dvakrát).

Opět bychom si rádi předpočítali něco jako sufixová maxima. V tomto případě bychom chtěli, aby $j = M[i]$ bylo číslo nepřítele takového, že jeho pozice $S[j]$ je větší nebo rovna $S[i] + 2X + 1$ a zároveň počet zmražených $Z[j]$ je maximální.

To uděláme zase tak, že projdeme seznam nepřátel zprava doleva. Ale tentokrát si pořídíme do tohoto seznamu dvě ukazovátka (levé i a pravé j). Na začátku dáme pravé ukazovátko na konec a levé posouváme od konce doleva tak dlouho, než je vzdálené alespoň $2X + 1$ (tedy $S[j] - S[i] \geq 2X + 1$). Průběžně si udržujeme pozici s maximálním $Z[j]$, přes kterou přišlo pravé ukazovátko ($j_{\max}$).

V každém kroku zapíšeme aktuální maximum na pozici levého ukazovátka ($M[i] \leftarrow j_{\max}$). Poté posuneme levé ukazovátko o jednu pozici doleva. Následně posouváme pravé ukazovátko doleva, dokud můžeme (abychom neporušili podmínku na vzdálenost alespoň $2X + 1$).

Takto v čase $\mathcal{O}(N)$ naplníme pole M požadovanými maximy.

Nyní stačí vybrat nejlepší pozici levého agenta, tedy i, pro které je celkový počet zmražených $Z[i] + M[i]$ maximální.

Hotovo, vystačíme si s časem i pamětí $\mathcal{O}(N)$.

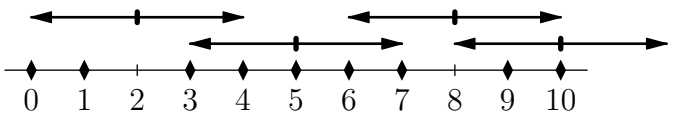
Zde bychom se ještě měli přiznat k malé zradě. V zadání jsme schválně zdůraznili, že souřadnice jsou celočíselné, aby vás napadlo i horší řešení v $\mathcal{O}(K)$. Ale doufali jsme, že jej nakonec zavrhnete ve prospěch tohoto lepšího. Kdyby souřadnice nebyly celočíselné, je řešení v $\mathcal{O}(N)$ v zásadě jediné možné a člověk nemusí rozmýšlet, které vybrat.

Na druhou stranu jsme se vám snažili i trochu pomoci, konkrétně tím, že jsme slíbili předem setříděné souřadnice nepřátel. To proto, že kdyby se musely třídit, mohl by někdo nabýt mylného dojmu, že složitost $\mathcal{O}(K)$ je lepší než $\mathcal{O}(N \log N)$. Není.

Nebuďte hladoví

Někteří řešitelé zkoušeli štěstí s hladovým algoritmem: nejprve umístím prvního agenta tak, aby zmrazil co nejvíc nepřátel. Tyto nepřátele smažu a pak umístím druhého tak, aby zmrazil co nejvíc ze zbylých. Toto řešení nefunguje.

Uvažte následující vstup s $X = 2$:



Hladové řešení umístí prvního agenta na pozici 5, kde zmrazí 5 nepřátel. Ale druhý agent pak už může zmrazit maximálně 2 nepřátele, tedy celkem 7. Optimální řešení umístí zmrazovače na pozice 2 a 8, kde zmrazí každý 4 nepřátele, tedy celkem 8.

Filip Štědronský

30-3-3 Teleportér

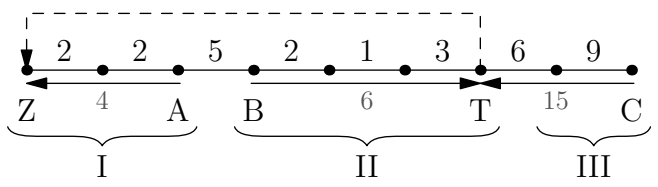
Nejkratší cestě z nějaké základny X do hlavního sídla budeme říkat *úniková cesta* z X a její délce *úniková vzdálenost* z X (značíme $U(X)$). Dále si označme M délku nejdelší únikové cesty. Hledáme umístění teleportéru takové, aby M bylo co nejmenší.

Představme si cestu spojující základy jako vodorovnou, kde hlavní základna je úplně vlevo. Určitě se nikdy nevyplatí projít teleportérem ve směru zleva doprava, tím bychom se vzdálili od cíle a museli se vracet pěšky.

Dále si uvědomíme, že levý konec teleportéru můžeme vždy umístit do hlavní základny. Kdyby byl někde jinde, jeho přesunutím do hlavní základny se únikové cesty používající teleportér zkrátí a cesty nepoužívající teleportér se nezmění (případně také zkrátí, pokud se nově vyplatí teleportér použít). Určitě se ale odnikud úniková cesta neprodlouží.

Zbývá určit, kam umístit pravý konec teleportéru. To uděláme tak, že postupně vyzkoušíme všechna možná umístění, pro každé spočteme M a vybereme nejlepší řešení.

Představme si, že už jsme jej někam umístili (označme si toto místo T). Pak lze základny pomyslně rozdělit na tři úseky:



Úsek I tvoří základny, z nichž je nejkratší cesta do hlavního sídla pěšky, tedy takové, které jsou blíže k Z než k T. To jsou přesně základny ležící v levé polovině spojnice ZT ($|ZX| < |ZT|/2$).

Můžeme si všimnout, že ze všech základen v úseku I má nejdelší únikovou cestu poslední základna, kterou si označíme A ($U(A) = |ZA|$). Ostatní základny z tohoto úseku můžeme tedy při výpočtu maxima ignorovat.

Úsek II naopak tvoří základny, ze kterých už se vyplatí jít doprava k teleportéru. Analogicky nejdelší úniková cesta v tomto úseku je z jeho první základny B ($U(B) = |BT|$).

Úsek III jsou základny napravo od T, ze kterých se vždy vyplatí jít přes teleport. Nejdelší úniková cesta v tomto úseku je z nejpravější základny C ($U(C) = |TC|$).

Když víme, že nejdelší úniková cesta je ze základny A, B nebo C, spočteme její délku snadno:

$$M = \max(U(A), U(B), U(C)) = \max(|ZA|, |BT|, |TC|),$$

v našem případě $\max(4, 6, 15) = 15$.

Abychom mohli M spočítat, potřebujeme:

- umět rychle určit vzdálenost dvou základen,
- vědět, kde je hranice mezi úseky I a II (základny A a B).

První úkol je jednoduchý: nad polem délek tunelů si spočítáme prefixové součty[2] – tedy vlastně pro každou základnu spočítáme její tunelovou vzdálenost $D(X)$ od hlavního sídla. Potom vzdálenost $|XY|$ snadno spočítáme v konstantním čase jako $D(Y) - D(X)$.

Základny A a B můžeme najít binárním vyhledáváním v poli prefixových součtů. Pokud vyhledáme hodnotu $|ZT|/2$, trefíme se přesně mezi A a B.

Tím bychom dostali řešení v čase $\mathcal{O}(N \log N)$, kde N je počet základen. Musíme vyzkoušet N možných umístění teleportéru, pro každé binárně vyhledat hranici mezi úseky a pak v konstantním čase spočítat M, přičemž si průběžně udržujeme nejlepší zatím nalezený výsledek.

Lineární řešení

Existuje ale i řešení v lineárním čase. Využívá následující pozorování: když posuneme teleportér doprava, mohou se A a B posunout také jen doprava (nebo zůstat na místě).

Na začátku umístíme teleportér do základny hned vedle hlavního sídla. Pak $A = Z$ a $B = T = Z + 1$. Budeme postupně posouvat T doprava a průběžně si udržovat čísla základen A a B. Po posunutí T musíme opravit pozici A a B. To uděláme jednoduše: dokud platí $|ZB| < |ZT|/2$ (B leží v levé polovině spojnice ZT, tedy v úseku I), posuneme A i B o jednu základnu doprava.

Po jednom posunutí T se můžou A a B posunout i víckrát. Protože se ale posouvají jen doprava, za celý běh algoritmu dohromady se posunou maximálně N-krát, tedy jejich posouváním strávíme celkem čas $\mathcal{O}(N)$. Všechno ostatní (výpočet M a udržování průběžného minima) zvládneme v konstantním čase. Celkem si tedy vystačíme s časem $\mathcal{O}(N)$.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/30-3-3.py`

Filip Štědronský

30-3-4 Hmota a antihmota

Naším úkolem bylo oddělit hmotu a antihmotu aneb zjistit, jestli lze v rovině oddělit dvě množiny bodů přímkou tak, aby všechny body jednoho typu byly na jedné straně přímky a všechny body druhého typu na straně druhé.

Operovat se spoustou bodů přímo by ale bylo zbytečně složité, a tak si situaci trochu zjednodušíme použitím technik z odkazované kuchařky o geometrii. Uvědomíme si, že když nějaké dvě množiny bodů umíme oddělit pomocí přímky, umíme oddělit i jejich *konvexní obaly* (konvexní mnohoúhelníky obsahující všechny zadané body).

Spočítáme si konvexní obal bodů hmoty i antihmoty (což podle algoritmu z kuchařky umíme v čase $\mathcal{O}(N \log N)$ – protože si body musíme setřídit podle nějaké souřadnice) a pak se zamyslíme, že mohla nastat jedna ze tří situací:

- Konvexní obaly se protínají – v takovém případě určitě body hmoty a antihmoty oddělit nelze.
- Konvexní obaly se neprotínají, ale hranice jednoho leží uvnitř druhého – tady také zjevně nelze body hmoty a antihmoty oddělit přímkou.
- Konvexní obaly se neprotínají a neleží v sobě – zde oddělovací přímka existuje.

Proč v posledním případě oddělovací přímka vždy existuje? Vezměme si jeden bod na obvodu konvexního obalu hmoty a druhý na obvodu konvexního obalu antihmoty takové, že jejich vzdálenost je nejmenší možná. Rozmyslete si, že to bez újmy na obecnosti je buď dvojice vrchol-vrchol, nebo vrchol-hrana.

Sestrojme úsečku spojující tyto dva nejbližší body a pak si vezměme osu této úsečky (nazvěme ji q) a rozmysleme si, že q určitě neprotíná ani jeden z konvexních obalů. Rozmyslíme si to pro obal hmoty, pro obal antihmoty to bude to samé: Pokud na straně hmoty je nejbližší bod vrchol, tak hrany z něj vycházející se od q určitě vzdalují a zároveň vymezují výsek roviny, ve kterém se celý konvexní obrazec nachází, takže k protnutí obrazce s q již určitě nedojde. A pokud je nejbližší bod na nějaké hraně, tak si rozmysleme, že úsečka spojující tento bod s nejbližším bodem druhého konvexního obalu bude kolmá na tuto hranu a q tak bude s touto hranou rovnoběžná. A protože hrana vymezuje polorovinu, ve které se celý konvexní obrazec musí nacházet, tak ani v tomto případě k protnutí obrazce s q určitě nedojde.

Dokázali jsme tak, že když se konvexní obrazce neprotínají a neleží v sobě, tak určitě existuje alespoň jedna oddělovací přímka. Než se zamyslíme, jak nějakou oddělovací přímku najít co nejrychleji, pojďme si spočítat, jak rychle umíme vyřešit samotnou otázku její existence.

Jak jsme si řekli výše, spočítání konvexních obalů nás bude stát čas $\mathcal{O}(N \log N)$. Jak rychle umíme udělat test protnutí? Určitě bychom mohli zkusit každou hranu jednoho obrazce s každou hranou druhého, ale to by zabralo čas $\mathcal{O}(N^2)$, což nechceme. Namísto toho se opět můžeme podívat do kuchařky a zjistit, že zde máme algoritmus hledající průsečíky úseček v rovině v čase $\mathcal{O}((N + P) \log N)$ (kde P je počet průsečíků).

Nás budou zajímat průsečíky hran jednoho obalu s hranami druhého obalu a už jediný takový průsečík postačí ke zjištění, že se konvexní obaly protínají. Stačí tedy spustit algoritmus a zastavit ho okamžitě, jak na něco takového přijde. Můžeme tedy P shora odhadnout pomocí N a čas nám tak vyjde $\mathcal{O}(N \log N)$ na tuto kontrolu.

Nakonec musíme ověřit, jestli se jeden konvexní obal nenachází celý uvnitř druhého. Vezmeme si tedy z každého

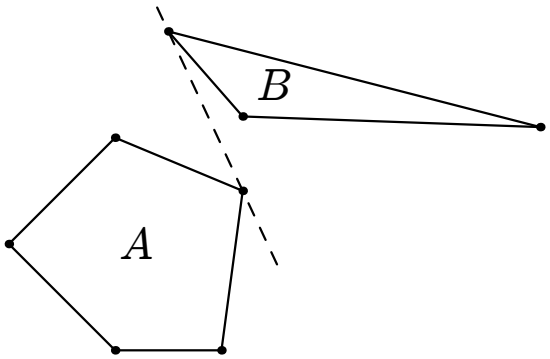
[2] `http://ksp.mff.cuni.cz/viz/kucharky/zakladni`

konvexního obalu jeden bod a otestujeme, jestli se nenachází uvnitř konvexního obalu druhého obrazce, tento test zvládneme v $\mathcal{O}(N)$.

Pokud se nám konvexní obaly neprotínají a pokud se ani jeden z nich nenachází uvnitř druhého, tak můžeme oznámit, že oddělující přímka existuje. Vyřešili jsme tak lehčí verzi úlohy jenom s jedním obsáhlejším důkazem a s algoritmy z geometrické kuchařky v čase $\mathcal{O}(N \log N)$.

Nalezení oddělovací přímky

Na úvod si povolme, že oddělovací přímka nám bude stačit i taková, která se některých bodů dotýká (pokud by nám to nestačilo, tak ji vždy můžeme posunout o dostatečně malé ε tak, aby se bodů nedotýkala). Příkladem může být čárkovaná přímka na obrázku.



Teď se zamysleme, že když si vezmeme libovolnou oddělovací přímku dvou konvexních mnohoúhelníků, můžeme tuto oddělovací přímku pootočit (a posunout) tak, aby se každého z obrazců dotýkala v nějakém vrcholu. Náš cíl bude hledat právě takovéto oddělovací přímky, tedy přímky, které jsou prodloužením nějaké úsečky mezi nějakým vrcholem prvního a nějakým vrcholem druhého obrazce.

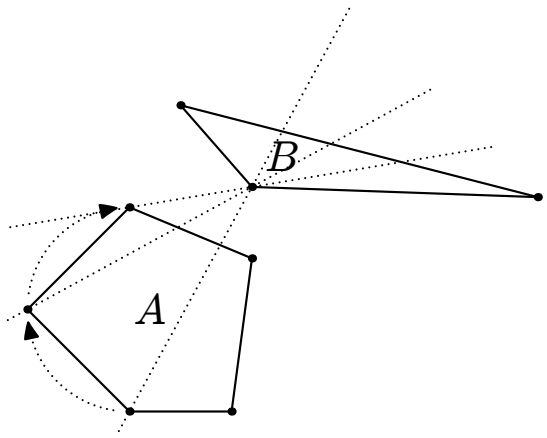
Pro vybranou dvojici vrcholů umíme rychle ověřit, jestli je jimi určená přímka oddělovací – pro každý z vrcholů si ověříme, jestli je v něm přímka tečnou (jen se obrazce v tomto bodě dotkne, ale nevstoupí do něj), nebo ne. Stačí nám ověřit, že je přímka oproti oběma hranám vycházejícím z tohoto vrcholu umístěná na stejné straně (pokud je, tak již nezasáhne ani do zbytku obrazce). Druhá podmínka, kterou oddělovací přímka musí splnit, je, že jeden konvexní mnohoúhelník se od ní nachází na jednu stranu a druhý na opačnou stranu. To umíme lehce ověřit tím, že si vybereme jeden bod z každého mnohoúhelníku a zjistíme, jestli leží na opačných stranách. Otestování tak umíme pro dvojici vrcholů vyhodnotit v konstantním čase.

Možných dvojic vrcholů je $\mathcal{O}(N^2)$, takže při vyzkoušení všech možností bychom úlohu zvládli v čase $\mathcal{O}(N^2)$, ale to nám stačit nebude.

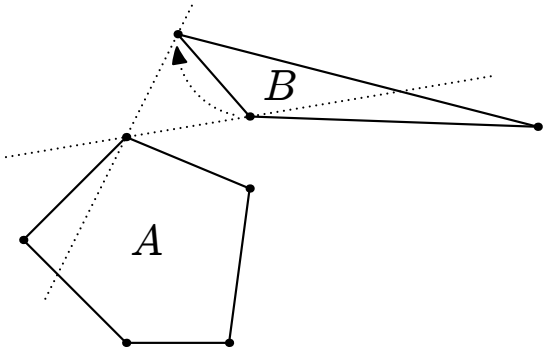
Pojďme na to tedy jinak – na počátku si zvolíme libovolné dva vrcholy, každý z jednoho mnohoúhelníku, a natáhneme přímku mezi nimi. Když testem výše zjistíme, že není oddělovací, tak se po nějakém z mnohoúhelníků posuneme a zkusíme to znovu. Toliko postup v kostce, nyní podrobněji.

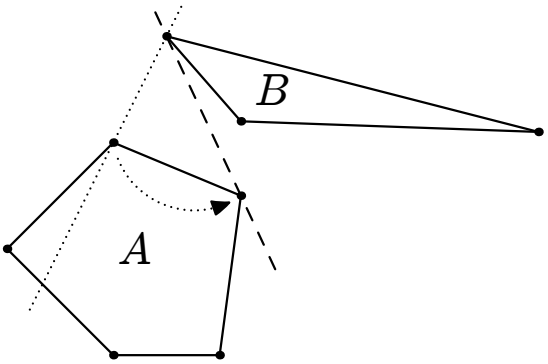
Pro vrchol a na mnohoúhelníku A a vrchol b na mnohoúhelníku B provedeme první pravidlo, které má splněné podmínky, a opakujeme:

1. Pokud přímka určená a-b splňuje podmínky výše (neprotíná A ani B a A leží na opačné straně, než B), tak a-b vydáme jako řešení a skončíme.
2. Pokud přímka určená a-b protíná A směrem od a k b, tak je a na špatné straně $A \rightarrow$ posuneme a po směru hodinových ručiček na další vrchol z A.



3. To samé pro b a B.
4. Pokud přímka určená a-b protíná A směrem od a dál $\rightarrow$ posuneme a po směru ručiček na další vrchol z A.
5. To samé pro b a B.





Pravidla 2 a 3 nám zaručí, že body na sebe „vidí“ (pokud na sebe přestanou vidět, tak „oběhnou“ svůj mnohoúhelník na druhou stranu). A pravidla 4 a 5 nám posunou přímku tak, aby byla pro každý z mnohoúhelníků tečnou.

Pokud oddělovací přímka existuje (což po kontrole z první části úlohy víme), tak náš algoritmus projde všechny možné kandidáty na ni. Dál po obvodu se posouváme jenom tehdy, pokud již víme, že na současné pozici oddělovací přímku nezkonstruujeme. Přitom maximálně jednou obkroužíme každý z obrazců, tato část je tedy lineární.

Společně s první částí umíme celou úlohu vyřešit v čase $\mathcal{O}(N \log N)$.

Jirka Setnička

30-3-5 Rozbití skupin

Úloha je podobná úloze 30-Z3-4 ze začátečnické kategorie, tak se inspirujeme jejím řešením. Dostali jsme řetězec $x_1, \ldots, x_N$, v němž jsou na některých pozicích „písmenka“ z P-prvkové abecedy a všude jinde mezery. Na místa mezer chceme vyplnit další písmenka tak, aby nikde nebylo více než M stejných písmenek vedle sebe.

Nejprve si rozmyslíme, jak bychom úlohu řešili, kdyby všude byly mezery. Budeme postupně počítat pro čím dál delší prefixy $x_1, \ldots, x_i$ čísla A_{ip}. Ta nám budou říkat, kolika způsoby jde prefix vyplnit tak, aby končil písmenem p.

Odložme na chvíli, co dělat na úplném začátku. Představujme si, že už jsme v řetězci pokročili „dostatečně daleko“ a chceme spočítat nějaké A_{ip}. Jak k tomu využít už známá $A_{i'p'}$ pro $i' < i$?

Víme, že vyplněný prefix má končit písmenem p. To je součástí nějakého delšího souvislého p-ček, který má délku $k \leq M$. Před tímto úsekem už musí ležet nějaké jiné písmeno p' (nebo začátek řetězce, ale to se nestane, protože jsme dostatečně daleko). Pro každé $p' \neq p$ tedy máme $A_{i-k,p'}$ možností, jak vyplnit předchozí část prefixu, a za každou z nich můžeme připojit koncový úsek z k p-ček.

Sečteme tedy tyto možnosti přes všechny možné volby k a p' a dostaneme:

$$A_{ip} = \sum_{k=1}^{M} \sum_{p' \neq p} A_{i-k,p'}.$$

Nyní vrátíme do hry možnost, že některá písmena jsou už předem daná. Tím pádem musíme při zvyšování k kontrolovat, zda x_{i-k+1} není předepsáno jako písmeno různé od p, a tehdy se zastavit. Může se samozřejmě stát, že se zarazíme hned o x_i, takže celé A_{ip} vyjde nulové.

Zbývá dořešit případ, kdy nejsme dostatečně daleko. Tehdy musíme počítat s tím, že $i - k$ může prolézt před začátek řetězce. Pomůžeme si snadno: rozšíříme abecedu o nové písmeno Q a předvyplníme ho na pozici 0. Nové písmeno nebudeme vyplňovat nikam jinam. Nyní zjevně platí $A_{0\mathrm{Q}} = 1$ a $A_{0p} = 0$ pro $p \neq \mathrm{Q}$. Výpočet všech ostatních A_{ip} se pak vždy zarazí o začátek řetězce a správně započítáme jednu možnost, protože začít se dá jedním způsobem.

K vyřešení úlohy nám tedy stačí spočítat všechna A_{ip} a pak sečíst A_{np} přes všechna p. Jak dlouho to bude trvat? Všech A_{ip} je $\mathcal{O}(NP)$, při výpočtu každého z nich sčítáme $\mathcal{O}(MP)$ čísel. Časová složitost tedy činí $\mathcal{O}(NP^2M)$.

Zrychlujeme...

Několika jednoduchými úpravami můžeme algoritmus ještě zrychlit. Především se zbavíme druhé sumy ve výpočtu A_{ip}. K tomu si předpočítáme čísla $T_i = A_{i1} + A_{i2} + \ldots + A_{ip}$ a všimneme si, že druhá suma je rovna $T_{i-k} - A_{i-k,p}$. (Druhá suma vždy z P členů součtu T_i jeden vynechává, tak ho prostě od T_i odečteme.)

Tím pádem spočítat jedno A_{ip} trvá pouze $\mathcal{O}(M)$, takže jedno i zpracujeme v čase $\mathcal{O}(PM)$. Do toho se jistě vejde čas $\mathcal{O}(P)$ potřebný na spočítání T_i. Vida, celý algoritmus jsme zrychlili na $\mathcal{O}(NPM)$.

Nyní vytáhneme z rukávu další trik. Představte si, že bychom se dozvěděli, kam až se zvýší k v první sumě, než se zarazíme o předepsaný znak různý od p. Označme tuto hodnotu K. Počítáme tedy

$$A_{ip} = \sum_{k=1}^{K} (T_{i-k} - A_{i-k,p}) = \sum_{i'=i-K}^{i-1} (T_{i'} - A_{i',p}) =$$
$$= \left(\sum_{i'=i-K}^{i-1} T_{i'} \right) - \left(\sum_{i'=i-K}^{i-1} A_{i',p} \right).$$

To jsou nějaké dvě sumy souvislých úseků posloupností, které zvládneme spočítat v konstantním čase, pokud si pro tyto posloupnosti budeme udržovat prefixové součty.

Aby tento trik fungoval, musíme ovšem znát K. Ukážeme, že si můžeme dovolit přepočítávat při každém prodloužení prefixu *zarážky* – čísla Z_p, která nám budou říkat, na které pozici leží poslední nemezerové písmenko různé od p. Udržování je snadné: kdykoliv prodloužíme prefix o nějaké písmenko p, přenastavíme všechny zarážky $Z_{p'}$ pro $p' \neq p$ na aktuální pozici i. Pokud prodloužíme o mezeru, zarážky zůstanou stejné.

Rozmysleme si, kolik práce vykonáváme při každém z N prodloužení prefixu. Nejprve přepočítáme zarážky, to trvá $\mathcal{O}(P)$. Pak pro každé p počítáme A_{ip} v konstantním čase, celkem tedy zase $\mathcal{O}(P)$. Každé A_{ip} také připočítáme k T_i, což nám čas nezhorší. Nakonec aktualizujeme všechny prefixové součty; ty udržujeme pro $P+1$ různých posloupnost (jedna z nich je T), a to opět stihneme v $\mathcal{O}(P)$.

Tím jsme získali algoritmus s časovou složitostí $\mathcal{O}(NP)$.

Poznámka o samých mezerách

Dovolíme si krátkou poznámku na závěr: Jak by se úloha chovala, kdyby v zadaném řetězci byly jenom mezery? Tehdy by hodnota A_{ip} evidentně nezávisela na p, takže by stačilo počítat A_{i1}. Druha suma by tedy sčítala $P-1$ stejných členů a dostali bychom:

$$A_{i1} = \sum_{k=1}^{M} (P-1) \cdot A_{i-k,1} = (P-1) \cdot \sum_{k=1}^{M} A_{i-k,1}.$$

To je nějaká lineární rekurence, na kterou by se dal pro konkrétní P a M vymyslet explicitní vzoreček, do nějž bychom rovnou dosadili i a vypadlo by řešení. Například pro $P = M = 2$ vyjdou známá Fibonacciho čísla.

Martin „Medvěd“ Mareš

30-3-6 Střežení oblasti

Na této úloze není nic příliš záludného – jednoduše stačí příkazy odsimulovat. Už však záleží na tom, jaké datové struktury přitom využijeme.

Hranice se v každém čase skládá z několika navzájem se nepřekrývajících *hlídaných intervalů* – souvislých úseků, jejichž celý vnitřek je hlídaný a jejichž krajní body hraničí s nehlídanými oblastmi. Takové intervaly můžeme v programu reprezentovat jako dvojice (*začátek*, *konec*). Právě na počet hlídaných intervalů po každé operaci se úloha ptá.

Pořídíme si tedy nějakou datovou strukturu, ve které si budeme udržovat všechny intervaly a po každé operaci ji odpovídajícím způsobem upravíme. Pro začátek budeme skromní a intervaly budeme v nějakém libovolném pořadí udržovat v poli.

Co udělají obě operace s naším polem?

- Hlídej (A, B): všechny intervaly překrývající se s (A, B) se slijí do jednoho intervalu. Ten bude začínat v nejlevějším z jejich začátků (popř. A, pokud je A ještě více nalevo) a končí v nejpravějším z jejich konců (popř. B).

V naší reprezentaci nám stačí pole projít, smazat právě všechny intervaly zasahující do (A, B) a pak přidat na konec nový interval (A', B') odpovídající jejich „slití“.

- Přestaň hlídat (A, B): intervaly ležící plně v (A, B) zmizí, intervaly částečně zasahující do (A, B) se z jedné strany ořízno u a intervaly zcela obsahující (A, B) se rozdělí na dva. Opět stačí pole projít a změnit pouze intervaly mající překryv s (A, B).

Rozmyslete si, že to, zda se dva intervaly překrývají, umíme ověřit v konstantním čase. Celkový počet intervalů si také jistě umíme udržovat v konstantním čase.

Toto řešení má paměťovou složitost $\mathcal{O}(N)$ a časovou složitost $\mathcal{O}(N^2)$, kde N je počet operací. Každá operace totiž vytvoří maximálně jeden interval, takže po N operacích jich můžeme mít nejvýše N. Když nebudeme operace provádět hloupě (např. nebudeme všechny intervaly zasahující do (A, B) mazat po jednom v $\mathcal{O}(N)$, ale smažeme je všechny najednou), budou obě operace trvat $\mathcal{O}(N)$.

Rozmyslíme si, že v průměru po každé operaci přidáme, změníme a odstraníme jen konstantně mnoho intervalů. Proč tomu tak je? V obou operacích odstraňujeme libovolně mnoho intervalů, ale přidáváme jich vždy jen konstantně mnoho. Abychom ale mohli mazat, musíme mít co, tedy pokud jsme provedli k mazání, muselo před nimi následovat aspoň k přidání. Víme však, že přidání je dohromady $\mathcal{O}(N)$, tedy i mazání je $\mathcal{O}(N)$ a průměrný počet mazání na jednu operaci je konstantní.

Zrychlujeme

To, že každá operace v průměru změní jen konstantně intervalů, naznačuje, že by mohlo existovat i nějaké rychlejší řešení. To naše je pomalé hlavně kvůli tomu, že prochází hromadu intervalů, které pro danou operaci nejsou nijak zajímavé.

Pojďme ho upravit. Začneme stejně, s polem uchovávajícím jednotlivé intervaly. Tentokrát v něm však intervaly budou uspořádané vzestupně podle svých začátků. Díky tomu budeme v poli umět binárním vyhledáváním najít nejlevější a nejpravější interval, které nás pro danou operaci zajímají a budeme moci měnit jen intervaly mezi nimi. Obě operace nám pak postačí upravit tak, aby uspořádání zachovávaly – stačí všechna vkládání provádět nikoliv na konec, ale na místo určené dalším binárním vyhledáváním. Odstranění jistě uspořádání neporuší a úprava intervalu jde nahradit kombinací odstranění a vkládání.

Problém s takovým řešením je, že je úplně stejně pomalé jako to předchozí. Na nalezení správného místa sice používáme binární vyhledávání, ale všechno si pokazíme vkládáním do pole, jelikož už tentokrát nevkládáme jeho konec, ale do jeho prostředka, což samozřejmě trvá $\mathcal{O}(N)$. Nabízí se použít spojové seznamy, které umí vkládat v $\mathcal{O}(1)$. V těch však zase neumíme rychle binárně vyhledávat, protože jen s ukazatelem na první prvek neumíme rychle najít tolik potřebný prostředek.

Zachrání nás vyhledávací stromy. Pokud jste o nich ještě neslyšeli, můžete si přečíst naši kuchařku,[3] ve které popisujeme, k čemu taková datová struktura je a jak může vypadat uvnitř. Pro popis řešení stačí vědět, že binární vyhledávací stromy umí udržovat uspořádanou množinu a v $\mathcal{O}(\log N)$ nad ní provádět operace „přidej prvek do množiny“, „odeber prvek z množiny“ a „vrať nejmenší prvek z množiny větší než x“.

Stačí tedy, když v předchozím řešení vyměníme pole za nějaký vyhledávací strom (ve kterém opět řadíme intervaly podle jejich začátku), a dostaneme řešení pracující v čase $\mathcal{O}(N \log N)$.

Na závěr malá poznámka: i řešení se spojovými seznamy by šlo upravit, aby v průměru běželo v $\mathcal{O}(N \log N)$, takové randomizované struktuře se říká *skip list*. Hrubá myšlenka je taková, že každé buňce spojového seznamu kromě odkazu na následníka náhodně přidáme ještě několik zkratkových odkazů, každý další odkaz ukazující zhruba dvakrát dál než předchozí. Při vkládání prvku na správné místo si pak zkratkami urychlujeme čas a v průměru nám nalezení správného místa trvá $\mathcal{O}(\log n)$. Více si o skip listech můžete přečíst například na anglické Wikipedii.[4]

Říša Hladík

30-3-7 Funkce očima assembleru

Úkol 1: Třetí největší číslo

Hledání třetího největšího čísla můžeme napsat podobně jako třídění vkládáním. V registrech `r3` až `r5` si budeme udržovat první až třetí největší číslo. Na začátku všechna tři inicializujeme na nuly. Pak z paměti načítáme prvky posloupnosti jeden po druhém a zatřiďujeme je mezi tři největší čísla. To se dá hezky popsat pomocí podmíněných instrukcí `MOV`.

Program je přímočarý, jen si musíme dávat pozor na volací konvenci: počet čísel dostaneme v `r0`, adresu prvního čísla v `r1`, registry od `r4` výše musíme vrátit do původního stavu a z funkce se vracíme pomocí `BX lr`.

```
max3:
  PUSH  {r4-r6}
  MOV   r3, #0     @ r3 = zatím největší číslo
  MOV   r4, #0     @ r4 = druhé největší
  MOV   r5, #0     @ r5 = třetí největší
dalsi:
  LDR   r2, [r1], #4  @ r2 = aktuální číslo
  CMP   r2, r3     @ vyměníme za r3?
  MOVHS r6, r3
  MOVHS r3, r2
  MOVHS r2, r6
  CMP   r2, r4     @ vyměníme za r4?
  MOVHS r6, r4
  MOVHS r4, r2
  MOVHS r2, r6
  CMP   r2, r5     @ vyměníme za r5?
  MOVHS r5, r2
  SUBS  r0, #1     @ snížíme počítadlo
  BNE   dalsi
  MOV   r0, r5     @ vrátíme třetí největší
  POP   {r4-r6}
  BX    lr
```

Úkol 2: Nulování paměti

Jak vynulovat blok paměti co nejméně instrukcemi? K tomu se hodí instrukce `MOVHS` – ta umí zapsat do paměti hned několik registrů najednou. My si uvolníme registry `r2` až `r12`, naplníme je nulami a pokaždé zapíšeme všechny najednou.

[3] `http://ksp.mff.cuni.cz/viz/kucharky/vyhledavaci-stromy`
[4] `https://en.wikipedia.org/wiki/Skip_list`

Než se do toho pustíme, musíme ovšem zařídit, aby adresa nulované paměti byla dělitelná čtyřmi. A když už bude zbývat méně než 44 bajtů, je potřeba „včas sundat sedmimílové boty“ a zbytek vynulovat citlivěji (v našem případě po bajtech). Zbývá poslední detail: ošetřit případy, kdy je celý zadaný blok příliš krátký – tehdy rovnou přepneme na nulování po bajtech.

```
nuluj:
  MOV   r2, #0    @ r2 bude vždy 0
  CMP   r1, #48   @ příliš krátký blok
  BLO   pomalu

zarovnej:         @ adresu zarovnáme na 4 B
  ANDS  r3, r0, #3
  BEQ   rychle
  SUB   r1, #1
  STR   r2, [r0], #1
  B     zarovnej

rychle:           @ r2-r12 budou nuly
  PUSH  {r4-r12}
  MOV   r3, r2
  MOV   r4, r2
  @ ... podobně do r5 až r12
  SUB   r1, #43
rychleee:         @ nulujeme najednou 44 B
  STMIA r0!, {r2-r12}
  SUBS  r1, #44
  BHS   rychleee
  ADD   r1, #43
  POP   {r4-r12}

pomalu:           @ zbylých nejvýše 48 B
  STR   r2, [r0], #1
  SUBS  r1, #1
  BNE   pomalu
  BX    lr
```

Jak rychle nulujeme? Na jeden průchod hlavní smyčkou (od návěští `rychleee`) vynulujeme 44 bajtů a stojí nás to 3 instrukce. Nulujeme tedy $44/3 \doteq 14.67$ bajtu za instrukci. Režie zbylých dvou smyček (zarovnávací a koncové) je shora omezena nějakou konstantou, takže pro velké bloky je zanedbatelná.

Všimněte si ale, ze tří instrukcí v hlavní smyčce jenom jedna nuluje paměť, zatímco zbylé dvě se starají, aby smyčka běžela a včas se zastavila. Pouze třetinu instrukcí tedy spotřebujeme na užitečnou práci, zbytek je režie smyčky. Tento poměr můžeme zlepšit tak, že několik iterací smyčky sloučíme do jedné, třeba takto:

```
rychleeeee:
  STMIA r0!, {r2-r12}
  STMIA r0!, {r2-r12}
  STMIA r0!, {r2-r12}
  SUBS  r1, #132
  BHS   rychleeeee
```

Nyní v jedné iteraci smyčky provedeme celkem 5 instrukcí, z nichž 3 jsou výkonné a stále 2 režijní. Poměr se tedy zlepšil z 1/3 na 3/5. Za jednu iteraci nyní vynulujeme 132 bajtů, takže rychlost jsme zvýšili na $132/5 = 26.4$ bajtu na instrukci. Takto se můžeme libovolně přiblížit k teoretickému maximu 44 bajtů na instrukci, ovšem platíme za to tím, že pro malé bloky se efektivita programu snižuje.

Mimochodem, kombinování více iterací do jedné je standardní technika, kterou například překladače Céčka běžně dělají. Říká se jí *loop unrolling,* nebo česky rozbalování smyček.

Úkol 3: Součet argumentů

Sčítání všech argumentů až do prvního nulového je jednoduché, jen se musíme poprat s tím, že první 4 argumenty dostaneme v registrech, zatímco zbývající na zásobníku. Přitom předem nevíme, kolik jich celkem bude. Pomůžeme si snadno: argumentové registry `r0-r3` hned uložíme na zásobník, čímž zařídíme, že všechny argumenty budou uloženy na zásobníku pěkně za sebou, a tak je odtamtud jeden po druhém přečteme. Ve skutečnosti je lepší první argument nechat v registru `r0` a k tomuto registru pak přičítat všechny ostatní argumenty. To je současně registr, ve kterém máme vrátit výsledek.

```
soucet_arg:
  CMP   r0, #0    @ součet 0 argumentů je 0
  BEQ   hotovo    @ v r0 bude výsledek
  PUSH  {r1-r3}   @ argumenty 2 až 4 na zásobník
  MOV   r1, sp
znovu:            @ procházíme zásobník
  LDR   r2, [r1], #4
  ADD   r0, r2
  CMP   r2, #0
  BNE   znovu
  POP   {r1-r3}   @ uklidíme zásobník
hotovo:
  BX    lr
```

Úkol 4: Nebudu si číst pod lavicí...

Nechat programátora, ať za trest něco napíše stokrát, jak známo potrestá spíš jeho učitele. Stačí ve smyčce volat `printf` podle volací konvence. Jediné, na co bychom se mohli nachytat, je, že funkce mají dovoleno přepsat registry, v nichž dostaly argumenty, takže si je nesmíme zapomenout uložit.

```
  LDR  r0, =trest  @ r0 = formátovací řetězec
  MOV  r1, #1      @ r1 = počítadlo
otrocina:
  PUSH {r0,r1}
  BL   printf
  POP  {r0,r1}
  ADD  r1, #1
  CMP  r1, #100
  BLS  otrocina
  B    konec

trest:
  .ASCII "%d. Nebudu si číst pod lavicí"
  .BYTE 10, 0   @ konec řádku, konec řetězce
  .ALIGN 4

konec:
```

Úkol 5: Ohackované printf

Zlatým hřebem tohoto dílu seriálu bylo upravit `printf`, aby číslovalo řádky. Nabízí se vyměnit všechna volání `printf` za volání nějaké naší funkce. Tomu brání zdánlivá maličkost: neumíme všechna volání najít.

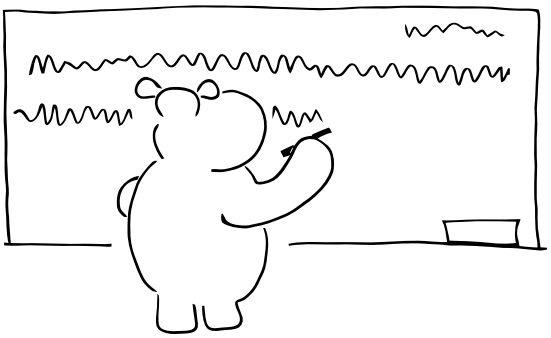
Půjdeme na to chytřeji: přepíšeme první instrukci funkce `printf` na skok do naší vlastní funkce. Naše funkce vypíše číslo řádku a následně vypíše to, kvůli čemu byla zavolaná. Na obojí se hodí zavolat původní `printf`. To zařídí funkce `orig_printf`, která nejprve provede první instrukci původního `printf` (tu, kterou jsme přepsali skokem) a pak skočí na zbytek původního `printf`.

Jediný další chyták je, že v kódu instrukce skoku je cílová adresa uložena relativně (32-bitová absolutní adresa by se do 4 bajtů instrukce nevešla). Nemůžeme tedy instrukci zkopírovat z jiného místa v programu. To by se dalo obejít zkonstruováním adresy v registru a `BX` na tento registr, ale pak bychom si nevystačili s přesunutím jediné instrukce. Radši se tedy naučíme, jak se instrukce skoku kóduje (to najdeme v popisu instrukční sady ARMu, nebo se prostě v simulátoru podíváme, jak jsou různé skoky zakódované).

Pokud na adrese x leží instrukce `B` y, uložíme do ní, že má skákat o $(y - x - 8)/4$ dopředu. Osmičku odečítáme proto, že se snaží mít rozpracovaných několik instrukcí v předstihu, takže v okamžiku vykonávání instrukce z adresy x je v registru `pc` (`r15`) už hodnota $x + 8$. Navíc adresy instrukcí jsou vždy dělitelné čtyřmi, takže nejnižší dva bity rozdílu jsou vždy nulové a není je potřeba ukládat. Kód instrukce skoku má pak ve svých horních 8 bitech uloženo `0xea` a ve zbylých 24, o kolik dopředu skáče (záporná čísla ukládá ve dvojkovém doplňku). Kód tedy v programu snadno spočítáme.

```
printf_hack:
  @ Zkopírujeme 1. instrukci printf
  LDR  r1, =printf
  LDR  r2, =orig_printf
  LDR  r0, [r1]
  STR  r0, [r2]

  @ Nahradíme ji skokem na fake_printf
  LDR  r0, =fake_printf
  SUB  r0, r1
  SUB  r0, #8  @ kompenzace posunu pc
  LSL  r0, #6  @ dělení 4 a současně nulování
  LSR  r0, #8  @ horních 8 bitů
  ORR  r0, #0xea000000  @ horních 8 bitů
  STR  r0, [r1]
  B    konec

  @ Toto je ekvivalentní s původním printf
orig_printf:
  .WORD 0     @ sem přijde 1. instrukce printf
  B printf+4  @ pokračování původního printf

  @ Tímto printf nahradíme
fake_printf:
  PUSH {r0,r1,lr}
  LDR  r0, =pocitadlo  @ Zvýšíme počítadlo o 1
  LDR  r1, [r0]
  ADD  r1, #1
  STR  r1, [r0]
  LDR  r0, =zprava     @ Vypíšeme ho
  BL   orig_printf
  POP  {r0,r1,lr}  @ Vypíšeme, co chtěl volající
  B    orig_printf

pocitadlo:
  .WORD 0

zprava:
  .ASCIIZ "%d: "  @ zkratka za .ASCII + .BYTE 0
  .ALIGN 4

konec:
```

Martin „Medvěd“ Mareš



Vzorová řešení čtvrté série třicátého ročníku KSP

30-4-1 Černobílé hádání

Cílem je nalézt rozumně dobrou strategii pro zjištění, kde jsou bílá a černá políčka v poli, do kterého nemáme přístup. Můžeme se nicméně ptát, zda je v nějakém úseku aspoň jedno políčko černé/bílé, tedy o úseku zjistit, jestli je čistě černý, čistě bílý, anebo míchaný.

Úlohu můžeme samozřejmě vyřešit postupným položením N dotazů a pokud bychom o poli nic nevěděli, byla by to i optimální strategie. Máme ale slíbeno, že pole bude tvořeno jednobarevnými úseky a bude jich řádově méně než celkový počet políček. Mohli bychom binárně vyhledávat konec každého úseku – testovat jednobarevnost intervalů začínajících na konci předchozího úseku. Tak dostaneme za řádově $\log N$ dotazů přesnou informaci o tom, kde jeden úsek končí, a když to uděláme pro každý z M úseků, tak celé pole projdeme za $\mathcal{O}(M \log N)$.

Trochu detailněji popsáno: Při binárním vyhledávání začneme dotazem v polovině, tedy dotazem na interval $[0, N/2]$. Tím zjistíme, jestli je konec v první nebo druhé půlce pole. Pak postup opakujeme pro tu polovinu, kde má být konec – například $[0, 3/4 \cdot N]$, kdybychom prvním dotazem zjistili, že interval je jednobarevný a konec v první půlce není. Abychom otestovali jednobarevnost úseku, tak se můžeme zeptat na první políčko (jedním dotazem „je úsek $[0, 0]$ bílý?“) a pak nám pro každý další stačí zjistit, jestli má stejnou barvu, jako ten první (např. dotazem „je úsek $[0, X]$ bílý?“). Až takto najdeme, jak je velký první jednobarevný úsek, tak se můžeme pustit do druhého. Protože ten první úsek už nebudeme potřebovat, můžeme jeho barvu vypsat a „zapomenout“, že tam něco bylo – snížíme si M o jedna, N o délku prvního úseku a všechny následující dotazy budeme posílat posunuté o délku prvního úseku.

Celkem použijeme přinejhorším $\mathcal{O}(M \log N)$ dotazů, v každé iteraci jeden na zjištění barvy a $\log N$ na dohledání konce. Žádné složité výpočty mimo dotazování neprovádíme, časová složitost bude také $\mathcal{O}(M \log N)$ (pokud vypisujeme pole po skupinách; kdybychom ho vypisovali po dílkách, dostaneme časovou složitost $\mathcal{O}(N + M \log N)$). Za povšimnutí také stojí, že pokud budeme výstup rovnou vypisovat, nepotřebujeme alokovat žádnou paměť, kromě konstantního množství proměnných.

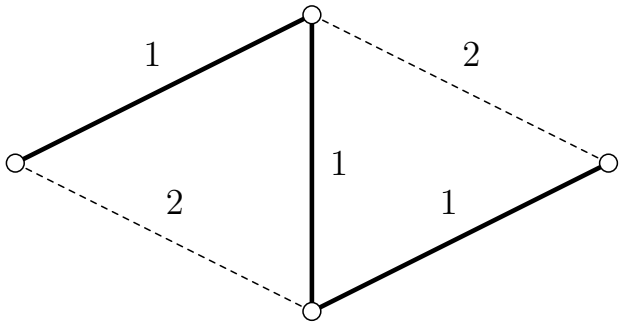
Standa Lukeš

30-4-2 Kočka na stromě

Nefunkční řešení

Na první pohled to může vypadat, že by mohly jít najít dvě nejkratší cesty „postupně“ – najdu jednu nejkratší cestu a potom druhou v grafu bez vnitřních vrcholů té první.

Takové řešení se ale rozbije například na grafu na obrázku. I přesto, že dvě cesty mezi nimi existují, první hledání nejkratší cesty by mohlo najít cestu, jejíž odstranění z grafu vrcholy úplně odstřihne.

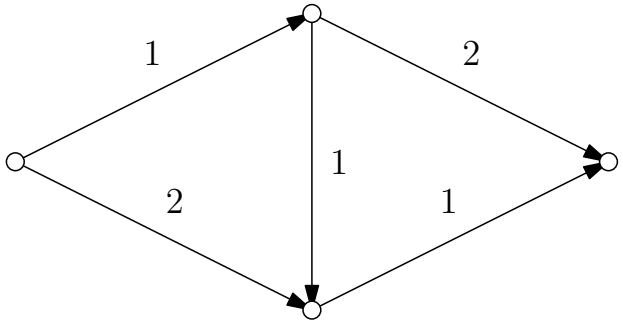


Funkční řešení

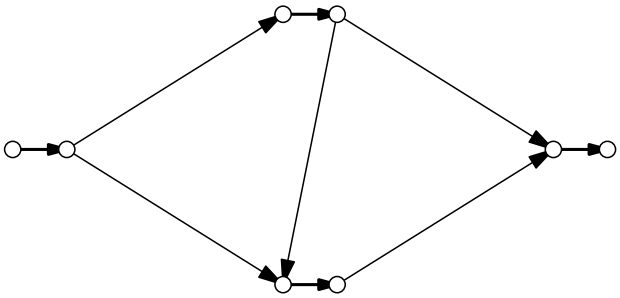
Mohli jste si všimnout, že úloha je kuchařková a kuchařka je o tocích v síti, takže asi nebude špatný nápad toky použít. V kuchařce je dokonce popsáno, jak najít dvě (hranově) disjunktní cesty, jenom ta cesta nemusí být nejkratší. Mohli bychom ale pomocí trochu modifikovaného Dijkstrova algoritmu vyrobit graf, kde všechny cesty budou nejkratší – nebude tam žádná delší cesta než nejkratší cesta mezi hasiči a kočkou. Pak v tomto grafu všechny hrany ohodnotíme jedničkou a najdeme největší tok. Když bude velký alespoň 2, tak v něm najdeme dvě cesty mezi hasiči a kočkou a obě budou nejkratší.

Abychom vyrobili orientovaný graf se samými nejkratšími cestami, prohledáme mapu skoro-Dijkstrovým algoritmem a skončíme, až když vzdálenost převýší vzdálenost kočky (ne hned, jak k ní dojdeme, jako bychom dělali běžně). Když prohledávání zastavíme až po překročení vzdálenosti, máme jistotu, že tam už žádná další nejkratší cesta není.

Během průběhu si navíc budeme psát, přes jaké hrany se dá do každého vrcholu dostat. Když narazíme na alternativní nejkratší cestu do vrcholu, tak ji přidáme do seznamu. Delší cesty tam nepřidáváme. Do výsledného grafu zahrneme jen hrany, které jsme si při prohledávání poznamenali, a ať v něm půjdeme po orientovaných hranách ke kočce jakoukoli cestou, bude nejkratší, protože žádné hrany, které by cestu mohly prodloužit, jsme do grafu jednoduše nepřidali. Na obrázku vidíte co tato transformace udělala s grafem z prvního obrázku. Tentokrát jsou tam všechny hrany, ale to je spíše náhoda.



Teď potřebujeme najít nejlepší tok v síti s průtokem maximálně 1 ve všech hranách a vrcholech. Drobný problém je, že Fordův-Fulkersonův algoritmus z kuchařky umí limitovat jen průtok v hranách, ne ve vrcholech. To ale obejdeme jednoduchým trikem – nahradíme každý vrchol dvěma vrcholy spojenými hranou s kapacitou 1. Všechny vstupní hrany dáme do prvního vyrobeného vrcholu a všechny výstupní do druhého. Takto nám vrcholem poteče maximálně 1. Na obrázku je příklad takového převedeného grafu.



Když máme tok, tak podle něj jenom najdeme cestu, jako je to popsané v kuchařce. Jednoduše půjdeme (projdeme prohledáním do hloubky) jednou cestou, kudy teče tok, a pak půjdeme druhou cestou, kde teče.

Rychlé řešení

Toto řešení funguje a doběhne v polynomiálním čase, ale hledání maximálního toku docela trvá. Můžeme ovšem využít faktu, že nepotřebujeme maximální tok, nýbrž nám stačí jen nějaký tok velikosti 2 (nebo informace, že neexistuje). Fordův-Fulkersonův algoritmus běží v krocích, v každém kroku najde prohledáním grafu do šířky zlepšující cestu, a protože jsou všechny kapacity 1, každá zlepšující cesta zlepší tok o 1. Standardní algoritmus se zastaví, až když žádnou další zlepšující cestu nenajde, což nemusí být zrovna brzy. My ho ale můžeme zastavit už po dvou krocích, tok velikosti 2 nám na dvě paralelní cesty bohatě stačí.

Na celý algoritmus tak potřebujeme prohledat graf podobně, jako to dělá Dijkstrův algoritmus, což zabere $\mathcal{O}(M \log N)$ času. Pak vyrobíme tokovou síť, to stihneme v lineárním čase. Na tokové síti najdeme dvakrát zlepšující cestu prohledáním do šířky, takže to také bude trvat lineárně dlouho. Nakonec v nalezeném toku najdeme dvě cesty, opět to nebude problém stihnout v lineárním čase nějakým prohledáním grafu. Takže ve výsledku jsme se dostali na stejný čas, jako má Dijkstrův algoritmus – $\mathcal{O}(M \log N)$.

Standa Lukeš

30-4-3 Hippocoin

Nejprve provedeme několik jednoduchých pozorování. Uvědomíme si, že pokud chceme obchodovat, vždy se vyplatí využít všechny peníze (ať už koruny, nebo hippocoiny), které máme. Buď se totiž jedná o nějakou transakci, která je pro nás výhodná, a je tedy dobré investovat vše, a nebo nevýhodná, kterou je lepší vůbec neuskutečnit. Dále si uvědomíme, že je výhodné nakupovat hippocoiny těsně před tím, než nákupní cena začne růst, a prodávat je těsně před tím, než prodejní cena začne klesat (takové body se nazývají lokální minimum, respektive maximum). Pokud bychom totiž hippocoiny chtěli koupit dříve (za stejnou nebo vyšší cenu), stejně bychom s prodejem museli počkat na růst nákupní ceny, abychom je mohli výhodně prodat (za vyšší cenu, než jsme je koupili), protože prodejní cena je vždy menší nebo rovna nákupní. Naopak pokud bychom hippocoiny chtěli kupovat později, jednak bude nákupní cena vyšší, jednak se zdržením můžeme připravit o možnost výhodného prodeje.

Jednoduchá varianta

Řešením jednoduché varianty je algoritmus, který v každém minimu hippocoiny nakoupí a v každém maximu je prodá.

Stačí jednou projet celou posloupnost a řídit se podle následujících dvou podmínek:

1. Pokud další den cena poroste, nakup za všechny peníze hippocoiny.
2. Pokud další den cena klesne nebo je poslední den, prodej všechny hippocoiny.

Dokázali jsme, že se nevyplatí obchodovat jinde než v minimech, respektive maximech, nyní dokážeme ještě, že se nevyplatí žádné minimum nebo maximum vynechat. Při našem obchodování pravidelně střídáme nákup a prodej. Po každé dvojici transakcí *nákup hippocoinů – jejich prodej za vyšší cenu* budeme mít víc peněz než před ní. A protože cena v každém maximu je vždy vyšší než cena v předchozím minimu, vyplatí se provést všechny tyto transakce. Popsaný algoritmus je tedy správným řešením s časovou složitostí $\mathcal{O}(N)$, protože projdeme posloupnost pouze jednou, a paměťovou složitostí $\mathcal{O}(1)$, protože nám stačí pamatovat si cenu v dnešním a v následujícím dni.

Rozšíření na těžkou variantu

Přímočarým řešením těžší varianty by byl stejný algoritmus jako u jednoduššího řešení, který by ovšem bral v úvahu pouze minima z nákupní ceny a maxima z prodejní. To ale nestačí. Jednak se mohou objevit dvě minima v nákupní ceně za sebou, aniž by mezi nimi bylo maximum z ceny prodejní (musíme se tedy rozhodnout, ve kterém minimu chceme nakoupit), jednak maximum prodejní ceny může být menší než minimum nákupní ceny a nemusí se nám vyplatit vždy obchodovat, protože bychom neprodali hippocoiny za vyšší cenu, než jsme je koupili. Budeme si tedy pamatovat poslední transakci (nákup nebo prodej nenulového množství peněz a jeho cenu) a přidáme dvě podmínky:

3. Pokud chceme provést transakci stejného typu jako předchozí a dnešní cena je výhodnější, zruš poslední transakci a proveď ji až dnes.
4. Pokud chceme prodat hippocoiny a prodejní cena je nižší než ta, za kterou jsme je nakoupili, prodej neprováděj.

Zbývá nám už jen jedna maličkost – zajistit, že poslední den budeme mít koruny a ne hippocoiny. V lehčí variantě jsme to vyřešili tak, že jsme hippocoiny nakupovali, pouze když jsme věděli, že cena ještě poroste. Tady však nevíme, jestli prodejní cena ještě překročí tu nákupní, proto přidáme poslední podmínku:

5. Pokud jsme dojeli na konec a poslední transakce je nákup, transakci zruš.

Algoritmus má stejně jako u jednodušší varianty časovou složitost $\mathcal{O}(N)$ a paměťovou $\mathcal{O}(1)$.

Jiné řešení podle Jirky Škrobánka

Vyjdeme z toho, že pokud známe maximální možný počet korun k_i a hippocoinů h_i, který můžeme i-tý den mít, pak $(i+1)$-ní den je maximální možný počet korun maximum z k_i (pokud si necháme koruny v korunách) a $h_i \cdot p_{i+1}$ (pokud si převedeme včerejší částku v hippocoinech na koruny), kde p_{i+1} je prodejní cena hippocoinů $(i+1)$-ní den.

Na začátku máme dané množství peněz k_0. Pokud si je hned první den převedeme na hippocoiny, získáme h_0. Projdeme celou posloupnost a budeme si vždy pamatovat aktuální k_i a h_i. Jakmile dojdeme na konec, zjistíme nejvyšší počet korun, který můžeme při obchodování během dané doby získat. Pokud chceme znát i postup, jak obchodovat, budeme si muset pamatovat kromě k_i a h_i také to, kdy jsme peníze měnili.

I toto řešení má lineární časovou a paměťovou složitost.

Zuzka Urbanová

30-4-4 Malování 2.0

Před samotným hledáním algoritmu si pojďme projít detaily zadání. Máme pouze dvě barvy, takže o obrázku můžeme uvažovat jako o bitové mapě. Štětec nám barvy pouze prohazuje, nezáleží tudíž na přesném počtu kliknutí na pixel, ale pouze na sudosti a lichosti. Každé dvě aplikace štětce na tomtéž místě se navzájem vyruší. U každého pixelu obrázku se tedy musíme pouze rozhodnout, jestli na něj kliknout, nebo ne. Navíc o štětci můžeme uvažovat jako o binární operaci XOR – malování je XORování s jedničkou.

Pokud jde obrázek štětcem namalovat, tak jde určitě i smazat. Stačí ho namalovat podruhé přes sebe sama. Dostaneme tím v každém bodě obrázku dvě aplikace štětce nebo žádnou – celkově žádnou změnu. A naopak: pokud jde smazat, stejný postup aplikovaný na prázdnou plochu ho namaluje.

Pojďme tedy obrázek smazat! Kde ale začít? Podíváme-li se na rohové pixely obrázku, tak každý z nich je možné změnit pouze jedním možným způsobem. Uvažujme pouze o levém horním rohu obrázku. Můžeme aplikovat štětec přímo v rohu, tím se pak barva pixelu změní. Když ale klikneme kamkoliv jinam, tak se pixel akorát dostane mimo dosah. Takže barva levého horního pixelu nám pevně určuje první aplikaci štětce – musíme tam dostat bílou. Akorát při kliknutí musíme změnit barvu celé oblasti $k \times k$ a nejen rohového pixelu. Co dál? Levý horní pixel považujeme za vyřešený, takže na něj již nechceme sahat. Vznikly nám tím dva nové rohové pixely. Vezměme třeba ten vpravo – můžeme ho také změnit pouze jedním způsobem.

Budeme-li takto pokračovat, dostaneme se na konec prvního řádku. Na posledních $k-1$ pixelů ale nejde přímo kliknout, takže pokud nejsou bílé, tak obrázek smazat nejde. Tím jsme vyřešili první řádek a můžeme to samé opakovat pro další. Posledních $k-1$ řádků opět nepůjde změnit přímo. Ty opět rozhodují o tom, jestli je obrázek smazatelný. Celý výsledný algoritmus vypadá takto:

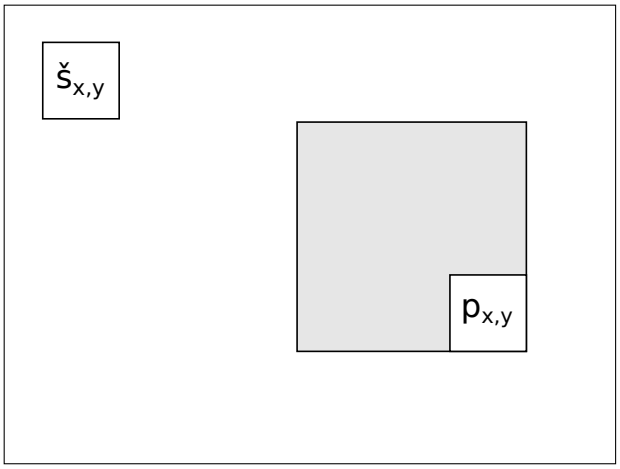
```
1. Pro každý řádek x:
2.   Pro každý sloupec y:
3.     Když je pixel [x,y] černý:
4.       Když se štětec na [x,y] vejde na obrázek:
5.         Změníme barvu všem políčkům štětce.
6.       Když se nevejde:
7.         Obrázek nejde nakreslit.
8. Obrázek nakreslit jde.
```

Jelikož krok 5 trvá $\mathcal{O}(k^2)$, celková složitost tohoto algoritmu činí $\mathcal{O}(MNk^2)$. Paměť nám stačí na uložení obrázku, takže $\mathcal{O}(MN)$.

To musí jít rychleji!

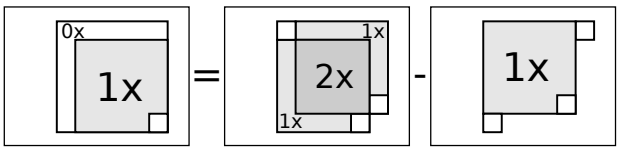
Proč je předchozí algoritmus pomalý? Při každé aplikaci štětce musíme změnit barvu k^2 pixelů. Dá se to ale obejít. Protože se jedná o souvislou oblast, můžeme si pomoci 2D intervalovými stromy. Tím bychom místo $\mathcal{O}(MNk^2)$ dostali složitost $\mathcal{O}(MN \log M \log N)$. To může být lepší v závislosti na velikosti k. Jde to ale ještě rychleji a pomocí méně komplikované datové struktury!

Vezměme si nějaký pixel uvnitř obrázku. Kolikrát mu změníme barvu? Tolikrát, kolikrát jsme ve čtverci vlevo nad ním velikosti $k \times k$ použili štětec. Na toto můžeme použít variantu prefixových součtů. U každého pixelu si budeme pamatovat, kolikrát byla jeho barva změněna. Pro pixel na souřadnicích $[x, y]$ říkejme této hodnotě $p_{x,y}$. Začínáme s tím, že každý pixel nebyl nikdy prohozen, takže si každý pamatuje nulu. Navíc si u každého pixelu pamatujme, jestli na něj byl použit štětec – hodnotu $\check{s}_{x,y}$. Ta nabývá hodnot nula nebo jedna.



Budeme obrázek opět procházet z levého horního rohu a u každého pixelu se zastavíme a rozhodneme se, jestli na něj potřebujeme použít štětec, nebo ne. Pak spočítáme jeho hodnotu p a uložíme si ji.

Protože krajní vrcholy je potřeba ošetřit speciálně, začněme pro ukázku nějakým vnitřním pixelem obrázku. Řekněme, že leží na souřadnicích $[x, y]$. Než jsme se k tomuto pixelu dostali, prošli jsme všechny pixely vlevo a nahoře. Takže speciálně máme určitě spočítané hodnoty $p_{x-1,y}$, $p_{x,y-1}$ a $p_{x-1,y-1}$. Kolikrát byla pixelu $[x, y]$ změněná barva? Nejlépe je to vidět asi z následujícího obrázku:



Dostáváme tedy výraz $p_{x,y} = p_{x-1,y} + p_{x,y-1} - p_{x-1,y-1} - \check{s}_{x-k,y} - \check{s}_{x,y-k} + \check{s}_{x-k,y-k}$, díky kterému jsme schopní hledaný počet prohození barvy konkrétního pixelu spočítat v konstantním čase. Navíc nás nutně nezajímá přesný počet, můžeme vše počítat modulo 2. Takže také víme, jestli je po všech těchto prohozeních pixel bílý nebo černý a můžeme postupovat stejně jako v prvním algoritmu.

Abychom mohli algoritmus dokončit, musíme se ještě vypořádat s případy, kdy chceme číst hodnoty p a $\check{s}$ z oblastí mimo obrázek. To můžeme vyřešit velmi jednoduše, mimo hranice obrázku prostě žádné změny dělané nebyly, takže obě hodnoty jsou nulové. Celý algoritmus tedy nakonec vypadá následovně:

1. Pro každý řádek x:
2. Pro každý sloupec y:
3. $p_{x,y} \leftarrow (p_{x-1,y} + p_{x,y-1} - p_{x-1,y-1} - \check{s}_{x-k,y} - \check{s}_{x,y-k} + \check{s}_{x-k,y-k}) \bmod 2$
4. Když je pixel $[x, y]$ černý po $p_{x,y}$ prohozeních:
5. Když se štětec vejde na obrázek:
6. $\check{s}_{x,y} \leftarrow 1$
7. $p_{x,y} \leftarrow 1 - p_{x,y}$
8. Když se nevejde:
9. Obrázek nakreslit nejde.
10. Jinak $\check{s}_{x,y} \leftarrow 0$
11. Obrázek nakreslit jde.

Výslednou složitost dostáváme nakonec $\mathcal{O}(MN)$. Paměťová je stejná, protože v každém pixelu si udržujeme pouze konstantní množství informace – dvě čísla.

Vašek Šraier

Líné přebarvování

Pomalé přebarvování čtverců $k \times k$ z našeho prvního řešení jde také zrychlit takzvaným *líným vyhodnocováním.* Pořídíme si pomocné pole ℓ, ve kterém budou jedničky na místě *požadavků* na přebarvení: $\ell_{x,y} = 1$ bude znamenat, že všechna políčka v „nekonečném čtverci“ od políčka $[x, y]$ doprava a dolů se mají přebarvit. Všimneme si, že přebarvení čtverce $k \times k$ s levým horním rohem $[x, y]$ lze provést znegováním požadavků $\ell_{x,y}$, $\ell_{x+k,y}$, $\ell_{x,y+k}$ a $\ell_{x+k,y+k}$. To je stejný trik, jako u 2D prefixových součtů.

Dobrá, tím jsme přebarvování odložili na později. Kdo ho ale provede? Dohodneme se, že kdykoliv dojdeme na nějaké políčko $[x, y]$ a leží na něm požadavek, tak tento požadavek vyřešíme zadáním nějakých požadavků na ještě neprojitých políčkách. Konkrétně si uvědomíme, že požadavek $\ell_{x,y} = 1$ můžeme vyřídit znegováním samotného políčka $[x, y]$ a požadavků $\ell_{x+1,y}$, $\ell_{x,y+1}$ a $\ell_{x+1,y+1}$.

Zadat požadavek i vyřešit ho tedy dokážeme v konstantním čase, takže celé líně přebarvovací řešení má složitosti $\mathcal{O}(MN)$. Mimochodem, všimněte si, že je vlastně docela podobné předchozímu rychlému řešení, jen něco jako prefixové součty počítá pro budoucnost místo minulosti.

Martin „Medvěd“ Mareš

30-4-5 Frňákovník

Jakým nejjednodušším způsobem by se tato úloha dala vyřešit? Můžeme použít hrubou sílu. Vyzkoušíme všechny možné kombinace nádob (je jich 2^N), spočítáme jejich celkový objem, zkontrolujeme dělitelnost sedmi a najdeme maximum. Rychlé to ale nebude, kvůli počtu všech kombinací dostáváme časovou složitost $\mathcal{O}(2^N \cdot N)$ a paměťovou složitost $\mathcal{O}(N)$.

Jak bychom to implementovali? Docela se na to hodí rekurze, protože se jí hezky dají generovat ty kombinace. Vezmeme první nádobu dohromady se všemi rekurzivně spočítanými kombinacemi ostatních nádob. Tím dostaneme půlku všech kombinací. Pro druhou půlku první nádobu zahodíme a vezmeme opět rekurzivně všechny možné kombinace zbytku. Abychom si nemuseli kombinace ukládat, můžeme v poslední vrstvě rekurze zkontrolovat součet a hledat maximum. Je potřeba to ale nějak zrychlit!

Jedno drobné vylepšení se hned nabízí. Nemusíme počítat součty až po výběru nádob, můžeme je sčítat již průběžně při rekurzivním generování a předávat si do rekurze součet. Snižujeme tak časovou složitost na $\mathcal{O}(2^N)$. Můžeme to ale také celé otočit. Rekurzivní funkce nebude vytvářet kombinace, bude počítat přímo výsledek – největší možný součet objemů nádob dělitelný sedmi. Vezmeme první nádobu a největší součet ze zbylých nádob se zbytkem po dělení sedmi takovým, aby nám nakonec vyšel zbytek nulový. To samé zkusíme s tím, že první nádobu nebudeme započítávat.

V kódu to může vypadat například následovně:

```
N = 5
nadoby = [2, 3, 2, 2, 17]

def vyber(zacatek, zbytek):
  # vrací celkový objem a seznam nádob

  # když už nejsou nádoby, ukončíme rekurzi
  if zacatek == N:
      return 0, [] # rozbije se pro zbytek != 0

  # tady vyzkoušíme obě varianty součtů
  z = nadoby[zacatek] % 7
  soucet_s, vybrane_s =
    vyber(zacatek + 1, (7 + zbytek - z) % 7)
  soucet_bez, vybrane_bez =
    vyber(zacatek + 1, zbytek)

  # a vezmeme tu lepší
  if soucet_s > soucet_bez:
    return (soucet_s + nadoby[zacatek],
            vybrane_s + [nadoby[zacatek]])
  else:
    return soucet_bez, vybrane_bez

print(vyber(0, 0))
```

Stále je to ale pomalé. Funkce `vyber` se spustí řádově 2^N-krát. Jaké ale dostává vstupní parametry? Index do pole s nádobami – těch je N – a zbytek po dělení sedmi – těch je celkem sedm. Takže máme celkem $7N$ možných způsobů, jak zavolat funkci, ale voláme ji 2^N-krát. To je spousta zbytečné práce! To ji přeci musíme volat se stejným vstupem vícekrát! Pojďme si pořídit cache a ukládejme si do ní předchozí výsledky. Díky tomu nebude potřeba počítat pořád dokola to samé. Sníží nám to časovou složitost na krásných $\mathcal{O}(N)$. Zůstává tam ale pořád ta rekurze, kterou z praktických důvodů nechceme mít moc hlubokou (kvůli velikosti zásobníku).

Jak naší cache vyplňujeme? Rekurze se první zanoří do maximální hloubky, spočítá výsledek pro nula nádob, uloží ho do cache a vrátí se o vrstvu výše. S pomocí cache pak spočítáme hodnotu pro jednu nádobu, pro dvě, atd. To ale není nic jiného, než lineární procházení cache odzadu. Takže vlastně rekurzi nepotřebujeme, můžeme to samé spočítat přímo.

Cache má velikost $\mathcal{O}(7N) = \mathcal{O}(N)$ a můžeme si ji reprezentovat například jako dvourozměrné pole. Obsahuje pro každý počáteční index a zbytek po dělení sedmi objem nejlepšího výběru nádob a odkaz na stav, ze kterého byla tato hodnota vypočítána. Můžeme díky tomu rekonstruovat výběr nádob.

Cache budeme lineárně procházet od posledních nádob. Pro každou nádobu navíc projdeme všechny zbytky po dělení sedmi a z předchozích hodnot dopočteme ještě nespočítané stejně jako při rekurzi. Když je cache plná, tak poslední spočítaná vrstva obsahuje výsledek u zbytku rovného nule. Navíc odkazuje na předchozí použitý stav, takže si z toho umíme spočítat, jestli byla první nádoba použitá nebo ne. Obdobně pro všechny další nádoby.

Ještě si můžeme všimnout jednoho drobného detailu – rekurze cache vyplňuje pozpátku, ale ve výsledku je to úplně jedno. Můžeme průchod otočit a začít od první nádoby, skončit u poslední. Nic to na situaci nezmění.

Vyřešeno tedy máme v čase $\mathcal{O}(N)$ a ve stejné paměti. Použitá technika postupného vylepšování rekurzivního řešení spadá pod tzv. *dynamické programování*. Více se o tomto tématu můžete dočíst v naší kuchařce.[5]

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/30-4-5.py`

Vašek Šraier

30-4-6 Takřka nudná úloha

Připomeňme si úlohu: máme dvě posloupnosti A a B, obě délky n. Chceme v lineární paměti spočítat jejich nejdelší společnou podposloupnost, tj. vyškrtat z nich co nejméně prvků, aby se zbylé posloupnosti rovnaly.

Jak se píše v zadání, bez požadavku na lineární paměť jde o známou úlohu. Připomeňme řešení z kuchařky o dynamickém programování.[6] Výpočet probíhá v n krocích, v i-tém kroku uvažujeme jen společné podposloupnosti A a B, které v A končí nejpozději na pozici i. Přechod z i-tého do $(i+1)$-tého kroku probíhá tak, že se podíváme, jak se dosavadní posloupnosti prodlouží, když jim povolíme využít navíc i prvek $A[i+1]$.

Neuvažujeme však všechny podposloupnosti, ale jen ty, které mají potenciál být prodlouženy na nejdelší společnou. V kuchařce je zdůvodněno, že v jednom kroku nám stačí pamatovat si jen jednu podposloupnost od každé délky, a to konkrétně tu, která v B končí co nejdříve. Říkejme takovým podposloupnostem *slibné*. Navíc si z paměťových důvodů nepamatujeme všech těchto $\mathcal{O}(n)$ podposloupností, ale jen jejich poslední prvky, resp. indexy posledních prvků v poli B.

Označme tento index pro slibnou podposloupnost délky ℓ v i-tém kroku jako $d_i[\ell]$. Kuchařkové řešení ve své podstatě funguje tak, že se při přechodu k $(i+1)$-tému kroku podívá na všechny slibné podposloupnosti z i-tého kroku a podívá se, kde by v B nejdříve končily, kdyby se prodloužily o prvek $A[i+1]$. V praxi to znamená, že se vezme hodnota $d_i[\ell]$, v B se od ní napravo najde index nejbližšího výskytu prvku $A[i+1]$ a tímto indexem se pokusíme zlepšit hodnotu $d_{i+1}[\ell+1]$ (tj. nahradíme ji, právě pokud je nový index menší než současná hodnota).

Každou hodnotu $d_i[\ell]$ jsme pak mohli získat dvěma způsoby: buď vylepšením z $d_{i-1}[\ell-1]$, nebo ponecháním $d_{i-1}[\ell]$. Když si tedy ke každému $d_i[\ell]$ připíšeme, jak jsme ho získali, jsme schopni zpětně zrekonstruovat nejdelší společnou podposloupnost: najdeme nejvyšší ℓ, pro které je $d_n[\ell]$ definované, a budeme zpětně následovat „šipky“ a za každý přechod z $d_i[\ell]$ do $d_{i-1}[\ell-1]$ současný prvek přidáme do nejdelší společné podposloupnosti. Označme jako p_i hodnotu ℓ v i-tém kroku podle tohoto postupu rekonstrukce. Všimněte si, že k rekonstrukci nám stačí znát všechna p_i a k nim příslušné hodnoty $d_i[p_i]$ a zbytek pole d_i vlastně nepotřebujeme.

Pomalé řešení

Problém kuchařkového přístupu je, že si kvůli rekonstrukci musíme pamatovat všechna pole d_i. Přímočaré řešení se hned nabízí: Nebudeme si při rekonstrukci pamatovat všechna d_i, ale vždy jen to d_i, které právě potřebujeme. Když pak přecházíme od d_i k d_{i-1}, zapomeneme pole d_i, načež spustíme celý algoritmus od začátku a zastavíme ho

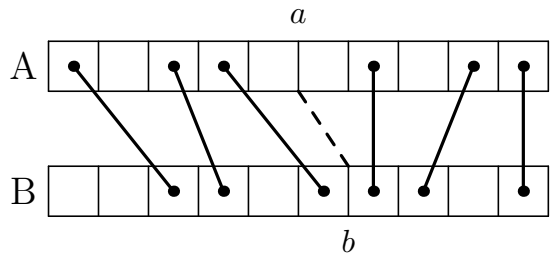
[5] `http://ksp.mff.cuni.cz/viz/kucharky/dynamika`
[6] `http://ksp.mff.cuni.cz/viz/kucharky/dynamika`

v $(i-1)$-tém kroku, abychom spočítali pole d_{i-1}. Takto celkem algoritmus zavoláme $\mathcal{O}(n)$-krát a celková časová složitost bude $\mathcal{O}(n^3)$.

Zrychlujeme

Začneme pozorováním: kdykoliv jsme v kroku i a máme spočítané pole d_i, jsme schopni v čase $\mathcal{O}(n(n-i))$ a paměti $\mathcal{O}(n)$ zjistit hodnotu p_i, tedy zjistit, která ze slibných posloupností v poli d_i se nakonec prodlouží na tu nejdelší. Můžeme si totiž každý prvek d_i pomyslně obarvit a nechat kuchařkový algoritmus postupně spočítat $d_{i+1}, \ldots, d_n$, přičemž každé políčko $d_j[k]$ vždy obarvíme barvou políčka, ze kterého získalo svou hodnotu. Na konci se pak jen podíváme na barvu, kterou má nejdelší společná podposloupnost.

Nyní toto pozorování využijeme spolu s radou ze zadání navádějící k použití metody Rozděl a panuj. Spustíme kuchařkový algoritmus a pozastavíme ho v kroku $n/2$ (BÚNO je n sudé, jinak zaokrouhlíme dolů) a zapamatujeme si pole $d_{n/2}$. Pak algoritmus necháme doběhnout do konce a pomocí výše popsaného postupu zjistíme hodnotu $p_{n/2}$.



Jak ilustruje obrázek, teď jsme si celý problém rozdělili na dva menší. Označme $a = n/2$ a $b = d_{n/2}[p_{n/2}]$. Spočítáním $p_{n/2}$ jsme nejdelší společnou podposloupnost A a B rozdělili na dvě části: nejdelší společnou podposloupnost $A[1 \ldots a]$ a $B[1 \ldots b]$ a nejdelší společnou podposloupnost $A[a+1 \ldots n]$ a $B[b+1 \ldots n]$. Můžeme tedy rekurzivně vyřešit nejprve levou část a pak pravou. Postupně tak spočítáme všechna p_i a hodnoty $d_i[p_i]$ a z nich už snadnou úpravou původního algoritmu pro rekonstrukci získáme nejdelší společnou podposloupnost.

Jakou má tento algoritmus složitost? Nejprve si rozmyslíme, že jsme se opravdu vešli do lineární paměti. Na začátku si vždy udržujeme nejvýše dvě lineárně velká pole d_*: d_i a d_{i+1}; později nám k tomu přibude ještě pole $d_{n/2}$. Jakmile kuchařkový algoritmus doběhne, stačí nám pamatovat si jen $\mathcal{O}(1)$ hodnot: $p_{n/2}$ a hodnotu $d_{n/2}[p_{n/2}]$. Rekurze nám analýzu trochu komplikuje, ale stačí si uvědomit, že vždy běží nejvýše jedno rekurzivní volání najednou, které (jak už jsme si rozmysleli) samo o sobě spotřebuje lineárně mnoho pomocné paměti, kterou po svém doběhnutí uvolní. Všechna ostatní aktuálně neběžící volání spotřebovávají jen $\mathcal{O}(1)$ paměti a je jich $\mathcal{O}(n)$.

Časová složitost vyjde $\mathcal{O}(n^2)$. Abychom to nahlédli, uvědomíme si, že na jedné úrovni rekurze, kde zkoumáme nějaký úsek A velký a a úsek B velký b, strávíme nejvýše ab jednotek času.[7] To je triviální, protože pouze pouštíme kuchařkový algoritmus na posloupnosti velké a a b a děláme navíc konstantní množství další práce. Celkem tedy na první úrovni strávíme nejvýše n^2 jednotek času a pak se zavoláme na oba podproblémy, jeden velikosti $n/2$ a b, druhý velikosti $n/2$ a $n-b$. Na těch strávíme nejvýše $n/2 \cdot b + n/2 \cdot (n-b) = n/2 \cdot n = n^2/2$ jednotek času. Z nich se opět zavoláme na ještě menší podproblémy, na kterých dohromady strávíme $n/4 \cdot b_1 + n/4 \cdot (b-b_1) + n/4 \cdot b_2 + n/4 \cdot (n-b-b_2) = n^2/4$ času. Takto to pokračuje dál a celkem za všechna volání můžeme strávený čas odhadnout jako $n^2 + n^2/2 + n^2/4 + \ldots \leq 2n^2 = \mathcal{O}(n^2)$ pomocí známého vzorečku pro součet geometrické řady.

Na závěr poznamenáme, že na tento algoritmus přišel v roce 1975 jistý Dan Hirschberg a na jeho počest se algoritmus jmenuje Hirschbergův.

Říša Hladík

30-4-7 Překřikující se procesory

Úkol 1: Read-write spinlock

Read-write (RW) spinlocky je možné implementovat mnoha způsoby. My si stav RW spinlocku uložíme do jedné 32-bitové proměnné, se kterou budeme zacházet pomocí atomických instrukcí `LDREX` a `STREX`. Je-li zámek odemčen, v proměnné bude 1. Je-li zamčen pro zápis, uložíme 0. Pokud ho má zamčeno pro čteni k procesorů, uložíme $k+1$.

Zamčení pro zápis funguje takto: čekáme, dokud v proměnné nebude jednička, a tu pak atomicky změníme na nulu. Řídíme se volací konvencí, takže parametr s adresou zámku očekáváme v `r0`.

```
zamkni_pro_zápis:
  LDREX r1, [r0]
  CMP   r1, #1
  BNE   zamkni_pro_zápis
  MOV   r1, #0
  STREX r2, r1, [r0]
  CMP   r2, #0
  BNE   zamkni_pro_zápis
  BX    lr
```

Pokud chceme zamykat pro čtení, čekáme, dokud proměnná je nulová, a jakmile přestane být, zvýšíme ji atomicky o 1.

```
zamkni_pro_čtení:
  LDREX r1, [r0]
  CMP   r1, #0
  BEQ   zamkni_pro_čtení
  ADD   r1, #1
  STREX r2, r1, [r0]
  CMP   r2, #0
  BNE   zamkni_pro_čtení
  BX    lr
```

Odemčení po zápisu pouze přepíše 0 na 1. Jelikož víme, že v tomto okamžiku máme k zámku exkluzivní přístup, nemusíme používat `STREX`, stačí nám vědět, že zápis jednoho slova do paměti je atomický.

```
odemkni_po_zápisu:
  MOV   r1, #1
  STR   r1, [r0]
  BX    lr
```

Odemčení po čtení musí proměnnou snížit o 1, a to už je potřeba provádět atomicky:

```
odemkni_po_čtení:
  LDREX r1, [r0]
  SUB   r1, #1
  STREX r2, r1, [r0]
```

[7] Kde jako jednotku času volíme nějakou vhodnou konstantu. Zde musíme na chvíli odložit $\mathcal{O}$-notaci, protože je pro naše potřeby moc hrubá.

```
  CMP   r2, #0
  BNE   odemkni_po_čtení
  BX    lr
```

Nevýhodou tohoto řešení je, že pokud nějaký procesor chce zapisovat, čtoucí procesory ho mohou neomezeně dlouho blokovat: Představte si, že jeden procesor chce zapisovat a mnoho dalších procesorů velmi často krátkodobě zamyká pro čtení. Počítadlo je proto stále větší než 1 (vždy najdeme aspoň jeden procesor, který chce číst), takže uzamčení pro zápis se nikdy nepovede. Tomuto stavu se někdy říká *vyhladovění*.

Úkol 1: Řešení bez vyhladovění

Ukážeme trochu složitější konstrukci RW spinlocku, která vyhladověním netrpí. Kromě počítadla si pořídíme pomocný obyčejný spinlock S. Při zamykání RW pro čtení zamkneme S a hned ho zase odemkneme. Při zamykání RW pro zápis zamkneme S a neodemkneme ho dřív, než se zamčení RW podaří. Tím způsobíme, že jakmile začneme čekat na zamčení RW pro zápis, další požadavky na zamykání RW už budou čekat a k žádnému vyhladovění nedojde.

Ukážeme implementaci. Na adrese `r0` bude opět uloženo počítadlo, na `r0+4` spinlock. Budeme používat funkce `zamkni` a `odemkni` ze zadání. Opět použijeme standardní volací konvenci, takže si musíme dát pozor na to, že funkce, kterou voláme, může přepsat `r0`.

```
zamkni_pro_zápis_2:
  PUSH  {r0}
  ADD   r0, #4    @ adresa spinlocku
  BL    zamkni
  POP   {r0}
  @ sem vložíme zamkni_pro_zápis bez BX lr
  ADD   r0, #4
  BL    odemkni
  BX    lr

zamkni_pro_čtení_2:
  PUSH  {r0}
  ADD   r0, #4    @ adresa spinlocku
  PUSH  {r0}
  BL    zamkni
  POP   {r0}      @ opět adresa spinlocku
  BL    odemkni
  POP   {r0}      @ adresa počítadla
  @ sem vložíme zamkni_pro_čtení
```

Funkce pro odmykání zůstanou stejné – stačí jim počítadlo a pomocným spinlockem se vůbec nemusí zabývat.

Úkol 2: Problém se dvěma zámky

Deadlocku se zbavíme velice snadno: kdykoliv chceme zamknout více zámků, seřadíme je podle adres v paměti a zamykáme je v tomto pořadí. Odemykat už můžeme v libovolném pořadí.

Pojďme dokázat, že deadlock pak nemůže nastat. Nejprve si rozmyslíme, kdy deadlock vzniká: nějaký procesor se chystá zamknout zámek z_1; ten je ovšem zamčený jiným procesorem, který zrovna čeká na zámek z_2, takže neodemkne z_1 dřív, než získá z_2; další procesor drží z_2 a čeká na z_3; až nakonec nějaký procesor drží z_k a čeká na z_1.

Tuto situaci můžeme elegantně popsat orientovaným grafem. Jeho vrcholy budou zámky, hrana povede ze z_i do z_j, pokud nějaký procesor má zamčený zámek z_i a čeká na z_j. K deadlocku dojde právě tehdy, když v tomto grafu existuje cyklus.

Seřazením zámků podle adres jsme způsobili, že kdykoliv v grafu existuje hrana z_1z_2, leží z_1 na nižší adrese než z_2. Na jakémkoliv cyklu by tudíž musely adresy stále růst, a to není možné. Takže žádný cyklus neexistuje, tedy ani žádný deadlock.

Úkol 3: Fronta

Jak po nás zadání chtělo, frontu reprezentujeme jednosměrným spojovým seznamem (každý prvek seznamu bude v paměti tvořen jedním slovem s hodnotou a jedním s adresou následujícího prvku). Chceme umět rychle jak přidávat na konec seznamu, tak odebírat z jeho začátku. Proto si musíme pamatovat jak adresu prvního prvku (hlavičku seznamu H), tak posledního (ocásek O). Pokud hlavička, ocásek nebo následník prvku neexistují, uložíme nulovou adresu.

Kdyby byl seznam vždy aspoň dvouprvkový, úloha by byla jednoduchá. Stačilo by chránit hlavičku jedním zámkem (Z_H) a ocásek druhým (Z_O). Kdykoliv bychom chtěli přidat na konec, zamkli bychom ocáskový zámek, přidali nový prvek, nastavili adresu následníka v původním ocásku, aktualizovali adresu ocásku a odemkli zámek. Podobně odebírání z hlavičky by obnášelo pouze zamčení hlavičkového zámku.

Ještě si musíme dát pozor na to, abychom za zamčení zámku a před jeho odemčení umístili bariéru zabraňující přesunutí čtení či zápisů přes operace se zámkem. Obecně se hodí bariéry doplnit přímo do implementace zámků, abychom se o to nemuseli explicitně starat.

Ovšem pokud je seznam krátký, věci se začnou komplikovat. Hlavička a ocásek mohou ukazovat na tentýž prvek, nebo dokonce na žádný. Přidání prvku do prázdného seznamu obnáší nastavení jak hlavičky, tak ocásku. Odebrání jediného zbývajícího prvku způsobí vynulování hlavičky i ocásku. Přidání druhého prvku způsobí nastavení adresy následníka v prvním prvku, který mezitím někdo může odebírat. Zkrátka možností, co by se mohlo pokazit, je nemálo.

Proto na to půjdeme chytřeji – kdykoliv bude hrozit, že je seznam příliš krátký, zamkneme pro jistotu oba zámky (vždy nejprve hlavičkový, pak ocáskový, aby nedošlo k deadlocku). Může se nám stát, že hrozba byla nakonec planá, protože před zamčením někdo stihl do seznamu přidat další prvek. To ale nevadí – důležité je, abychom ve všech případech, kdy seznam je triviální, měli zamčeno.

Odebírání z hlavičky bude vypadat následovně: Nejprve zamkneme Z_H. Pak zkontrolujeme, jestli adresa hlavičky je nulová nebo hlavička nemá následníka – v těchto triviálních případech zamkneme i Z_O a můžeme podle potřeby manipulovat jak s hlavičkou, tak s ocáskem. Pokud naopak hlavička existuje a má následníka, můžeme si být jistí tím, že se tento stav během odebírání nezmění, protože kdokoliv další, kdo by chtěl odebírat prvky, čeká na hlavičkový zámek, a ten máme v ruce my – stačí tedy mít zamčený Z_H. Když jsme se vším hotovi, odemkneme Z_H a případně i Z_O.

Přidávání na konec provedeme takto: zamkneme Z_O. Pokud je adresa ocásku nulová, zamkneme i Z_H (přesněji řečeno odemkneme Z_O, zamkneme Z_H a pak znovu Z_O, abychom dodrželi pořadí zámků) a nastavíme hlavičku i ocásek na nově přidaný prvek. Pokud je ocásek nenulový, stačí nám Z_O, nastavíme následníka ocásku na nový prvek a pak nový prvek učiníme ocáskem. Nakonec odemkneme všechny zámky.

Při odebírání je potřeba si rozmyslet, že zatímco připojujeme nový prvek k původnímu ocásku, nemůže nám někdo

původní ocásek pod rukama smazat. Dopadne to dobře: do té doby, než přepíšeme jeho adresu následníka, je tato adresa nulová, takže odebírající procesor by se pokusil získat oba zámky, čili by počkal, až my odemkneme. A jakmile adresu následníka přepíšeme, už původní ocásek nebudeme potřebovat, takže jeho smazání nevadí.

Úkol 4: Seznam počítadel

Pro práci se seznamem zavedeme následující pravidla:

1. Aby seznam nikdy nebyl prázdný a nemuseli jsme na první prvek ukazovat jinak než na ty ostatní, přidáme na začátek seznamu hlavičku. To je pevný prvek, jehož klíč ani počítadlo nebudeme používat.
2. Každý prvek bude vybaven svým zámkem. Obsah prvku (počítadlo a adresu následníka) smíme číst i zapisovat jen tehdy, když máme zamčený příslušný zámek.
3. Seznam procházíme vždy ve směru od začátku do konce. Zkoumáme-li prvek, máme ho zamčený a také máme zamčeného jeho předchůdce.

Procházení seznamu tedy probíhá takto: Nejprve zamkneme hlavičku. Pak z ní přečteme adresu následníka (tedy prvního opravdového prvku), zamkneme tohoto následníka a přesuneme se do něj. Obecně máme zamčený nějaký prvek x a jeho předchůdce p. Vždy přečteme z x adresu následníka n, zamkneme n a odemkneme p. Pak se n stane novým x a x novým p.

V každém okamžiku máme zaručen exkluzivní přístup k úseku seznamu mezi p a x. Díky tomu můžeme pracovat s počítadlem v x, vložit nový prvek mezi p a x, jakož i smazat x.

Jelikož všechny procesory zamykají zámky ve stejném pořadí, nemůže dojít k deadlocku.

Ještě dodejme, že bychom se také mohli pokusit místo zamykání upravovat ukazatele atomicky pomocí `STREX`. Takové pokusy většinou selžou na recyklování adres: pokud prvek vytažený ze seznamu vzápětí zapojíme na jiné místo, můžeme se nachytat na to, že stejná adresa prvku neimplikuje stejnou roli prvku v seznamu.

Závěrem

Jak vidíme, paralelní přístup k datovým strukturám je záležitost značně ošemetná. I v našich jednoduchých případech bylo docela obtížné rozmyslet si, že si procesory nemohou navzájem škodit a že nemůže nastat deadlock ani vyhladovění.

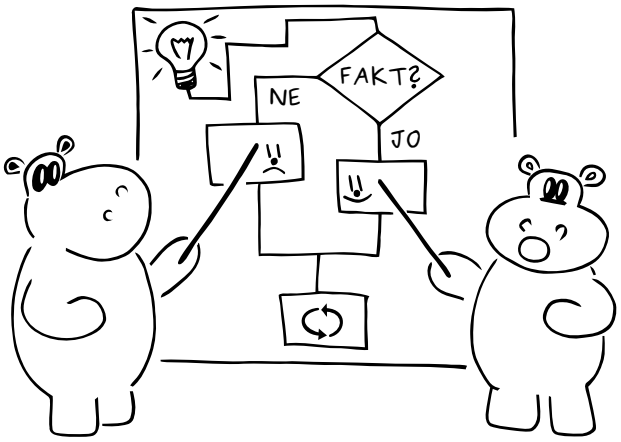
Proto si při praktickém paralelním programování obvykle vytvoříme knihovnu pro základní paralelní datové struktury (různé fronty, slovníky a podobně), detaily zamykání necháme schované uvnitř implementace knihovny, a zbytek programu bezpečně stavíme z hotových kostiček.

I tento přístup má své mouchy: jak jsme viděli ve 2. úkolu, i když máme jednotlivé atomické seznamy, neumíme snadno zařídit atomický přesun z jednoho seznamu do druhého. Naše řešení muselo zasáhnout dovnitř implementace seznamu a mluvit do toho, v jakém pořadí se zamyká.

Někdy se proto může hodit zacházet s pamětí abstraktněji a výpočet rozdělit do *transakcí.* V každé transakci si vedeme žurnál (log) všech přístupů do sdílené paměti. Kdykoliv z paměti přečteme nějakou hodnotu, zapíšeme do žurnálu, odkud jsme četli a co jsme dostali. Kdykoliv chceme zapisovat, tak to neprovedeme přímo, ale pouze zapíšeme do žurnálu, kam chceme co zapsat. (Při čtení tedy musíme také kontrolovat žurnál.)

Na konci transakce provedeme *commit.* Ten atomicky zkontroluje, že přečtené hodnoty ve sdílené paměti mezitím nikdo jiný nezměnil, a zapíše do sdílené paměti to, co jsme změnili my. V praxi se to může realizovat například společným zámkem pro commity transakcí. Každá transakce se tedy tváří, jako by se provedla atomicky. Musíme ovšem počítat s tím, že commit může selhat, a tím pádem je potřeba celou transakci zopakovat. Je to tedy takové zobecnění mechanismu `LDREX`/`STREX` na libovolně mnoho přístupů do paměti.

Martin „Medvěd“ Mareš



Vzorová řešení páté série třicátého ročníku KSP

30-5-1 Úklid po soustředku

Máme najít cestu v bludišti, tak bychom na to mohli jít nějakým prohledáváním grafu. Mohli bychom třeba zkusit všechny velikosti vozíku a vzít tu největší, se kterou jde bludiště projít. Na to ale budeme určitě potřebovat zjišťovat, jestli se nám vozík vejde na konkrétní políčko. Kdybychom to dělali naivně – pokaždé kontrolovali, jestli jsou volná všechna políčka, na kterých je vozík – tak by nám to nepříjemně zhoršilo časovou složitost. Vozík může být řádově stejně velký jako celá mapa, takže by jedno zkontrolování stálo $\mathcal{O}(M^2)$ času.

Mnohem lepší bude si pro všechna políčka předpočítat, jak velký vozík se tam vejde (jak velký může být vozík s levým horním rohem v daném políčku). Půjdeme postupně od spodní strany zprava. Spodní řádek mapy je jednoduchý: tam, kde je políčko volné, bude maximální velikost vozíku 1, jinak to bude 0. V dalším řádku se vždy podíváme jak velké vozíky se vejdou na políčko o jedna dolů, na políčko o jedna vpravo a na políčko o jedna vpravo dolů. Vezmeme si velikost vozíku, který se vejde na všechna z nich, to bude minimum z těch třech čísel. Na aktuální políčko se nám tak akorát vejde vozík o jedna větší. Pro každé políčko se tedy podíváme na políčko vpravo, vpravo-dole a na políčko dole, vezmeme z nich minimum, přičteme jedničku a zapíšeme. Celou mapu takto projdeme v čase $\mathcal{O}(MN)$, kde M a N jsou její rozměry. Při dotazu na velikost se stačí v konstantním čas jednoduše podívat na jedno políčko.

Kdybychom teď chtěli najít nejkratší cestu pro konkrétní velikost vozíku, stačí použít běžné prohledávání do šířky, přičemž políčko budeme považovat za průchozí, pokud se

na něj vozík vejde (podle toho, co jsme si předpočítali). To stihneme v lineárním čase vzhledem k počtu políček. Navíc funguje, že když projdeme s nějak velkým vozíkem, tak to půjde i se všemi menšími, a tak můžeme na maximální možnou velikost vozíku použít binární vyhledávání. Když si označíme A velikost vozíku, která se vejde na začátek, budeme půlit interval $[0, A]$ a celé to bude potřebovat $\mathcal{O}(MN \log A)$ času a $\mathcal{O}(MN)$ paměti.

Sice už to nezrychlíme víc než o logaritmus A, ale proč to neudělat, když můžeme. V principu začneme prohledávat graf do šířky s největším vozíkem, který se vejde na začátek, ale políčka, na která se nevejdeme, nebudeme úplně zahazovat. Místo toho si pro ně na začátku vyrobíme A front, jednu pro každou velikost, a budeme používat tu pro největší vozík. Když narazíme na políčko, jehož předpočítaná velikost vozíku je menší, přidáme políčko do fronty, která odpovídá této menší velikosti vozíku. Pokud se dostaneme do cíle (nebo se spíš cíl dostane pod vozík), můžeme prohlásit, že cesta s daným vozíkem určitě existuje, a skončit.

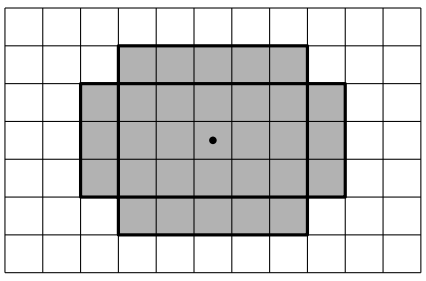
Když nám fronta dojde, smíříme se s tím, že s takto velkým vozíkem cesta do skladu nevede, a zkusíme vozík o jedna menší. Prázdnou frontu zahodíme a začneme používat tu určenou pro o jedna menší vozík. Tímto jsme plynule přešli na prohledávání s menším vozíkem. Všude, kam se dalo dostat s původním vozíkem, se dostaneme i s menším, takže můžeme všechno nalezené použít znovu. V průběhu celého algoritmu tento přechod provedeme maximálně A-krát, a každý nás bude stát jen konstantně času. Celkově každé políčko vytáhneme z fronty jen jednou a pokaždé na něm uděláme jen konstantně práce (projdeme 4 okolní), takže celý algoritmus doběhne v čase $\mathcal{O}(MN)$.

Drobný problém je, že výše uvedený postup nám nemusí dát nejkratší cestu, jenom nám řekne, pro jak velký vozík ještě cesta existuje. To ale snadno obejdeme tak, že pro ten největší možný vozík spustíme běžné prohledání do šířky popsané výše. To také doběhne v lineárním čase a složitost nám tak nezhorší.

Standa Lukeš

30-5-2 Útěk z trezorů

Na jaké políčko chceme položit bombu? Především musí být dosažitelné ze startu. Současně také oblast, kterou bomba vyboural, buďto musí obsahovat políčko dosažitelné z cíle, nebo aspoň s takovým políčkem musí sousedit. Nějaké políčko dosažitelné z cíle tedy musí být v „obdélníku s ušima“ okolo políčka s bombou:



Zbytek je už technické cvičení na základní algoritmy. Dvojím prohledáním do šířky zjistíme, která políčka jsou dosažitelná ze startu a která z cíle. Pak si předpočítáme dvojrozměrné prefixové součty,[8] pomocí nichž půjde v konstantním čase zjistit, kolik políček dosažitelných z cíle leží v daném obdélníku. A nakonec si uvědomíme, že každý ušatý obdélník je sjednocením pěti disjunktních obdélníků bez uší.

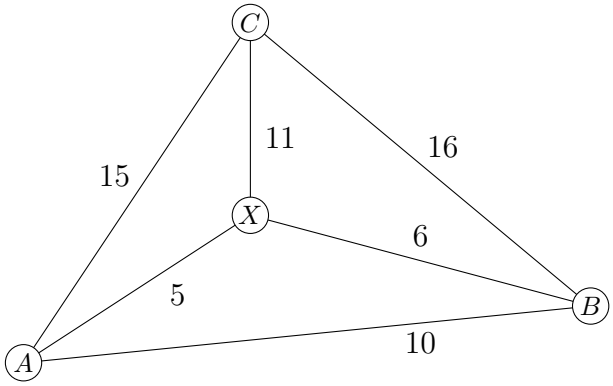
Celkově tedy strávíme čas $\mathcal{O}(RS)$ předvýpočty a pak pro každé z RS políček v konstantním čase rozhodneme, zda je užitečné umístit tam bombu. Časová složitost algoritmu tedy činí $\mathcal{O}(RS)$.

Program (C):
`http://ksp.mff.cuni.cz/viz/30-5-2.c`

Martin „Medvěd“ Mareš

30-5-3 Energetické úspory

Většina z vás úlohu řešila tak, že si našla nejkratší cestu mezi dvěma ze zadaných křižovatek a poté se k ní pokoušela připojit třetí křižovatku. Řešení by to bylo pěkné, nebýt chybného předpokladu, že hledaná podmnožina hran musí obsahovat nejkratší cestu mezi některými význačnými vrcholy. Jeden protipříklad můžete najít na obrázku níže: nejkratší cesty mezi význačnými vrcholy A, B a C jsou strany trojúhelníku ABC, ale nejmenší podmnožina hran, po kterých se dá dojít do všech tří význačných vrcholů, obsahuje hrany AX, BX a CX.



Zajímavé je pozorování, že ačkoliv nám řešení pomocí dvou nejkratších cest nedá vždy nejmenší množinu hran, nemůže být o mnoho horší než optimum. Přesněji, součet délek dvou nejkratších cest mezi vrcholy A, B a C je maximálně o třetinu delší než nejmenší podmnožina hran. Plyne to z toho, že cesty přes X jsou vždy stejně dlouhé nebo delší než nejkratší cesty. Dostaneme tři nerovnosti:

$$|AX| + |XB| \geq |AB|,$$
$$|BX| + |XC| \geq |BC|,$$
$$|CX| + |XA| \geq |CA|.$$

Níže dokážeme, že hledaná nejmenší podmnožina je tvořena cestami z každé význačné křižovatky do X. Sečtením těchto tří nerovností tedy dostaneme dvojnásobek délky nejmenší podmnožiny:

$$2 \cdot (|AX| + |BX| + |CX|) \geq |AB| + |BC| + |CA|.$$

Nyní vybereme nejdelší ze tří nejkratších cest (na obrázku BC) a vyjádříme ji pomocí zbylých dvou:

$$|BC| \geq \frac{|AB| + |CA|}{2}.$$

Obě nerovnosti zkombinujeme a získáme nerovnost

$$\begin{aligned} &2 \cdot (|AX| + |BX| + |CX|) \geq |AB| + |BC| + |CA| \geq \\ &\geq |AB| + |CA| + (|AB| + |CA|)/2 = \\ &= 3/2 \cdot (|AB| + |CA|), \end{aligned}$$

[8] `http://ksp.mff.cuni.cz/viz/kucharky/zakladni`

ze které vidíme, že součet dvou nejkratších cest je nejvýše 4/3 délky nejmenší podmnožiny hran. Z tohoto důvodu jsem tedy i za toto řešení udělovala nějaké body.

Správné řešení

Jak jsem již naznačila, musíme najít nějaký bod X, ze kterého vedou „krátké“ cesty do A, B i C. Takový bod určitě existuje. V hledané nejmenší množině hran totiž musí být cesta (ne nutně nejkratší) mezi body A a B. Zároveň se musíme umět dostat i do bodu C, z bodu C tedy musí vést nějaká cesta, která se napojuje na cestu mezi A a B. Bod napojení označíme jako X (může být shodný i s některým z bodů A, B nebo C). Protože hledáme nejmenší podmnožinu, musíme najít takové X, které bude mít součet vzdáleností k A, B a C nejmenší. Jak to udělat? Mohli bychom z každého bodu grafu spustit prohledávání a spočítat součet těchto vzdáleností, ale to by bylo pomalé. Místo toho na to půjdeme obráceně: spočítáme nejkratší vzdálenosti z bodů A, B a C do všech ostatních bodů grafu.

Spustíme postupně Dijkstrův algoritmus z A, B i C a u každého bodu si zapamatujeme všechny tři vzdálenosti. Poté projdeme celý graf ještě jednou a najdeme bod, který bude mít součet těchto vzdáleností nejmenší. Hledaná podmnožina hran jsou pak hrany tvořící nejkratší cestu z nalezeného bodu X do A, B a C. Tyto nejkratší cesty můžeme najít třeba tak, když si u každého bodu zapamatujeme kromě vzdálenosti i bod, ze kterého jsme přišli. Poté cestu budeme umět zrekonstruovat.

Dijkstrův algoritmus spustíme pouze třikrát, výsledná časová složitost tedy bude stejná jako složitost tohoto algoritmu: $\mathcal{O}((M+N)\log N)$. Paměťová složitost bude $\mathcal{O}(M+N)$, protože nám stačí pamatovat si graf a konstantně mnoho informací u každého vrcholu.

Zuzka Urbanová

30-5-4 Kartotéka

Na vstupu jste dostali dva spojové seznamy (tak se říká struktuře ze zadání) a měli jste za úkol nalézt, kde „srůstají“. Jenže tentokrát jste měli k dispozici jen konstantní množství paměti a bylo tedy potřeba nad problémem přemýšlet z trochu jiného úhlu.

Jako vždy, pokud nevíme, jak začít, tak se vyplatí základní řešení – vyzkoušet všechny možnosti. Tedy postupně půjdeme prvek po prvku v prvním seznamu a vždy se zeptáme, zda se vyskytuje i ve druhém. První takový je místem srůstu. A jak zjistit, že se prvek vyskytuje v druhém seznamu? Nejjednodušší způsob je projít druhý seznam prvek po prvku a každý s ním porovnat. Tedy za každý prvek v prvním seznamu projdeme celý druhý seznam a dostaneme tak časovou složitost $\mathcal{O}(NM)$ (kde N a M jsou délky seznamů). Do konstantní paměťové složitosti jsme se jistě vešli.

Pojďme toto řešení vylepšit. Často, když hledáme nějaké místo v posloupnosti, tak se nabízí využít binární vyhledávání.[9] Tedy místo abychom se ptali postupně každého prvku prvního seznamu jestli je i ve druhém, tak se nejprve zeptáme na tuto otázku prostředního prvku. Všimněte si, že není problém prostřední prvek najít. Stačí spočítat délku seznamu (tak, že jej projdeme a přičteme jedničku za každý prvek) a pak tuto délku vydělíme dvěma. Výsledkem je počet prvků, které musíme projít od začátku seznamu abychom se dostali do jeho středu.

Jak nám odpověď pomůže? Víme, že pokud se tento prvek nachází i ve druhém seznamu, tak „místo srůstu“ je nejpozději v tomto prvku. Zbylé prvky seznamu tedy jistě můžeme zahodit. Naopak pokud tento prvek ve druhém seznamu není, tak „srůst“ nastává až za tímto prvkem. Tedy můžeme naopak zahodit všechny prvky z prvního seznamu až do tohoto prvku. Tak jako tak se vždy zbavíme poloviny prvků v seznamu. Pak už jen opakujeme celou tuto proceduru jen místo celého prvního seznamu budeme uvažovat jeho zkrácenou verzi. Opakujeme tolikrát, dokud nám z prvního seznamu nezbude jediný prvek, tento prvek je jistě místem „srůstu“.

Jelikož vždy půlíme velikost seznamu, celkem opakujeme $\log N$ kroků. Pro každý krok musíme stále projít celý druhý seznam, dostáváme tedy časovou složitost $\mathcal{O}(M \log N)$.

Všimněte si, že v každém kroku půlíme jeden seznam. Nic nás ale nenutí vždy vybírat ten stejný. Intuice nám tedy velí vybrat vždy ten seznam, který je aktuálně (po zahození prvků) větší, protože se tím zbavíme většího počtu prvků.

Pomůže nám to ale? Ukážeme, že ano. Označme K součet prvků v obou seznamech. V prvním kroku musíme projít všech těchto K prvků. Spočítejme ale, kolik z nich zahodíme. Delší seznam má délku alespoň $K/2$ a polovinu z něj zahodíme, tedy alespoň $K/4$. Jinými slovy nám pro další krok zbude $3/4K$ prvků. Pro následující krok pak tři čtvrtiny z těchto tří čtvrtin, tedy $(3/4)^2K$. A tak dále, pokaždé zmenšíme počet prvků na tři čtvrtiny (nebo méně).

V každém kroku musíme jen projít všechny nezahozené prvky, takže časová složitost bude nanejvýš:

$$\mathcal{O}\left(\sum_{i=0}(3/4)^i K\right) = \mathcal{O}(4K) = \mathcal{O}(K)$$

Řešení s nápadem

Uff... Dostali jsme lineární řešení, a pokud jste zběhlí v programátorských praktikách a základech matematické analýzy, tak vám možná přišlo i příjemně přímočaré. Pro ty ostatní z vás, kterých je (podle došlých řešení) většina, ale nabízíme jiné řešení. Toto řešení sice vyžaduje nejprve na problém „koukat“, ale odměnou je elegantnější a stále optimální řešení.

Představme si, že bychom u každého prvku v obou seznamech znali jeho vzdálenost k profesorovi. Všimněte si, že pro prvky, které jsou v obou seznamech je tato vzdálenost stejná, bez ohledu na to, z pohledu kterého seznamu se na něj díváme. To nám dává jednoduchý nástroj, jak poznat, zda je prvek v obou seznamech. Stačí se u daného prvku podívat na jeho *kamaráda* z druhého seznamu, který má stejnou vzdálenost k profesorovi. Prvky, které jsou jen v jednou seznamu, mají nějakého jiného kamaráda v druhém seznamu. Naopak prvky, které jsou v obou seznamech se kamarádí... sami se sebou.

Jenže na pamatování si těchto čísel nemáme paměť. Všimněme si ale, že pokud známe nějakou dvojici kamarádů, tak snadno poznáme kamarády, všech jejich následníků (jsou-li A a B kamarádi, následník A se kamarádí s následníkem B). Stačí tedy najít první dvojici kamarádů (první prvky delšího seznamu jsou zcela bez kamarádů).

Jak tuto dvojici najdeme? Nejprve si spočítáme délku obou seznamů (N, M). Předpokládejme, že řetězec délky N je ten kratší. U delšího řetězce pak musíme ze začátku zahodit

[9] `http://ksp.mff.cuni.cz/viz/kucharky/binarni-vyhledavani`

tolik prvků o kolik je tento řetězec delší. Tedy formálněji ze začátku delšího řetězce se posuneme o $M - N$ prvků níže. Vzdálenost prvku, který dostaneme, k profesorovi je pak $M-(M-N) = N$. Takže se kamarádí se začátkem druhého seznamu. Teď stačí porovnat tyto dva prvky, pokud jsou různé, tak se přesuneme na jejich následníky (kteří jsou také kamarádi), zkontrolujeme, zda se nerovnají oni atd. Toto opakujeme, dokud se nedostaneme na první prvek, jenž se kamarádí sám se sebou. Tento prvek je pak místem srůstu.

Jelikož jsme každý prvek prošli jen dvakrát, tak časová složitost je opět lineární. Pamatujeme si vždy jen pár prvků a několik hodnot, takže jsme se vešli i do konstantní paměti.

Velikost konstantní paměti

Možná máte pocit, že jsme v řešeních trochu podváděli. Pamatovali jsme si vždy jen pár čísel, ale ne jen ledajakých čísel. Konkrétně jsme si potřebovali pamatovat velikost řetězců, což je ale číslo závislé na vstupu, takže by se do konstantní paměti vejít nemělo!

V řešení jsme jen zmínili, že máte konstantní množství pomocné paměti a dále to neupřesnili, za což se omlouváme. Obvykle konstantní množství paměti chápeme v KSP jako konstantní množství buněk, kde každá buňka je schopná uchovat číslo ze vstupu.

Proč je takováto představa rozumná? Zejména pokud u druhé varianty řešení se můžete šikovnou manipulací s ukazateli zcela vyhnout potřebě pamatovat si takové číslo? Všimněte si, že v řešení zcela automaticky pracujeme s ukazateli na karty (a uchováváme si je). Ukazatel musí být pro každou kartu zcela jiný. Máme-li K karet, musí každý ukazatel zabírat alespoň $\log K$ bitů paměti. A v této paměti si už pohodlně můžeme uchovávat i velikosti řetězců. Kdybychom měli skutečně jen pár bitů, nebyli bychom schopní udělat vůbec nic, jelikož bychom si ani nemohli pamatovat kartu, se kterou pracujeme.

Dominik Smrž

30-5-5 Výroba likvidátoru

Pro stručnost budeme v textu řešení pro největšího společného dělitele (NSD) čísel a a b používat (vymyšlené) značení $a \diamond b$. To, že tato notace připomíná třeba značení operace sčítání nebo násobení, není náhoda – ve skutečnosti se na $\diamond$ můžeme dívat taky jako na nějakou operaci, která vezme dvě čísla a nějak je „zkombinuje“ do nového.

Vzoreček z kuchařky pro NSD více čísel $a_1, \ldots, a_n$ v této notaci můžeme napsat jako $a_1 \diamond (a_2 \diamond (\ldots (a_{n-1} \diamond a_n)) \ldots)$. Když si uvědomíme, jak funguje školní algoritmus na počítání NSD pomocí prvočíselného rozkladu, zjistíme, že na závorkách, a tedy i pořadí vyhodnocování operací, nezáleží. Můžeme tedy psát jen $a_1 \diamond a_2 \diamond \cdots \diamond a_n$. Této nezávislosti operace na pořadí vyhodnocování se obecně říká *asociativita.*

Můžeme tedy úlohu vyřešit přímočaře: vyzkoušíme všechny možnosti, jak vynechat právě jedno číslo, a vybereme si tu, která bude dávat největší NSD. Jedno spočítání NSD $n - 1$ čísel nás bude stát čas $\mathcal{O}(n \log d)$, kde d je hodnota největšího čísla, a celkem jich provedeme $\mathcal{O}(n)$, tedy celková časová složitost je $\mathcal{O}(n^2 \log d)$.

Intervalové stromy

Jde to ovšem i rychleji, pokud nějak počítání NSD zrychlíme. Pomůže nám právě asociativita: když máme nějaká čísla $a_1, \ldots, a_k$ a jejich NSD n_1 a čísla $b_1, \ldots, b_\ell$ a jejich NSD n_2, pak $a_1 \diamond \ldots \diamond a_k \diamond b_1 \diamond \ldots \diamond b_\ell = (a_1 \diamond \ldots \diamond a_k) \diamond (b_1 \diamond \ldots \diamond b_\ell) = n_1 \diamond n_2$. Jinými slovy, dva největší společné dělitele nějakých dvou skupinek čísel umíme v čase $\mathcal{O}(\log d)$ zkombinovat do jednoho NSD obou skupinek.

Všechny tyto vlastnosti nápadně připomínají to, co potkáme u obyčejného sčítání nebo násobení. Dává tedy smysl, že s operací $\diamond$ fungují i třeba takové intervalové stromy (o nichž si více můžete přečíst v kuchařce).[10] Nejprve čísla na vstupu v nějakém pořadí uložíme do pole jako $a_1, \ldots, a_n$. Následně nad polem vytvoříme intervalový strom, který bude fungovat úplně stejně jako součtový nebo minimový intervalový strom, jen bude všude používat operaci $\diamond$.

S takovým intervalovým stromem umíme jedno škrtnuté číslo vyzkoušet v $\mathcal{O}(\log n \log d)$ času. Zeptáme se totiž intervalového stromu na NSD dvou intervalů: od začátku do škrtnutého prvku a od škrtnutého prvku dále. Každý z obou dotazů pak spočteme v čase $\mathcal{O}(\log n \log d)$, protože počítáme NSD $\mathcal{O}(\log n)$ hodnot ve vrcholech intervalového stromu a každá hodnota je číslo velké nejvýše $\mathcal{O}(d)$. Na vybudování intervalového stromu potřebujeme čas $\mathcal{O}(n \log d)$, takže celková časová složitost je $\mathcal{O}(n \log n \log d)$.

Optimální řešení

I toho logaritmu se jde zbavit, pokud místo intervalových stromů použijeme nějakou jednodušší strukturu. Inspirujeme se starými dobrými prefixovými součty a vybudujeme si v $\mathcal{O}(n \log d)$ pole p prefixových NSD, kde $p[i] = a_1 \diamond a_2 \diamond a_3 \diamond \cdots \diamond a_i$. To samotné nám ale stačit nebude, protože narozdíl od sčítání neumíme od NSD „odečítat“. Proto si pořídíme ještě pole s suffixových NSD, kde $s[i] = a_i \diamond \cdots \diamond a_n$. Snadno se nahlédne, že NSD všech čísel kromě j-tého se spočte jako $p[j-1] \diamond s[j+1]$. Řešení pak běží v čase $\mathcal{O}(n \log d)$ a paměti $\mathcal{O}(n)$.

Poznámky k došlým řešením

- Někteří z vás chybně psali, že čas na spočítání jednoho NSD je $\mathcal{O}(\log n)$, kde n je počet čísel na vstupu. Je to sice jen drobný prohřešek, ale přesto je dobré dávat pozor na to, které proměnné ve vašem řešení co znamenají.
- Jak se ukázalo, tato úloha sváděla k mnohým nefunkčním heuristikám typu „vezmi několik nejmenších čísel, a jedno z nich buď škrtni, nebo prohlaš za GCD celé řady“. Jakmile vaše řešení obsahuje rozbor mnoha různých případů, často je to znamení, že byste se měli zamyslet, zda pokrýváte všechny případy a zda je řešení každého případu správně. Často také pomáhá zkusit si vymyslet protipříklad, na kterém řešení selže (a třeba zjistit, že žádný protipříklad neexistuje).

30-5-6 Nápověda na lyžích

V úloze jsme měli za úkol najít největší možnou šířku lyžaře – neboli nejužší místo mezi dvěma množinami tyček, kterým musí lyžař projet. Uvědomíme si, že lyžař musí projet napravo od konvexního obalu všech západních tyček (a nalevo od konvexního obalu tyček východních), protože jede po přímce. Kdyby takto vjel dovnitř konvexního obalu, jistě

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

[11] `http://ksp.mff.cuni.cz/viz/kucharky/geometrie`

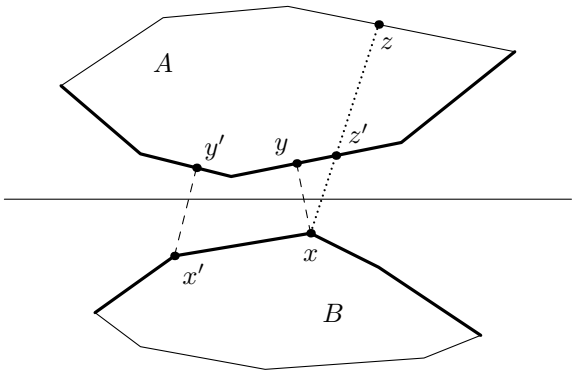
by projel kolem některé tyčky ve špatném směru. Jak konvexní obal sestrojit, si můžete přečíst v naší geometrické kuchařce.[11]

Hledáme tudíž nejkratší úsečku mezi dvěma konvexními mnohoúhelníky. Označíme si je A a B. Uvědomíme si, že takováto úsečka určitě bude mít jeden krajní bod v některém z vrcholů obou mnohoúhelníků. Pokud bychom totiž našli nejkratší úsečku s oběma krajními body na stranách mnohoúhelníků, určitě bychom dovedli najít stejně dlouhou úsečku s jedním krajním bodem ve vrcholu. Příslušná dvojice stran by totiž musela být rovnoběžná (jinak by existovala bližší dvojice bodů), takže by stačilo posunout nejkratší úsečku ve směru kolmém k oběma stranám do nejbližšího vrcholu.

Jednoduché řešení se tedy hned nabízí: v čase $\mathcal{O}(N^2)$ vyzkoušíme každý vrchol z A zkombinovat se všemi vrcholy a stranami z B a vybereme nejkratší vzdálenost.

Jde to ale lépe. V předchozím řešení pro každý vrchol procházíme celý mnohoúhelník B. To však není nutné: pomůžeme si tím, že si rozmyslíme, kterým směrem se máme k místu (vrcholu nebo hraně druhého mnohoúhelníku) s nejkratší vzdáleností z daného vrcholu vydat. Nejprve si zvolíme počáteční vrchol v A a najdeme z něj nejkratší vzdálenost k B jako v předchozím případě vyzkoušením všech možností. Pokračujeme po hranici A vrcholem sousedícím s počátečním. Nejprve vyzkoušíme tři vzdálenosti: do místa s nejkratší vzdáleností z předchozího vrcholu a do dvou sousedních míst. Vzdálenosti porovnáme a vydáme se po vrcholech a stranách ve směru klesání vzdáleností, dokud vzdálenost opět nezačne růst. Takto postupně pro všechny vrcholy z A najdeme nejbližší místa v B.

Rozmyslíme si, že tento algoritmus opravdu najde nejkratší vzdálenost. Uvažme přímku oddělující oba mnohoúhelníky (z řešení úlohy 30-3-4 víme, že existuje) a celou situaci si představme otočenou tak, aby tato přímka ležela vodorovně. Nyní si z přímky „posvítíme“ na oba mnohoúhelníky, oba se rozdělí na „osvětlenou“ a „tmavou“ část. Nejbližší dvojice bodů určitě leží v osvětlených částech.

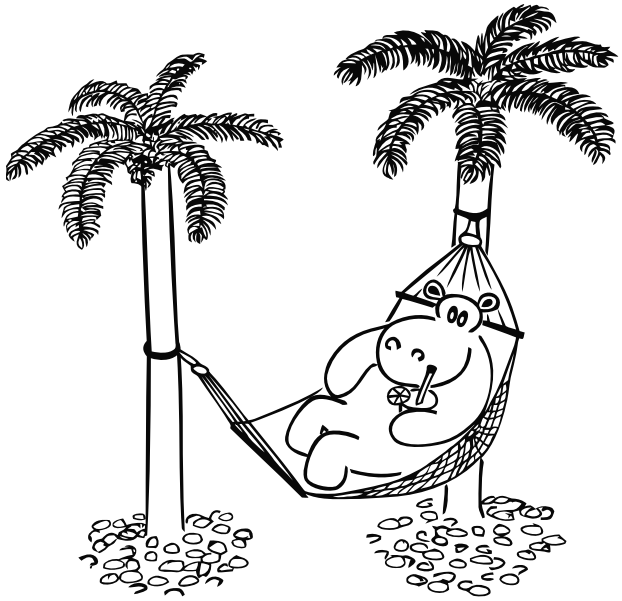


Nyní uvažme libovolný vrchol $x \in A$. K němu nejbližší místo $y \in B$ leží v osvětlené části B (ke každému tmavému místo $z \in B$ existuje bližší osvětlené místo $z' \in B$, totiž průsečík polopřímky xz s osvětlenou částí). Vzdálenosti k ostatním osvětleným místům v B směrem od y na obě strany rostou (přesněji neklesají: y může ležet na straně kolmé na úsečku xy; od této strany dál ale už vzdálenost ostře roste). Pokud jsme tedy k předchozímu vrcholu $x' \in A$ znali nejbližší místo $y' \in B$, stačí se od y' pohybovat po B ve směru klesající vzdálenosti k x, abychom našli správný bod y.

Algoritmus tedy funguje. Jaká je jeho časová složitost? Pokud projdeme osvětlenou část A zleva doprava, projdeme současně osvětlenou část B zleva doprava. Když procházíme tmavou část A, projdeme přitom také právě jednou osvětlenou část B, a to v opačném směru. Každý bod tedy celkem navštívíme $\mathcal{O}(1)$-krát. Jelikož spočítat vzdálenost mezi dvěma body nebo bodem a úsečkou zvládneme v konstantním čase, algoritmus běží v lineárním čase.

Nezapomínejme ovšem, že jsme nejprve museli najít konvexní obaly obou množin bodů. To podle geometrické kuchařky trvá $\mathcal{O}(N \log N)$, což je také celková složitost našeho řešení.

Martin Mareš & Zuzka Urbanová



KSP pro vás připravují studenti Matematicko-fyzikální fakulty Univerzity Karlovy.

Webové stránky: *https://ksp.mff.cuni.cz/*
E-mail: *ksp@mff.cuni.cz*
Diskusní fórum: *https://ksp.mff.cuni.cz/forum/*

Chcete-li s námi komunikovat bezpečně, můžete si ověřit náš HTTPS certifikát – jeho SHA1 fingerprint je: `E9:DB:EE:C6:62:BC:14:DE:09:E4:E8:97:DC:36:0E:87:B3:50:B0:01`.

Závěrečná výsledková listina 30. ročníku KSP

	řešitel	*škola*	*ročník*	*sérií*	*5-1*	*5-2*	*5-3*	*5-4*	*5-5*	*5-6*	*série*	*celkem*
0.					11	11	11	10	11	13	57,0	300,0
1.	Jiří Škrobánek	G Wicht	4	5	10		11	10	11	13	55,6	248,3
2.	Vladimír Chudý	G Chrudim	1	5	4	8	11	10	6	5	46,1	202,2
3.	Michal Kodad	SPŠSmíchov	2	13		11	4	10			24,4	198,1
4.	Klára Tauchmanová	GOhradníPH	4	6	8	11	4	10	6	12	50,6	193,4
5.	Jan Černý	BiGy Žďár	2	5	4	11	3	10	1	5	39,5	193,1
6.	Josef Minařík	GJarošeBO	3	3							0,0	159,7
7.	Veronika Nechaieva	KyivLyceum	4	4							0,0	157,8
8.	Matěj Kripner	GEBenešeKL	3	5	9	11		10	9,5	9	52,3	151,8
9.	Daniel Skýpala	GTomkovaOL	0	8	5	11		10		1,5	29,2	147,7
10.	Jakub Komárek	GUHradiště	3	5		11		10	6		29,1	146,7
11.	Petr Zahradník	GaSOŠ ÚL	3	3							0,0	131,0
12.	Jan Kaifer	GKepleraPH	2	10							0,0	130,0
13.	Václav Pavlíček	SPSEPard	2	12	6		2	10			17,5	110,9
14.	Jiří Kvapil	GTomkovaOL	0	5	5	11	4	10	1	1	36,2	108,7
15.	Jáchym Mierva	BiGy Žďár	1	3							0,0	98,7
16.	Jakub Pelc	G UherBrod	4	15		11					11,0	78,4
17.	Tomáš Černý	GArabskáPH	2	2							0,0	77,9
18.	Miroslav Hrabal	GTomkovaOL	4	8		11	11	10			32,0	73,8
19.	David Klement	GNAlejíPH	2	3	11	11	11	10	1	12	55,7	73,2
20.	Michal Zaslavský	GKepleraPH	3	4		11					11,0	71,1
21.	Ondřej Gonzor	G Brandýs	1	9			3	4	6		15,1	70,1
22.–23.	Ondřej Bleha	GBNěmcovHK	3	2							0,0	68,8
	Zuzana Urbanová	GFXŠaldyLI	4	5			11	7			19,5	68,8
24.	Martin Kurečka	GJarošeBO	4	5							0,0	66,5
25.	Adam Dejl	G JGJ PH	4	3							0,0	63,2
26.	František Kmječ	G Brandýs	2	8							0,0	58,8
27.	Pavel Hudec	G JGJ PH	4	1							0,0	57,5
28.	Vojtěch Žák	GŠpitálsPH	2	1	8	8	11	10	1	6	51,1	51,1
29.	Lucie Vomelová	GŠpitálsPH	2	1	5	8	11	10	1	6	49,3	49,3
30.	Marie Kalousková	GNAlejíPH	2	1	8	2	11	7	3,5	9	48,8	48,8
31.	Filip Geib	G MMH LM	4	6							0,0	41,3
32.	Jan Piroutek	GŠpitálsPH	2	1	5	8	1	10	1	6	40,9	40,9
33.	Tomáš Sládek	GJHroncaBA	2	3							0,0	40,2
34.	Vít Skalický	GPísnickáPH	0	4		11					11,0	36,1
35.	Martin Zimen	GJMasarJI	3	1							0,0	33,9
36.	Jiří Löffelmann	GLitoměřPH	4	9							0,0	31,5
37.	Vojtěch Michal	GNVPláníPH	3	2							0,0	29,3
38.	Anička Mlezivová	GCoubTábor	4	1			11	10	4		28,4	28,4
39.	Tomáš Strnad	GŽamberk	4	1							0,0	23,0
40.	Lucia Krajčoviechová	GJHroncaBA	2	1							0,0	19,8
41.	Karel Balej	GRokycany	3	4							0,0	18,0
42.	Ondřej Wrzecionko	GTěš	3	1							0,0	15,0
43.	Filip Masár	PiarGNitra	4	3							0,0	14,7
44.	Eliška Vlčinská	GHladnov	3	3							0,0	14,6
45.	Jakub Růžička	GNymburk	3	1							0,0	13,4
46.	Jakub Jirkal	GJungmanLT	3	2							0,0	13,2
47.	Michal Tomek	GHumpolec	4	1							0,0	12,3
48.	Vojtěch Káně	G Brandýs	2	2	8						10,0	11,3
49.–50.	Filip Hejsek	GPísnickáPH	1	1		11					11,0	11,0
	Radek Olšák	MensaG	3	2			11				11,0	11,0
51.	Vojtěch Zabořil	GTurnov	1	1							0,0	9,7
52.	Andrej Ohrablo	GJHroncaBA	2	1							0,0	9,1
53.	Václav Zvoníček	GJarošeBO	2	1							0,0	7,5
54.	Prokop Randáček	GFXŠaldyLI	−1	1							0,0	7,4
55.	Viktor Fukala	GKepleraPH	1	4							0,0	6,8
56.	Štěpán Zapadlo	GJŠkodyPŘ	2	2							0,0	6,0
57.–59.	Kristina Galikova	ŠPMNDaGB	3	1							0,0	5,7
	Jaroslav Paška	ŠPMNDaGB	3	1							0,0	5,7
	Adam Verner	SPŠ Prosek	3	1							0,0	5,7
60.	Lenka Vincenová	GTomkovaOL	4	1							0,0	4,0
61.	Lukáš Caha	GZborovPH	4	5							0,0	3,8
62.	Jakub Ucháč	ŠMaVVzt	2	1							0,0	2,7
63.	Vojtěch Březina	GCoubTábor	1	1							0,0	1,3