Korespondenční Seminář z Programování

29. ročník **KSP** **Listopad 2016**

Milí řešitelé a řešitelky!

Léto se přehouplo v podzim, venku se každou chvíli válí mlha, a když se roztrhá, ukáže tím, jak se v ulicích válí spadané listí. Přichází podzimní deště, sychravé večery i první kola lecjakých soutěží.

KSPčko má první sérii už za sebou, ale i do druhé se může zapojit kdokoliv, kdo má zájem lámat si hlavu nad úložkami, třeba v teple krbu s hrnečkem horkého čaje. Krb i čaj si musíte zajistit sami, ale úložky vám s radostí dodáme. Račte se začíst.

Také bychom vás rádi pozvali v podvečer 22. listopadu na **Kalíšek** (setkání v čajovně) v Praze a hned následující den na **Den otevřených dveří na Matfyzu**. Více informací se brzy objeví na našich stránkách i na Facebooku.

A pokud stále váháte, připomínáme, že každému řešiteli, který v tomto ročníku z každé série dostane alespoň 5 bodů, darujeme KSP propisku, blok, placku a možná i další překvapení.

Za řešení KSP je také možné být přijat na MFF UK bez přijímacích zkoušek. Na získání osvědčení úspěšného řešitele je letos třeba získat v hlavní kategorii alespoň 150 bodů. Maturanti, kteří by chtěli osvědčení využít letos, jej musí získat nejpozději za čtvrtou sérii.

Termín série: **Pondělí 5. prosince 2016 v 8:00**

Odevzdávání: Přes web na adrese *https://ksp.mff.cuni.cz/submit/*

Odměna série: **Sladkou odměnu** pošleme každému, kdo získá alespoň **polovinu bodů z pěti odevzdaných úloh**.

Druhá série dvacátého devátého ročníku KSP

„To ten den skvěle začíná," prolétlo Erice hlavou, když chvatně přesouvala svůj notebook z kolejního stolu do batohu. Hodiny na zdi navzdory výhružným pohledům ukazovaly patnáct minut do začátku výuky, která ovšem probíhala přes půl hodiny odtud.

Budík ji měl vzbudit už o půl osmé, ale když pak na chvilku zavřela oči, musela usnout, protože najednou bylo skoro devět. Konečně měla vše potřebné a mohla vyběhnout. Napadlo ji, kolik vlastně existuje stejně rychlých cest do školy, ale raději si zakázala experimentovat.

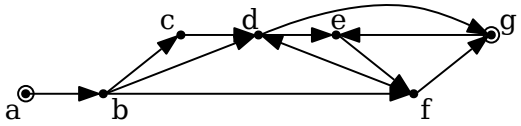
29-2-1 Cesty do školy — 10 bodů

Studentka se chce dostat do školy za právě K minut – ne více, ale ani ne méně (to by musela zbytečně čekat na chodbě). Zajímalo by ji, kolika různými způsoby to jde udělat. Protože venku už začíná být docela zima, nechce se ani po cestě nikde zastavovat. Raději celých K minut stráví chůzí, i kdyby to znamenalo trochu si zajít, nebo třeba jít kus tam a zpátky.

K dispozici má mapu, ve které je zakresleno N význačných míst – kromě startovního a cílového bodu (koleje a školy) také různé křižovatky a další místa, přes která je možné procházet. Dále mapa obsahuje seznam povolených přesunů: každá jeho položka říká, že z místa i jde dojít na jiné místo j, a to za přesně jednu minutu. Jinudy než podle povolených přesunů se pohybovat nelze. Pozor, u přesunů záleží na směru. Může se stát, že je povoleno jít z i do j, ale ne z j do i (třeba protože v opačném směru je to moc do kopce).

Formálněji: je dán orientovaný graf. Jeho vrcholy odpovídají místům a hrany povoleným přesunům. Zajímá nás, kolik v něm existuje různých sledů mezi danými dvěma vrcholy s a t dlouhých přesně K hran. Sled je něco jako cesta, jen se na něm mohou opakovat vrcholy a hrany.

Uvažujme například následující mapu a $K = 6$, $s = a$, $t = g$:



V ní existuje právě pět sledů délky 6 vedoucích z a do g. Jsou to *abcdefg*, *abfgefg*, *abdgefg*, *abdefdg*, *abfdefg*.

Obvyklý způsob dopravy dnes zafungoval. Erice se povedlo chytit autobus, tramvaj (přes hlavy lidí viděla přiskakující minuty) i další tramvaj a krátce před půl už přebíhala Malostranské náměstí, až skoro smetla nějakou podobně starou dívku. Schody vzala po více najednou, přeběhla celou chodbu a zkusila se dobýt do učebny — která se ovšem tvářila tiše, a hlavně zamčeně.

Erika hned vytáhla mobil a na Hangoutu napsala svému spolužákovi: `Ahoj, prosím Tě, analýza se někam přesunula?` *Vzápětí zůstala nevěřícně koukat na čas 8:28 vedle své zprávy. Odpověď přišla záhy.* `Ne ale zacina prece v 9.` *A pak přišla další zpráva:* `Vis ze je zimni cas?`

Jen malým zázrakem nedošlo k násilí na nevinném telefonu. Místo toho se Erika vydala zkoumat nástěnky na chodbách. Na jednu někdo vylepil papír se zašifrovaným textem a výzvou k rozluštění. Spíš než opravdové řešení teď ale Erika toužila najít něco jako „pomsta vynálezci letního času".

29-2-2 Hledání pomsty — 13 bodů

Předpokládejme, že text na nástěnce je zašifrovaný jednoduchou substituční šifrou. Ta funguje tak, že pro každé písmeno abecedy nahradíme všechny jeho výskyty v původním textu nějakým jiným písmenem (např. každé A změníme na X, každé B na U atd.).

Navíc platí, že dvě různá písmena nikdy nenahradíme stejným, protože pak by se text nedal jednoznačně dešifrovat.

Předpisu, které písmenko nahrazujeme kterým, se říká *klíč*.

Korektními způsoby, jak zašifrovat slovo `PAPIR`, jsou např. `UWUXI` nebo `PEPZN`, ale nikoli `SDFGH` (každé P jsme nahradili za něco jiného) nebo naopak `CLCKC` (P i R jsme nahradili stejným písmenem).

Dostanete zašifrovaný text (neznámým klíčem) a hledaný řetězec (nezašifrovaný). Vaším úkolem je najít všechny pozice, na kterých se v původním textu mohl zadaný řetězec vyskytovat. Různé výskyty mohou předpokládat různé klíče.

Například slovo `POTOPA` v textu `ZAGHAHGZGHLQWUW` můžeme najít na dvou místech:

```
POTOPA
ZAGHAHGZGHLQWUW
POTOPA
```

V prvním případě klíč překládá P→G, O→H, T→A, A→Z. Ve druhém je správné přiřazení P→H, O→G, T→Z, A→L.

Snadno si rozmyslíte, že jinde už se toto slovo vyskytovat nemůže.

Po přednášce, která proběhla překvapivě v klidu (jen jeden nejasný důkaz a jen dvě rýpnutí od spolužáků, kteří změnu času zaregistrovali), čekalo Eriku ještě programovací cvičení. Obvykle ho měla ráda, ale dnes se jí povedlo jedním středníkem navíc vyrobit nekonečný cyklus. Přitom na původ chyby přišla až po hodině, takže se dnes ze školy vyloženě těšila.

Zvlášť když si na dnešek naplánovaly sraz s kamarádkou ze střední! Potkat se na zastávce Karlovo náměstí znělo jako skvělý nápad, dokud Erika nezjistila, že těch zastávek je více kus od sebe. Zavolat kamarádce znělo jako skvělý nápad, dokud telefon suše neoznámil, že „volaný účastník není dostupný“.

A tak Erika několikrát přebíhala mezi jednotlivými zastávkami. Přitom si všimla, že tu stále visí nejrůznější volební reklamy.

29-2-3 Billboardová většina — 13 bodů

Erika probíhá ulicemi, kde visí spousta volební reklamy: billboardy, plakáty, ... Víme, v jakém pořadí okolo reklamních materiálů proběhla a které strany propagovaly.

Rádi bychom uměli pro libovolnou část její cesty zjistit, jestli v tomto úseku měla nějaká strana nadpoloviční většinu reklamních materiálů (a tedy by přesvědčila lidi, kteří projdou jen tuto část cesty).

Na vstupu dostanete nejdřív posloupnost N přirozených čísel $(a_0, \ldots, a_{N-1})$. Ta udává, které strany propagují jednotlivé plakáty, v pořadí, v jakém je Erika míjela (tedy a_i je číslo strany propagované i-tou reklamou). Čísla stran můžou být libovolně velká.

Poté bude následovat Q dotazů. Každý dotaz je tvořen dvojicí čísel (k, ℓ). Úkolem vašeho algoritmu je pro každý dotaz určit, zda v úseku $a_k, a_{k+1}, \ldots, a_{\ell-1}$ posloupnosti má nějaké číslo strany nadpoloviční zastoupení.

Například pro posloupnost

a_0	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9
1	4	4	7	4	1	4	1	1	7

a dotazy $(1,6)$, $(3,6)$, $(2,5)$, $(5,10)$, $(0,10)$ jsou správnými odpověďmi $4, -, 4, 1, -$ (kde $-$ značí, že v daném úseku nemá většinu nikdo). První úsek $(1,6)$ je výše znázorněn podtržením.

Vyhodnocovat každý dotaz zvlášť by bylo pomalé. Zkuste si na začátku pro posloupnost něco předpočítat, abyste pak zvládli dotazy vyřizovat rychleji. Předpokládejte, že počet dotazů bude řádově srovnatelný s N.

Při třetím návratu na původní zastávku se ovšem zadařilo a obě dívky se konečně potkaly. V blízké kavárně pak nadšeně propovídaly dvě hodiny jako nic. Při loučení se si slíbily, že se zase brzy uvidí, a pak už každá vyrazila za svým dalším programem.

V Eričině případě to znamenalo vrátit se na kolej a sbalit si vše, co by mohla potřebovat na hodině powerjógy. K jejímu provozování se nechala přesvědčit už na začátku semestru Hankou z koleje, která nechtěla chodit sama. Když Erika dorazila na sraz s Hankou, bylo už dost pozdě. Přesto se Hanka nejvíc ze všeho tvářila zmateně.

„Máš určitě všechno?“ zeptala se nejistě. „No jasně,“ mávla Erika rukou... ve které, jak si právě uvědomila, neměla sportovní tašku. „Eh, ne, počkej, hned jsem zpátky!“

Hanka měla z Eriky náramnou legraci a dobírala si ji i po lekci, kdy se Erika chystala vydat za dalšími kamarády. „Nemám Tě raději doprovodit na místo?“ ptala se. „Huš,“ zkontrolovala Erika na mobilu jízdní řády, „za chvilku mi jede autobus a přestoupit na metro zvládnu.“

Dívky se rozloučily a Erika za chvíli skutečně nastoupila do autobusu. Když ale ani po pěti zastávkách nebyla v dočasném cíli, znejistěla a na další zastávce raději vystoupila. Zaujalo ji náměstíčko protkané chodníčky. Mezi nimi se nacházely květinové záhony podivuhodně nepravidelných tvarů. Sedmiúhelníkový záhon, kdo to kdy viděl!

29-2-4 Nejsložitější záhon — 9 bodů

Na náměstíčku tvaru N-úhelníku jsou různé chodníčky, které jsou ale vedené tak, že se navzájem nekříží a vycházejí jen z pomyslných rohů, nikoliv ze samotných stran.

Oblasti mezi chodníčky tvoří květinové záhony. Nás by zajímalo, který záhon má nejsložitější tvar, tedy je tvořen mnohoúhelníkem s nejvíce stranami.

Toto je praktická open-data úloha. V odevzdávacím systému si necháte vygenerovat vstupy a odevzdáte příslušné výstupy. Záleží jen na vás, jak výstupy vyrobíte.

Formát vstupu: Na prvním řádku vstupu dostanete čísla N a K udávající počet vrcholů N-úhelníka a počet chodníčků. Na dalších K řádcích následuje popis chodníčků – každý chodníček je určen dvěma čísly udávajícími, mezi kterými dvěma vrcholy náměstíčka vede. Vrcholy číslujeme od 0 do $N-1$ v pořadí na obvodu.

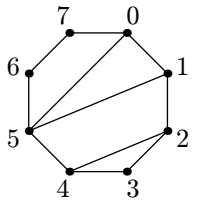
Formát výstupu: Na výstup na prvním řádku počet vrcholů na obvodu největší souvislé oblasti (záhonu) a na druhém řádku mezerou oddělená čísla *zajímavých* vrcholů ohraničujících tuto oblast – to jsou takové vrcholy, kde přecházíme z jednoho chodníčku na druhý. Čísla vypište v rostoucím pořadí, v jakém se vyskytují na obvodu této oblasti. Pokud existuje více takových oblastí, vyberte libovolnou.

Ukázkový vstup:

```
8 3
5 1
0 5
2 4
```

Ukázkový výstup:

```
4
0 5
```



Nejsložitější záhon má tvar čtyřúhelníku – můžeme si vybrat $(1,2,4,5)$ nebo $(0,5,6,7)$. Na prvním jsou významné všechny body, na druhém jen body 0 a 5.

Erika si ovšem záhy vzpomněla, že její hlavní starostí je něco jiného. Pohled na ceduli u zastávky jí prozradil, že zřejmě vystoupila nikoliv z autobusu 136, nýbrž z autobusu 135. Potlačila zanadávání na plus minus jedničky, našla si nový spoj a po další tři čtvrtě hodině úspěšně dorazila do čajovny určení.

„Jéje, Erika nám to určitě pokazí," uvítali ji se smíchem. „Co, co?" „Petr bude příští semestr ve Švédsku, tak už si plánujeme, kdy se kdo z nás pozve na návštěvu," vysvětlili jí kamarádi vesele. A všichni se znovu sklonili nad poznámkami.

29-2-5 Plánování návštěv — 10 bodů

Petr stráví příštích N týdnů ve Švédsku. Každý z jeho K kamarádů by rád přijel na návštěvu. Každý víkend může Petr ubytovat právě jednoho svého kamaráda. Zároveň se kvůli různým již naplánovaným akcím každému z kamarádů hodí právě 2 víkendy.

O každém z kamarádů se dozvíte, které 2 víkendy by se mu pro návštěvu hodily. Rozhodněte, zda se mohou u Petra vystřídat všichni, nebo zda bude muset Petr některé odmítnout.

Můžete předpokládat, že $K \leq N$.

Od vzrušeného dohadování se nad diáři se celá společnost postupně přesunula k mnoha dalším tématům. Ovšem čas nechtěl brát ohled na jejich veselí, ani se nenechal ukolébat klidem zbytku čajovny, poskakoval a postupně přinesl únavu.

Zrovna když část lidí řešila, jak si pomocí obyčejné mince vybrat ze tří čajů, zvedla se Erika k odchodu. Navzdory historkám celého dne vyrazila sama a bezpečně došla zpět na metro. Za jedinou komplikaci by mohla považovat zhasínající lampu, ale už tu párkrát šla a zrovna tahle lampa zhasínala pokaždé, když procházela okolo.

Na Muzeu přestupovala v zamyšlení, proplétala se mezi pár cestujícími. Najednou k ní dolehlo pobavené zavolání: „Tak Markovci udrželi Knot zas jenom dva dny!"

Erika se překvapeně rozhlédla. Z ostatních cestujících tu mezitím zůstal jen nějaký muž a dívka tak v jejím věku, která teď obarvila dvě políčka v tabulce na zdi. Čmárání na zeď v metru? Erika se zarazila. A lehce sebou trhla, když si nad tabulkou všimla nápisu „Linka α".

„Ale koukám, že se jim i tak velmi daří," prohlásil muž.

Dívka zavrtěla hlavou. „Tak to působí kvůli tomu, jak je to nakreslené."

29-2-6 Souvislá plocha — 11 bodů

Několik skupin lidí se přetahuje o určitý předmět. Do tabulky o R řádcích a S sloupcích barvami vyznačujeme, kdo ho vlastnil který den. Zakreslujeme po řádcích, když dojdeme na konec řádku, pokračujeme na dalším.

Informace k nám ale chodí, jen když se majitel změní. Dostaneme tak třeba informaci, že první skupina měla předmět 5 dní, druhá skupina 2 dny, první skupina 3 dny, ...

Nejúspěšněji působí skupina, jejíž barva je nejvíc vidět, a nejvíc vidět je souvislá oblast. Oblast je souvislá, když jsou v ní všechna políčka sousední, tj. sdílí hranu (nestačí sousedit rohem). Nás zajímá zejména to, která skupina má největší souvislou oblast.

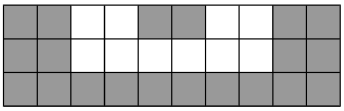
Předpokládejte, že skupin je málo, maximálně 200. Naopak tabulka může být obrovská, počítejte s tím, že se vám nemusí vejít do paměti. Ale jeden řádek se do paměti určitě vejde.

Na vstupu dostanete čísla R, S, K, Z, popisující rozměry tabulky, počet změn skupin a počet změn vlastnictví (včetně prvotního získání). Na dalších Z řádcích jsou popsané jednotlivé změny vlastnictví — vždy která skupina předmět získala a na jak dlouho. Na výstup vypište číslo skupiny, které patří největší souvislá oblast.

Ukázkový vstup:	*Ukázkový výstup:*

`3 10 2`
`0 2`
`1 2`
`0 2`
`1 2`
`0 4`
`1 6`
`0 12`
	`0`

Tabulka z příkladu vypadá následovně:



Největší souvislou oblast má skupina 0 (na obrázku šedá).

„Ale není to jediné, co působí jinak, než to opravdu je," pokračovala dívka.

„Věříš tomu, že ta holka má zvláštní moc."

„Jo! Do háje, vždyť má svou lampu na Starým městě," vyhrkla dívka rozčileně. Vzápětí se uklidnila a pokračovala: „Vážně, dneska jsem se s ní velmi zblízka potkala na Malostranském náměstí a je to z ní cítit.

Vlastně jsem se chtěla porozhlédnout i na jejím pokoji na koleji, ale přestože jsem se po budově potloukala, dokud neměla být pryč, jakmile jsem zamířila k jejímu pokoji, proběhla okolo mě, přímo k tomu pokoji. Takže jsem to radši vzdala. Ale když jsem si pak úplně jinde znova procházela, co o ní víme, najednou stála přede mnou. Koukla po náměstí, hodila po mně výhružný pohled, pak ještě koukla na zastávku, něco si poznamela a zmizela. Nepřišlo by ti to zvláštní?

A vůbec," podívala se dívka přímo na Eriku a její tón se změnil na pobavený, „kdyby neměla naši moc, jak by se sem jen tak dostala?"

Erika lehce lapla po dechu. Muž chvíli nechápavě stál, pak se najednou otočil na Eriku. Několikrát přejel pohledem mezi oběma dívkami. Mírně se usmál. Nakonec prohlásil: „To zní ovšem jako úplně jiný příběh... "

... který už s vámi nemohla sledovat

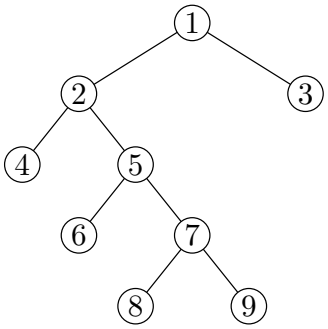
Karry Burešová

29-2-7 Strom ve stromu — 15 bodů

V druhém dílu našeho stromového seriálu ukážeme, jak popisovat podstromy pomocí DFS očíslování. Hlavně si ale předvedeme, jak vytvářet datové struktury pro stromy tím, že daný strom uložíme do úplně jiného stromu, totiž intervalového.

DFS očíslování

Začněme opakováním z minula. Spustíme-li na zadaném stromu prohledávání do hloubky (DFS), můžeme jeho průběh popsat posloupností levých a pravých závorek: levou zapíšeme, kdykoliv shora vstoupíme do vrcholu, pravou, jakmile se chystáme vystoupit nahoru.



Například pro strom na obrázku vyjde posloupnost závorek `(((()(()(()()))))())`. Každému vrcholu jsme takto přiděli jeden pár závorek a tyto páry se navzájem nekříží (říkáme, že tvoří dobré uzávorkování).

Navíc si ke každému vrcholu v zapamatujeme čísla $in(v)$ a $out(v)$. Ta budou říkat, na kolikáté pozici v řetězci závorek se nachází levá, resp. pravá závorka odpovídající vrcholu v. Pro náš ukázkový strom to vyjde takto:

v	1	2	3	4	5	6	7	8	9
$in(v)$	1	2	16	3	5	6	8	9	11
$out(v)$	18	15	17	4	14	7	13	10	12

Můžeme si to představit také tak, že máme nějaké hodiny (počítadlo), které odtikávají jednotlivé kroky DFS, a hodnoty *in* a *out* pro daný vrchol udávají čas prvního vstupu a posledního výstupu.

Všechny *iny* a *outy* dokážeme spočítat v lineárním čase. Potom nám prozradí ledacos zajímavého o tvaru stromu. Například pomocí nich můžeme poznat vzájemný vztah dvou vrcholů. Představme si nějaké dva vrcholy u a v:

- Pokud u je potomkem v (ať už přímým či nepřímým), musí být pár závorek patřících k u uvnitř toho od v, takže musí platit
$$in(v) < in(u) < out(u) < out(v).$$
- Pokud je naopak v potomkem u, dostáváme
$$in(u) < in(v) < out(v) < out(u).$$
- V ostatních případech musí jeden pár závorek skončit, než druhý začne (páry se přeci nekříží), takže nastane jedna z těchto možností:
$$in(u) < out(u) < in(v) < out(v),$$
$$in(v) < out(v) < in(u) < out(u).$$

Dovedeme tedy v konstantním čase zjistit, jaký je „příbuzenský vztah“ u a v.

Nestromové hrany

Občas se potkáme s případy, kdy mezi vrcholy stromu vedou ještě nějaké další hrany, které nejsou přímo součástí stromu (jako když vánoční stromeček ozdobíme řetězy). Těmto hranám říkáme *nestromové* a často nás zajímá, mezi jakými částmi stromu vedou.

Úkol 1 [2b]: Je dán strom na n vrcholech a m nestromových hran. Předpočítejte v čase $\mathcal{O}(n+m)$ tabulku a pak pomocí ní v konstantním čase odpovídejte na dotazy typu „vede nějaká nestromová hrana ven z podstromu s kořenem v?“.

Intervalové stromy

Brzy se nám bude hodit uložit všechno, co víme o daném stromu, do šikovné datové struktury – překvapivě opět stromové (její struktura má ale s podobou původního stromu pramálo společného).

Intervalový strom je datová struktura, která si pamatuje nějakou posloupnost $x_1, \ldots, x_m$ a umí s ní provádět následující operace:

- *Init*$(x_1, \ldots, x_n)$ – *inicializace* – $\mathcal{O}(n)$
vytvoří nový strom se zadanými hodnotami
- *Get*(i) – *bodový dotaz* – $\mathcal{O}(\log n)$
vrátí aktuální hodnotu x_i
- *Set*(i, t) – *bodový update* – $\mathcal{O}(\log n)$
nastaví x_i na hodnotu t
- *RangeMin*(i, j) – *intervalový dotaz* – $\mathcal{O}(\log n)$
spočítá minimum z $x_i, x_{i+1}, \ldots, x_j$
- *RangeAdd*(i, j, δ) – *intervalový update* – $\mathcal{O}(\log n)$
přičte ke všem prvkům $x_i, \ldots, x_j$ hodnotu δ

Existuje mnoho verzí intervalových stromů. Ty nejjednodušší popsané v naší kuchařce[1] neumí intervalový update, ale zato dokáží provádět bodové dotazy v konstantním čase. Nám se bude více hodit pokročilejší podoba s líným vyhodnocováním změn. Ta už zvládne všechny operace. Detaily si prosím přečtěte v Medvědově knížce,[2] v kapitole o datových strukturách.

Operace intervalového stromu lze navíc snadno ohýbat, aby počítaly něco trochu jiného. Intervalové dotazy mohou kromě maxima počítat třeba minimum nebo součet; intervalový update může například najednou nastavit všechny prvky v intervalu na novou hodnotu. Princip zůstává stejný. (Kdykoliv ale při řešení seriálu budete potřebovat nějakou atypickou operaci, nestačí si ji vymyslet – je nezbytné popsat, jak přesně ty standardní upravíte.)

Strom ve stromu

Pojďme si hrát. Dostaneme nějaký strom T, v jehož každém vrcholu je uloženo jedno číslo, na počátku nulové. Chceme umět provádět dvě operace: změnit číslo uložené ve vrcholu a zjistit minimum ze všech čísel uložených v zadaném podstromu.

To je něco podobného, jako umí intervalové stromy, ovšem s podstromy namísto intervalů. Pojďme jim tedy vstup trochu „předžvýkat“. Strom „rozvineme“ do posloupnosti podle toho, jak jím prochází DFS. Pro každý vrchol v definujeme $x_{in(v)}$ jako číslo uložené ve v a $x_{out(v)} = +\infty$.

Tím vznikla posloupnost délky $2n$. V ní podstrom ležící pod vrcholem v odpovídá intervalu $x_{in(v)}, \ldots, x_{out(v)}$. Pro výpočet minima podstromu tedy stačí položit intervalový dotaz (nekonečna v *out*-ech výsledek nijak neovlivní). Změna hodnoty vrcholu v je triviální: požádáme intervalový strom o bodový update prvku $x_{in(v)}$, na $x_{out(v)}$ není třeba sahat.

Obě operace tedy pracují v čase $\mathcal{O}(\log n)$, jen nesmíme zapomenout, že jsme také spotřebovali čas $\mathcal{O}(n)$ na vytvoření intervalového stromu.

Ukázali jsme tedy, jak pomocí DFS očíslování překládat strom na posloupnost, přičemž podstromy se přeloží na intervaly v posloupnosti. S těmi se pak často hodí zacházet pomocí nějakého druhu intervalového stromu. Pojďme si to vyzkoušet na dalších příkladech…

[1] `http://ksp.mff.cuni.cz/viz/kucharky/intervalove-stromy`
[2] `http://mj.ucw.cz/vyuka/ads/`

Úkol 2 [4b]: Vytvořte datovou strukturu pro strom s obarvenými vrcholy. Na počátku jsou všechny vrcholy zelené. Chceme umět provádět tyto operace: *SetColor*(v, c) – nastaví barvu vrcholu v na červenou, zelenou nebo modrou; *CountColor*(v, c) – zjistí, jaká barva je v podstromu pod vrcholem v nejčastější (je-li to nerozhodně, odpoví, že nerozhodně).

Úkol 3 [6b]: Ještě jedna datová struktura. Na začátku dostaneme strom s vrcholy ohodnocenými přirozenými čísly, Chceme umět operaci *Touch*(v), která v podstromu pod v provede následující: nalezne minimum z nenulových čísel ve vrcholech, toto minimum od všech nenulových čísel odečte a nakonec ohlásí, jestli už se povedlo všechna čísla v podstromu vynulovat.

Dvojrozměrné intervalové stromy

Další zajímavé triky můžeme provádět s dvojrozměrnými intervalovými stromy. Do těch ukládáme dvojice čísel, které si můžeme představovat jako body v rovině. Roli intervalů pak hrají libovolné obdélníky.

Pro naše účely postačí velmi jednoduchá podoba 2D intervalových stromů, která umí takovéto operace:

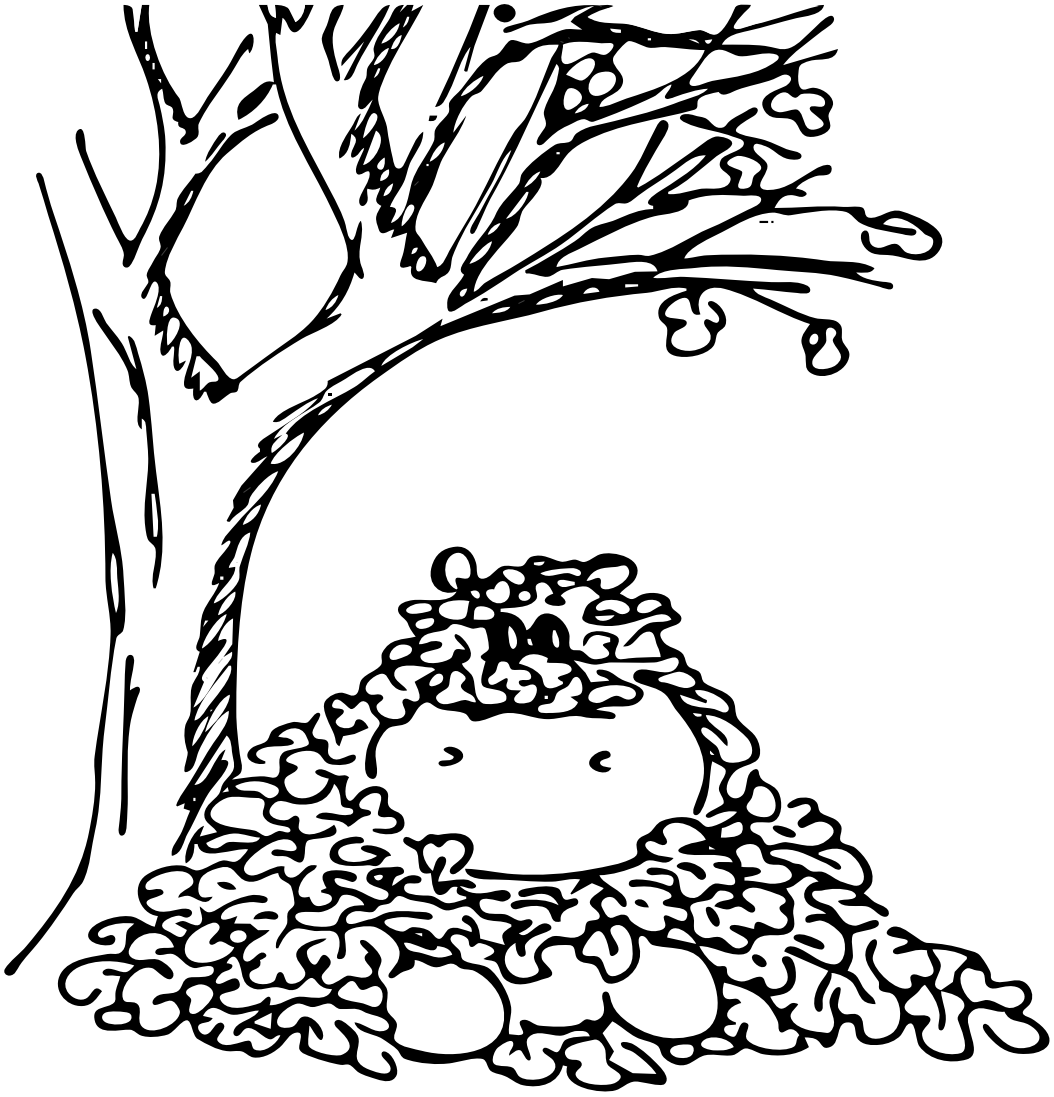
- *Init*$((x_1, y_1), \ldots, (x_n, y_n))$ – *inicializace* – $\mathcal{O}(n \log n)$
  vytvoří nový strom s body o zadaných souřadnicích
- *RangeCount*$(x_{min}, x_{max}, y_{min}, y_{max})$ – *obdélníkový počítací dotaz* – $\mathcal{O}(\log^2 n)$
  spočítá, kolik ze zadaných bodů leží v daném obdélníku, tedy pro kolik různých i je $x_{min} \leq x_i \leq x_{max}$ a současně $y_{min} \leq y_i \leq y_{max}$.

Jak takový 2D strom sestrojit, najdete například ve vzorovém řešení úlohy 24-4-7,[3] dokonce včetně mazaného triku, který zrychlí intervalový dotaz na $\mathcal{O}(\log n)$.

V následujícím úkolu nás ale konkrétní implementace nemusí zajímat, stačí použít 2D strom jako černou skříňku.

Úkol 4 [3b]: Dostaneme strom s nestromovými hranami. Chceme si něco předpočítat tak, abychom uměli rychle odpovídat na dotazy typu „existuje nestromová hrana mezi podstromem pod u a podstromem pod v?“.

Martin „Medvěd“ Mareš



[3] `http://ksp.mff.cuni.cz/viz/24-4-7/reseni`

Recepty z programátorské kuchařky: Hledání v textu

Řetězec je v podstatě jakákoliv posloupnost symbolů zapsaná za sebou a s nimi budeme v této kapitole pracovat. Každého napadne „vyhledávání v textu“ nebo „hledání jmen v telefonním seznamu;“ ale řetězce najdeme i na nižších úrovních informatiky. Například celé číslo zakódované v binární soustavě, které dostaneme na vstupu programu, je také jen řetězec nul a jedniček.

Jiný příklad použití řetězců (a algoritmů s nimi pracujících) najdeme v biologii. Například DNA není o mnoho více, než chytré uložení posloupnosti čtyř znaků/nukleových bazí – chceme-li hledat vzory anebo konkrétní podposloupnosti, bude se nám hodit znalost základních algoritmů pro práci s řetězci.

Nemáme bohužel šanci vysvětlit *všechny* algoritmy s řetězci, protože je příliš mnoho možných věcí, co s řetězci dělat. Převáděním řetězců na čísla (hešováním) jsme se věnovali v jiné kuchařce, v této se budeme soustředit na algoritmy, které se objevují spíše v práci s textem.

Kromě úvodu do práce s řetězci popíšeme dva *stavební kameny* textových algoritmů, což bude datová struktura pro slovníky – *trie* a vyhledání v textu s předzpracováním hledaného slova a jeho rozšíření pro více slov. S jejich znalostí pak bude mnohem snazší vymýšlet řešení složitějších, reálnějších problémů.

Jak řetězce chápat

Když programátor dělá první krůčky, často moc netuší, co s těmi řetězci vlastně může a nesmí dělat. V programovacím jazyce to je jasné – něco mu jazyk dovolí a na něco nejsou prostředky. Ale jak to je na úrovni ryze teoretické?

Jak jsme si řekli na začátku, řetězec bude posloupnost nějakých symbolů, kterým říkáme *znaky*. Tyto znaky jsou z nějaké množiny, které říkáme *abeceda*. Abeceda může být jen $\{0, 1\}$ pro čísla v binárním zápisu, klasické $\{\texttt{A-Z}, \texttt{a-z}\}$ pro anglickou abecedu anebo plný rozsah univerzální znakové sady Unicode, která má až 2^{31} znaků. Nezapomínejme, že nejenom písmena a číslice, ale i mezery a interpunkce jsou znaky!

Vidíme, že zanedbat velikost abecedy při odhadu složitosti by bylo příliš troufalé, a tak budeme velikost abecedy označovat $|\Sigma|$. Abeceda sama se v textech o řetězcích často značí řeckým Σ.

O znacích samotných předpokládáme, že jsou dostatečně malé, abychom s nimi mohli pracovat v konstantním čase, podobně jako s celými čísly v ostatních kapitolách.

Nyní hlavní otázka – máme chápat řetězec jako pole znaků, nebo jako spojový seznam? Šalamounská odpověď: můžeme s ním pracovat tak i tak. Když budeme potřebovat převést řetězec na spojový seznam (protože se nám hodí rychlé přepojování řetězců), tak si jej převedeme. Tento převod nás samozřejmě bude stát čas lineárně závislý na *délce* řetězce. Budeme ji dále značit L; časová složitost převodu bude $\mathcal{O}(L)$.

Standardně se ale počítá s tím, že řetězec je uložen v poli někde v paměti (již od začátku algoritmu), takže ke každému znaku můžeme přistupovat v konstantním čase.

Jelikož jsme řetězce definovali jako posloupnosti, nesmíme zapomínat ani na *prázdný řetězec* ε. A když už máme řetězec, určitě máme i *podřetězec* – souvislou podposloupnost znaků jiného řetězce. Například `BAR`, `RET`, ε i `KABARET` jsou podřetězce slova (řetězce) `KABARET`; `KAT` však podřetězcem není.

Často nás budou zajímat dva zvláštní druhy podřetězců. Pokud ze slova odstraníme nějaký souvislý úsek na konci, vznikne podřetězec, kterému říkáme *prefix* (česky předpona), a pokud odstraníme nějaký souvislý úsek ze začátku, dostaneme *suffix* neboli příponu. `RET` je suffix slova `KABARET`, `KABA` je zase jeho prefixem.

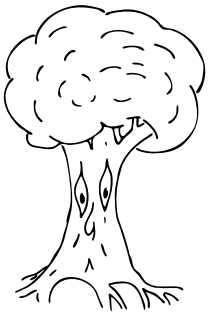
Terminologie dovoluje zepředu i zezadu odstranit prázdný řetězec – to znamená, že slovo je samo sobě prefixem i suffixem. Pokud chceme mluvit o prefixech, suffixech nebo obecně podřetězcích, kde jsme museli alespoň jeden znak odtrhnout, označíme takové podřetězce jako *vlastní*.

Pro některá použití řetězců je důležité, abychom je mohli porovnávat – když máme řetězce R a S, chceme umět rozhodnout, který je menší a který je větší. Jaké přesně toto uspořádání bude, závisí na naší aplikaci, ale mnohdy se používá tzv. *lexikografické uspořádání*.

Pro lexikografické uspořádání potřebujeme nejprve zadané lineární uspořádání na znacích. Tím se myslí takové, kde jsou všechny prvky navzájem porovnatelné, a v podstatě to znamená, že jsou uspořádány v jedné řadě za sebou (kromě binárního $\texttt{0} < \texttt{1}$ se často používá „telefonní“ $\texttt{A} = \texttt{a} < \texttt{B} = \texttt{b} < \ldots < \texttt{Z} = \texttt{z}$, které je ovšem lineární až na velikost znaků).

Když máme zadané uspořádání na znacích, na všechny řetězce je rozšíříme následovně: Nejkratší je prázdný řetězec. Ostatní řetězce třídíme podle jejich prvního znaku. Jestliže se první znak shoduje, tak podle druhého znaku, atd. Pokud přitom znaky jednoho z řetězců dojdou dřív, prohlásíme tento řetězec za menší.

Platí tedy třeba $\varepsilon < \texttt{A} < \texttt{AUTO} < \texttt{AUTOBUS} < \texttt{AUTOGRAM} < \texttt{AUTOR} < \texttt{BAMBITKA} < \texttt{BARNABAS} < \texttt{Z}$.



Adresář pomocí trie

Typický „textový“ problém je udržování množiny řetězců – můžete si představit třeba slovník. Slova ve slovníku si chceme šikovně předzpracovat, abychom pak mohli efektivně odpovídat na otázky typu: „Je slovo S obsaženo ve slovníku?“ Můžeme také po předzpracování chtít přidávat nové položky, nebo i odebírat staré.

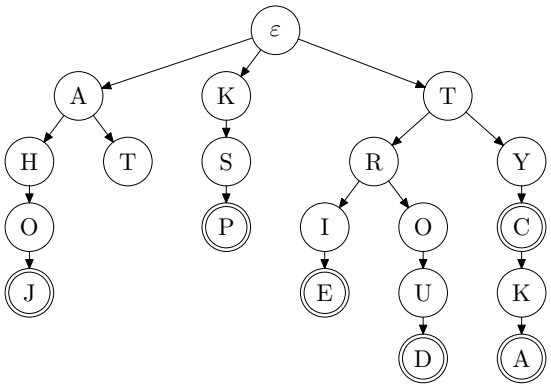
Pokud bychom nemuseli odebírat slova, můžeme použít hešování, které je rychlé a účinné. Více o něm najdete v hešovací kuchařce.[4] Má však tu nevýhodu, že při velkém zaplnění se začne chovat pomaleji a mírně nepředvídatelně.

[4] `http://ksp.mff.cuni.cz/viz/kucharky/hesovani`

Ukážeme si jiné řešení, které je také asymptoticky rychlé a není ani příliš náročné na paměť. Využívá stromové struktury a říká se mu *trie* (vyslovujeme česky „tryje“ a anglicky jako část slova „retrieval“, z něhož slovo trie vzniklo). V češtině se občas používá také označení „písmenkový strom“.

Trie bude zakořeněný strom. V prvním patře se bude větvit podle prvního písmene slova, ve druhém podle dalšího, a tak dále.

Obrázek vydá za tisíc definic, pojďme se podívat, jak vypadá trie pro slova `AHOJ`, `AT`, `KSP`, `TRIE`, `TROUD`, `TYC`, `TYCKA`. Pro přehlednost písmena místo na hrany kreslíme do následujících vrcholů:



Všimněte si, že vrcholy v hloubce h (tedy v h-tém patře trie) odpovídají prefixům délky h zadaných slov. Například prefixy délky 2 jsou `AH`, `AT`, `KS`, `TR` a `TY`. Hrana mezi prefixy vede právě tehdy, lze-li jeden z druhého získat připsáním písmene na konec.

Jak bychom takovou trii postavili algoritmem? Přesně, jak jsme ji definovali: každé slovo ze slovníku budeme procházet znak po znaku, a bude-li nějaká hrana chybět, tak ji vytvoříme a pokračujeme dále podle slova.

Z takto popsané trie bohužel nepoznáme, kde končí slovo ze slovníku a kde končí jen jeho prefix. Standardní způsoby, jak to vyřešit, jsou dva: buď si do každého vrcholu přidáme informaci o tom, je-li koncem celého slova, nebo ne (jak je to naznačeno dvojitými kroužky v obrázku), anebo si rozšíříme abecedu o speciální znak, který se v ní předtím nevyskytoval – třeba `$` – a pak všem slovům přilepíme tento `$` na konec.

Budeme-li se později ptát, bylo-li slovo ve slovníku, po průchodu trií zkontrolujeme ještě, jestli z konečného vrcholu vede hrana odpovídající znaku `$`.

Ještě jsme si nerozmysleli, jak budeme v jednotlivých vrcholech trie reprezentovat hrany do delších prefixů. Abychom mohli vyhledávat skutečně lineárně, potřebovali bychom umět v konstantním čase odpovědět na otázku „má vrchol P potomka přes hranu se znakem c?“.

Abychom zajistili konstantní čas odpovědi, museli bychom mít v každém vrcholu pole indexované znaky abecedy. To ovšem znamená, že takové pole budeme muset vytvořit, a tedy alokovat $|\Sigma|$ políček v každém znaku.

To zvýší paměťovou náročnost trie (a časovou náročnost její stavby) na $\mathcal{O}(D \cdot |\Sigma|)$, kde D značí velikost vstupu, čili součet délek všech slov ve slovníku. To je naprosto přijatelné pro malé abecedy, ale už pro {`A-Z`, `a-z`} je tento faktor roven 52 a pro Unicode je taková alokace nemyslitelná.

Jak z toho ven? Můžeme oželet konstantní rychlost dotazu a použít namísto pole třeba binární vyhledávací strom nebo hešovací tabulku všech znaků, kterými aktuální prefix může pokračovat. Nebo můžeme každý znak velké abecedy zapsat pomocí několika znaků menší abecedy. Tou menší abecedou může být třeba $\{0, 1\}$. Tehdy nahradíme každý znak původní abecedy $\lceil \log_2 |\Sigma| \rceil$ novými (jeho zápisem ve dvojkové soustavě). Tím se časová slozitost konstrukce zlepší na $\mathcal{O}(D \cdot \log |\Sigma|)$ a časová složitost dotazu na slovo délky L zhorší na $\mathcal{O}(L \cdot \log |\Sigma|)$.

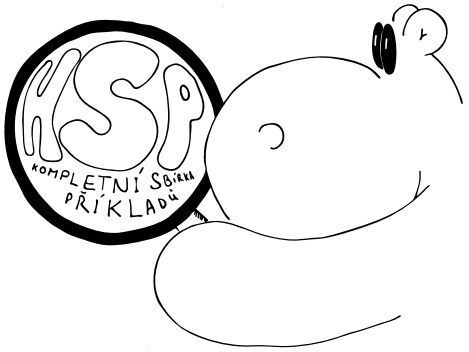
A jsme hotovi! S trií můžeme v lineárním čase odpovídat na dotazy „Vyskytuje se dané slovo ve slovníku?“, přidávat a odebírat další položky za běhu a nejen to – víc o tom ve cvičeních.

Poznámky

- Chcete-li algoritmus konstrukce trie vidět napsaný v Pascalu, podívejte se do knihy *Algoritmy a programovací techniky*.
- Triím se také říká *prefixové stromy*, což popisuje, že každý vrchol odpovídá prefixu některého slova ve slovníku.
- Kdybychom chtěli, mohli bychom pomocí trie vyhledávat v textu v lineárním čase. Můžeme přeci postavit slovník ze všech slov v daném textu, a pak procházet příslušnou trii. Má to ale pár háčků: jednak je často hledaný řetězec krátký, ale text se nevejde do paměti; jednak pokud bychom použili jako oddělovač mezery, mohli bychom hledat jen jednotlivá slova, a nikoli jejich konce nebo delší kusy věty.
- Asi se po poslední poznámce ptáte – existuje nějaká modifikace trie, která umí hledat libovolnou část textu? Ano, jmenuje se *suffixový strom* a jdou s ní dělat spousty krásných kousků. Říká se, že každou řetězcovou úlohu lze řešit v lineárním čase pomocí suffixových stromů. Víc se o nich dočtete třeba v knížce *Krajinou grafových algoritmů*.[5]

Cvičení

- Řekněme, že chceme slovník na vstupu setřídit v lexikografickém pořadí (definovaném v sekci „Jak řetězce chápat“). Problémem pro klasické třídící algoritmy je to, že porovnání dvou řetězců není bohužel konstantně rychlé. Vymyslete způsob, jak setřídit takový slovník rychle pomocí trie.
- *Komprese trie.* Co kdybychom chtěli odstranit přebytečné vrcholy trie, tedy ty, v nichž se slova nevětví? Rozmyslete si, jestli by něčemu vadilo místo takovýchto cest mít jen jednotlivé hrany. Zesložití se konstrukce nebo vyhledávání? Mimochodem, je celkem jasné, že takováto *komprimovaná trie* přinese jen konstantní zrychlení dotazů i prostoru, a tak na soutěžích apod. stačí použít základní variantu.

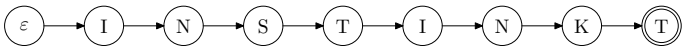


[5] `http://mj.ucw.cz/vyuka/ga/`

Vyhledávání v textu

Začátek situace je asi zřejmý – máme na vstupu zadán dlouhý text a krátké slovo. Slovo si můžeme nějak předzpracovat, načež projdeme co nejrychleji text a nahlásíme jeden nebo všechny výskyty slova. Zajímají nás při tom i výskyty, které se navzájem překrývají: v textu `NANANA` se slovo `NANA` vyskytuje dvakrát. Často se hovoří o „hledání jehly v kupce sena“, pročež se textu přezdívá *seno* a hledanému slovu *jehla*. Délku jehly označíme J a délku textu S.

Představme si nejdříve hledané slovo jako spojový seznam, třeba slovo `INSTINKT`:



Mohli bychom text začít procházet znak po znaku a kontrolovat, zda se text shoduje s naším slovem/spojovým seznamem. Pokud by si znaky odpovídaly, skočíme na další znak z textu a i na další znak v seznamu. Co když se ale neshodují? Pak nemůžeme jen skočit na další znak textu – co kdybychom v textu narazili na slovo `INSTINSTINKT`?

Musíme se tedy vrátit nejen na začátek spojového seznamu, ale i zpátky v textu na druhý znak, který jsme označili jako odpovídající, a zkoušet porovnávat s jehlou znovu od začátku. To už naznačuje, že takto získaný algoritmus nebude lineární, protože se musí vracet zpět v textu o délku jehly.

Sice je předchozí popis skutečně v nejhorším případě složitý $\mathcal{O}(S \cdot J)$, avšak stačí malá úprava a složitost přejde na lineární $\mathcal{O}(S + J)$. Ve skutečnosti algoritmus nezpomalovalo vracení se – za špatnou složitost mohl fakt, že jsme se vraceli *příliš zpátky*.

Třeba v našem příkladu s textem `INSTINSTINKT` se nemusíme vracet ve spojovém seznamu na začátek, jakmile načteme `INSTINS`. Mohli jsme se vrátit jen na druhý znak, tedy do prvního `N`, a pak kontrolovat, jaký znak pokračuje dál. Když následuje `S` jako v našem případě, můžeme pokračovat dále v čtení a nevracíme se v textu. Kdyby text byl jiný, třeba `INSTINB`, vrátili bychom se po načtení `B` na začátek spojového seznamu a v textu bychom pokračovali dále bez zastavení.

Pro každý znak ve spojovém seznamu si tedy určíme políčko spojového seznamu, na které skočíme, pokud se následující znak v textu liší od toho očekávaného. Pořadové číslo tohoto políčka nám poradí tzv. *zpětná funkce* F, což bude funkce definovaná pomocí pole, kde $F[i]$ bude pořadové číslo políčka, na které se má skočit z políčka číslo i. Porovnávat pak budeme s následujícím znakem. Pokud $F[i] = 0$, znamená to, že máme začít porovnávat úplně od prvního znaku jehly.

Pokud máte rádi grafovou terminologii, můžete se na náš spojový seznam dívat jako na graf a hovořit o *zpětných hranách*.

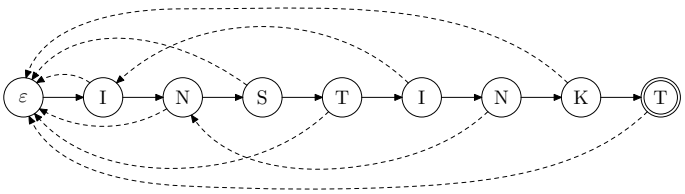
Zatím jsme ale přesně nepopsali, na které políčko přesně bude zpětná funkce ukazovat. Nechť chceme určit zpětné políčko pro druhé `N` ve slově `INSTINKT`. Pracujeme teď s prefixem `INSTIN`. Selsky řečeno, chceme najít „konec slova `INSTIN` takový, že je stejný, jako začátek slova `INSTIN`“.

Abychom náš požadavek upřesnili, zamyslíme se nad zpětným políčkem pro jiné slovo. Co kdyby jehlou bylo slovo `ABABABC` a my určovali zpětné políčko pro `ABABAB`? Kdybychom ukázali na první písmenko `B`, nebylo by to správně, protože pak bychom pro text `ABABABABC` nezahlásili výskyt jehly, což je jasná chyba. Musíme se vrátit už na `ABAB`!

Zajímá nás tedy ne libovolný suffix, který je stejný jako začátek, ale nejdelší takový konec/suffix. A ještě navíc ne jen ten nejdelší, ale nejdelší „netriviální“ – slovo `INSTIN` je samo sobě prefixem a suffixem, ale zpětná funkce pro `N` by se neměla cyklit, měla by vést zpátky.

Řekněme to tedy znova, zcela formálně: pokud bychom právě určovali hodnotu zpětné funkce pro znak číslo i, kterému odpovídá prefix P, pak její hodnota bude *délka nejdelšího vlastního suffixu* slova P, pro který ještě platí, že *je zároveň prefixem* P.

Pro slovo `INSTINKT` vypadá spojový seznam se zpětnou funkcí (zakreslenou pomocí ukazatelů) takto:

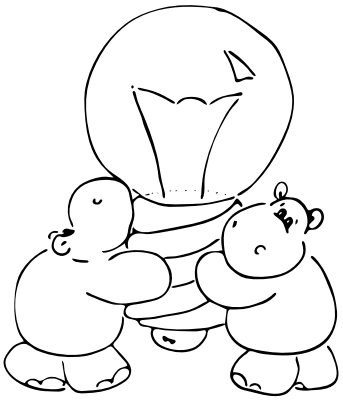


Nyní vyvstávají dvě otázky: Jakou to má celé časovou složitost? A jak spočítat zpětnou funkci?

Poperme se nejdříve s tou první. Pro každý znak vstupního textu mohou nastat dva případy: Buď znak rozšiřuje aktuální prefix, nebo musíme použít zpětnou funkci. První případ má jasně konstantní složitost, druhý je horší, neboť zpětná funkce může být pro jeden znak volána až J-krát.

Při každém volání však klesne pořadové číslo aktuálního stavu (políčka) alespoň o jedna, zatímco kdykoliv stav prodlužujeme, roste jen o jeden znak. Proto všech zkrácení dohromady může být nejvýše tolik, kolik bylo všech prodloužení, čili kolik jsme přečetli znaků textu. Celkem je tedy počet kroků automatu lineární v délce textu, tj. $\mathcal{O}(S)$.

Konstrukci zpětné funkce provedeme malým trikem. Všimněme si, že $F[i]$ je přesně číslo stavu, do nějž se dostaneme při spuštění našeho vyhledávacího algoritmu na řetězec, který tvoří prefix délky i z jehly bez prvního znaku.



Proč to tak je? Zpětná funkce říká, jaký je nejdelší vlastní suffix daného stavu, který je také stavem, zatímco políčko, ve kterém po i krocích skončíme, označuje nejdelší suffix textu, který je stavem. Tyto dvě věci se přeci liší jen v tom, že ta druhá připouští i nevlastní suffixy, a právě tomu zabráníme odstraněním prvního znaku.

Takže F získáme tak, že spustíme vyhledávání na část samotné jehly. Jenže k vyhledávání zase potřebujeme funkci F. Jak z toho ven? Budeme zpětnou funkci vytvářet postupně od nejkratších prefixů. Zřejmě $F[1] = 0$. Pokud již

máme $F[i]$, pak výpočet $F[i+1]$ odpovídá spuštění automatu na slovo délky i a při tom budeme zpětnou funkci potřebovat jen pro stavy délky i nebo menší, pro které ji již máme hotovou.

Navíc nemusíme pro jednotlivé prefixy spouštět výpočet vždy znovu od začátku – $(i+1)$-ní prefix je přeci prodloužením i-tého prefixu o jeden znak. Stačí tedy spustit algoritmus na jehlu bez prvního znaku a sledovat, jakými stavy bude procházet – to budou přesně hodnoty zpětné funkce.

Vytvoření zpětné funkce se nám tak nakonec zredukovalo na jediné vyhledávání v textu o délce $J-1$, a proto poběží v čase $\mathcal{O}(J)$. Časová složitost celého algoritmu tedy bude $\mathcal{O}(S+J)$. Dodáme už jen, že tento algoritmus poprvé popsali pánové Knuth, Morris a Pratt a na jejich počest se mu říká KMP. Naprogramovaný bude vypadat následovně (čtení vstupu jsme si odpustili):

```
jehla = "INSTINKT"
seno = "INSTINSTINKTINSTINKT"
J = len(jehla)
S = len(seno)
F = [None] * J  # Zpětná funkce

def krok(i, znak):
    if i < J and jehla[i] == znak:
        return(i + 1)
    elif i > 0:
        return krok(F[i - 1], znak)
    else:
        return 0

# Konstrukce zpětné funkce
F[0] = 0
for i in range(1, J):
    F[i] = krok(F[i - 1], jehla[i])

# Procházení textu
stav = 0
for i in range(S):
    stav = krok(stav, seno[i])
    if stav == J:
        print(i - J + 1, "až", i)
```

Poznámky

- Pro anglický nebo český text je použití takto sofistikovaného algoritmu skoro škoda, protože v obou jazycích se stává jen málokdy, že bychom měli několik slov spojených dohromady. Prakticky bude stačit i na začátku zmíněný naivní algoritmus. Na soutěžích a olympiádách ale pište raději algoritmus KMP.
- Hešování lze použít i na vyhledávání řetězce v textu. Obzvláště vhodné jsou na to *rolling hash functions* (neboli „okénkové hešovací funkce“), které umí v konstantním čase přepočítat heš, ubereme-li nějaký znak na začátku a přidáme-li jiný na konci – jako kdybychom se dívali na text skrz posouvající se okénko.

Cvičení

- Rozmyslete si, že když vyhledáváme více slov, ne jen jedno, a algoritmus musí vypsat všechny výskyty na výstup, můžeme se dobrat vyšší než lineární složitosti v závislosti na vstupu. Na čem potom taková časová složitost také záleží?
- Vymyslete nějakou vhodnou okénkovou hešovací funkci pro vyhledávání jedné jehly.

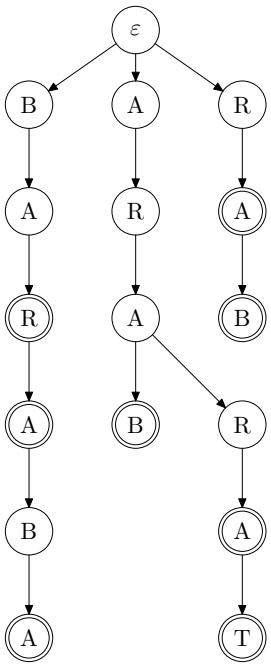
Vyhledávání jehelníčku

Co kdybychom neměli jen jednu jehlu/hledané slovo, ale celý jehelníček, čili seznam hledaných slov? I to lze řešit podobnou metodou, jakou jsme hledali jedno slovo. Tento algoritmus se nazývá po tvůrcích *algoritmus Aho-Corasicková* a spočívá v tom, že jednoduchý spojový seznam nahradíme trií a do trie opět přidáme zpětné hrany.

Budeme postupovat podobně jako u KMP. Nejprve naskládáme jehelníček do trie. Pro příklady v této kuchařce použijeme jehelníček `ARAB`, `ARARA`, `ARARAT`, `BAR`, `BARA`, `BARABA`, `RA` a `RAB`.

Dalším krokem v KMP bylo sestrojení zpětných hran. Nejprve jsme sestrojili zpětnou hranu pro první znak slova, pak pro druhý atd. V trii to bude o něco složitější.

Na první pohled se může zdát, že bychom mohli automat sestrojit tak, že bychom vyrobili KMP pro první slovo, pak KMP pro druhé slovo s využitím struktury prvního atd., ale to má háček.



Zpětné hrany totiž nemusí vést do předka. Například pro slovo `BARAB` povede zpětná hrana do slova `ARAB`, z toho do slova `RAB` a z toho do `B`. Kdybychom ale zkonstruovali automat výše popsaným způsobem (a začali slovem `BARAB`), nebude existovat v trii ani `ARAB`, ani `RAB`, takže bychom vedli zpětnou hranu chybně do `B`.

Můžeme se ale opřít o stejný trik, jako při konstrukci KMP. Budeme opět vyhledávat nejdelší vlastní suffix. Kam dojde výpočet po jeho vyhledání, tam povede zpětná hrana.

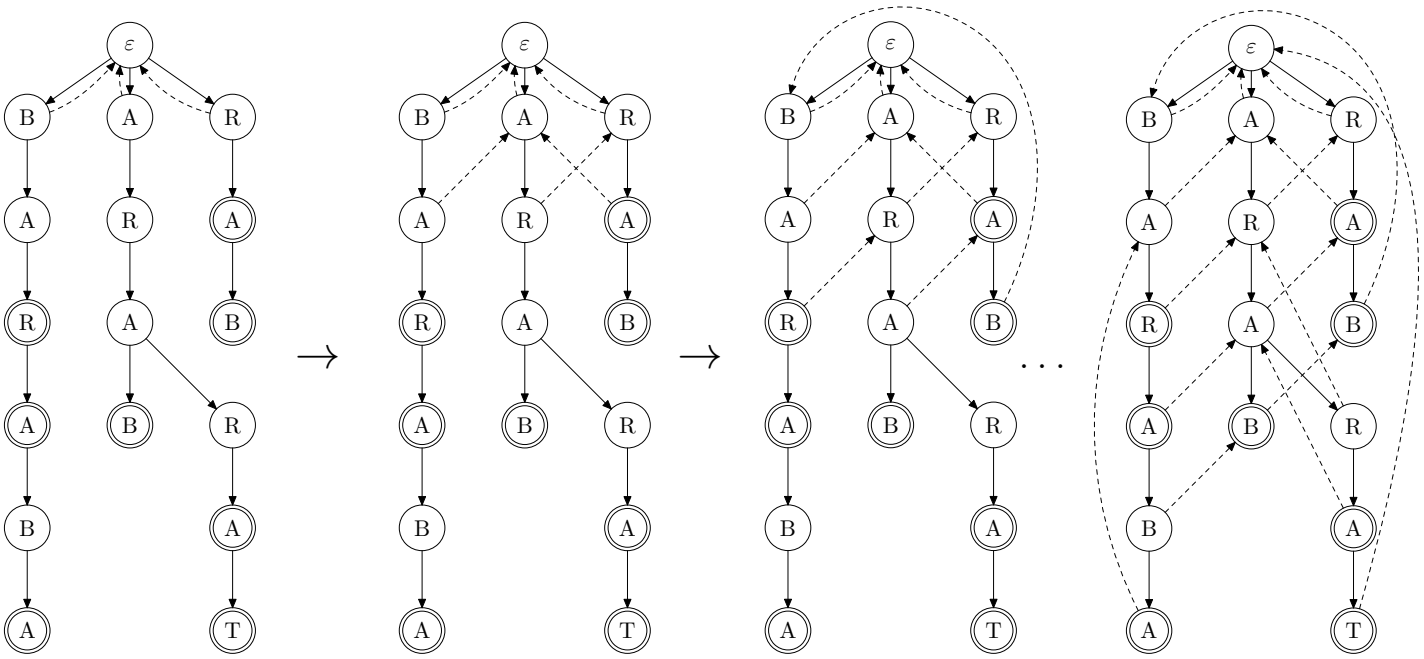
Zkusíme tedy nejprve sestrojit celou trii a pak postupně vyhledat nejdelší vlastní suffix pro každé ze slov. Ouha, to ale také nefunguje. Když začneme slovem `BARABA`, a budeme tedy vyhledávat `ARABA`, nalezneme v trii úspěšně prefix `ARAB`, ale `ARABA` již v trii není. Měli bychom přejít ze slova `ARAB` po zpětné hraně, ale tu ještě nemáme zkonstruovanou.

Rozdělíme si trii na *vrstvy* – první znaky slov budou první vrstva, druhé znaky budou tvořit druhou vrstvu atd., až i-té znaky slov budou tvořit i-tou vrstvu.

Zpětná hrana jistě povede do kratšího slova. Z i-té vrstvy tak povede do vrstvy s nižším pořadovým číslem. Pokud tedy budeme zpětné hrany konstruovat po vrstvách, dojdeme kýženého výsledku.

Ještě zbývá otázka, jak konstruovat zpětné hrany efektivně, když je musíme vyrábět po vrstvách. Mohli bychom prostě vzít slovo, pro které hledáme zpětnou hranu, utrhnout mu první znak a vyhledat. Jenže to budeme dělat spoustu práce zbytečně.

Například pro slovo `BARABA` bychom mohli vyhledávat `ARABA` v již zkonstruované části automatu, ale proč to dělat celé, když jsme při konstrukci předchozí vrstvy vyhledávali `ARAB` při konstrukci zpětné hrany pro `BARAB`?

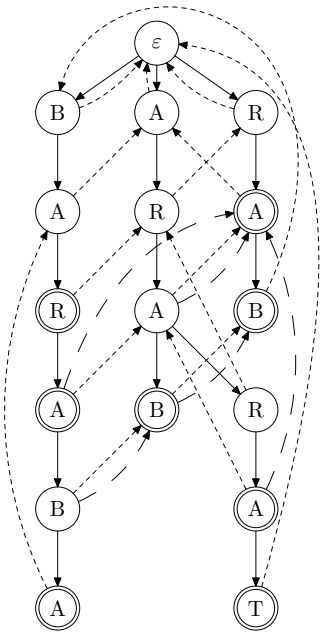


Při konstrukci další zpětné hrany tedy najdeme akorát, kde jsme minule skončili, a odtamtud pokračujeme dál. Jak to najdeme? Z otce našeho vrcholu tam přece vede zpětná hrana. Takže můžeme postup shrnout do bodů:

1. $c =$ poslední znak slova (znak stavu P, pro který hledáme zpětnou hranu);
2. přesuneme se do otce;
3. přesuneme se po zpětné hraně;
4. dokud neexistuje syn se znakem c nebo nejsme v kořeni, přesouváme se po zpětných hranách;
5. pokud existuje syn se znakem c, natáhneme do něj zpětnou hranu z P, jinak ji natáhneme do kořene.

Automat je zkonstruován. Časová složitost konstrukce sestává z konstrukce trie v $\mathcal{O}(J \cdot |\Sigma|)$, resp. $\mathcal{O}(J \cdot \log |\Sigma|)$ (pokud použijeme binární strom ve vrcholech) a z předpočítání zpětných hran. Při předpočítávání uděláme nějaký konstantní počet operací pro každý vrchol (celkem tedy $\mathcal{O}(J)$) a také paralelně vyhledáváme všechny jehly z jehelníčku, jejichž vyhledání nás stojí $\mathcal{O}(J)$, resp. $\mathcal{O}(J \cdot \log |\Sigma|)$.

Tedy konstrukce trvá celkem $\mathcal{O}(J \cdot |\Sigma|)$, resp. $\mathcal{O}(J \cdot \log |\Sigma|)$, paměťová náročnost je stejná jako u trie – $\mathcal{O}(J \cdot |\Sigma|)$, resp. $\mathcal{O}(J)$, přidali jsme jen $\mathcal{O}(J)$ zpětných hran.



Zkusme tedy automatem projít text `BARABARARAT`. Ohlásí postupně nález slov `BAR`, `BARA`, `BARABA`, `BAR`, `BARA`, `ARARA` a `ARARAT`.

Nenalezl však všechno. Chybí mu např. `ARAB`, který začíná druhým znakem a končí pátým. Dále chybí několik výskytů `RA` a jeden `RAB`.

Když byl algoritmus na pátém znaku, byl ve stavu `BARAB`, jehož suffixem je `ARAB`. Obecně na suffixy zapomínáme. Na rozdíl od KMP, kde suffix aktuálního stavu nikdy nebyl jehla, tady jehlou být může.

V každém stavu bychom tedy měli projít všechny suffixy a zkontrolovat je, jestli náhodou nejsou jehlami. Jak najdeme všechny suffixy? Projdeme postupně po zpětných hranách až do kořene. Má to jen jeden problém – je to pomalé.

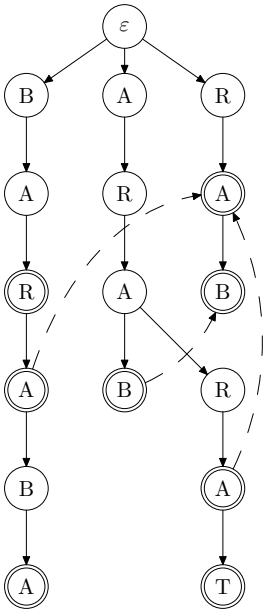
Představme si například slovník obsahující `A` a `AAAA...A` (délky $J-1$). Budeme-li jím vyhledávat v textu `AAAA...A` délky $S > J$, projdeme prakticky pro každý znak až $J-1$ zpětných hran, čímž složitost naroste až na nepoužitelných $\mathcal{O}(S \cdot J)$.

Všimněme si však, že většinou zpětných hran jsme prošli úplně zbytečně. Předpočítáme si tedy *zkratky* – z vrcholu vede zkratka do nejdelšího jeho suffixu, který je jehlou. Na obrázku jsou vyznačeny dlouze čárkovanými šipkami.

Předpočítání zpětných hran časovou složitost konstrukce automatu jistě nezhorší, neboť vyžaduje v nejhorším případě projít všechny zpětné hrany ještě jednou.

Potřebujeme-li ohlásit všechny výskyty slov včetně pozice, kde se nacházejí, jsme hotovi. Výsledná časová složitost prohledávání bude $\mathcal{O}(S + O)$, resp. $\mathcal{O}(S \cdot \log |\Sigma| + O)$, kde O je velikost výstupu – počet výskytů všech slov.

Celková časová složitost prohledávání včetně stavby automatu tedy bude $\mathcal{O}(O + S + J \cdot |\Sigma|)$, resp. $\mathcal{O}(O + (S + J) \cdot \log |\Sigma|)$.



Jak velký může být výstup? Obecně až S^2. Extrémně velký výstup je možné vygenerovat třeba slovníkem obsahujícím všechny prefixy slova `AAAA...A` délky S a senem taktéž `AAAA...A` délky S. Automat pak hlásí výskyt pro každé podslovo, kterých je řádově S^2.

Pokud nám stačí u každého slova jen počet výskytů, nemusíme zoufat – závislost na počtu výskytů umíme odstranit.

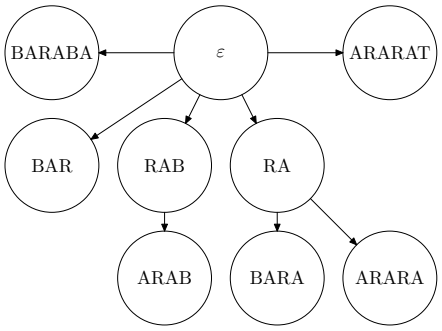
Použijeme trik – na každé pozici započítáme pouze nejdelší jehlu, která tam končí (u každé jehly si budeme udržovat čítač). Nebudeme tedy v každém kroku poskakovat po zkratkách až do aleluja, ale

maximálně jednou. Díky tomu nám z časové složitosti zmizí velikost výstupu.

V našem příkladu se senem `BARABARARAT` tedy na konci budeme mít uloženo, že `ARAB` se vyskytnul 1×, `ARARA` 1×, `ARARAT` 1×, `BAR` 2×, `BARA` 2× a `BARABA` 1×. `RA` a `RAB` nemají hlášený žádný výskyt.

Nyní si zkonstruujeme strom jenom ze zkratek a pro každý vrchol spočítáme součet celého jeho podstromu.

Tedy po přepočtu bude mít `RA` tři výskyty a `RAB` jeden výskyt; celkový počet výskytů pak bude 12.



Poznámky

- Dalším krokem po KMP a Aho-Corasickové jsou konečné automaty a regulární výrazy, o kterých jsme měli seriál ve 23. ročníku.
- Není moc rozumné snažit se implementovat Aho-Corasickovou v rozumné době například při soutěži, pokud tento algoritmus nemáte opravdu pod kůží. Radši zkuste použít hešování, pokud budete něco takového potřebovat.

Cvičení

- Redukci o velikost výstupu můžeme provést i pro případ, kdy výstup nebudeme vypisovat, ale stačí nám mít jej uložený v paměti. Vymyslete vhodnou úpravu triku s čítačem.
- Zkuste si implementovat Aho-Corasickovou vlastnoručně ve svém oblíbeném jazyce, abyste si byli jisti, že doopravdy chápete všechny záludnosti tohoto algoritmu.

Martin Böhm, Jan Matějka, Martin Mareš a Petr Škoda

Vzorová řešení první série dvacátého devátého ročníku KSP

29-1-1 Připálené placky

Než se pustíme do samotného řešení, tak si úlohu trochu upravíme. Namísto hromady placiček otočených tím či oním způsobem budeme otáčet pole jedniček a nul.

Jednička bude representovat placičku otočenou správně (tedy spálením dolů), nula otočenou špatně. Jedinou dovolenou operací je překlopení (jedničky na nulu a nuly na jedničku) všech hodnot od první až do nějaké i-té (včetně). Tedy vlastně negace prefixu (souvislé podčásti začínající na levém okraji) pole.

Hned na začátku si je třeba uvědomit jednu velmi důležitou věc ze zadání. Nechceme otáčení hodnot, tedy překlápění placiček, opravdu provést. Stačí nám říct, kde všude by to bylo třeba.

Toto pozorování způsobí, že naše řešení bude nakonec lineární a ne kvadratické, jak by se na první pohled mohlo zdát.

Pro začátek se pokusme na pole hodnot podívat jako na soustavu stejnorodých úseků jedniček a nul. Snadno si za chvilku dokážeme, že překlápět hodnotu má smysl jen u prefixů, které končí právě v přechodech mezi souvislými úseky jedniček a nul (či obráceně).

Určitě platí, že v konečném stavu, tedy když jsou všechny hodnoty jedničky, nejsou v poli žádné přechody. Celá věž je totiž jeden stejnorodý úsek.

Otočením hodnot úseku končícím na rozhraní určitě zrušíme jeden přechod. Změníme totiž hodnotu těsně před rozhraním a beze změny ponecháme tu těsně za ním. Což, pokud máme jen dvě možné hodnoty, které se před tím lišily (a vytvářely tak přechod), znamená, že nyní musí být stejné.

Operací také nikterak neovlivníme jiné úseky před nebo za původním rozhraním. Těch za rozhraním se totiž ani nedotkneme a těm před pouze otočíme hodnoty.

Pokud by úsek, v našem případě prefix, který operací změníme, končil mimo přechod, tak nejenže se žádného rozhraní nezbavíme, ale dokonce vytvoříme nové. Znegujeme totiž hodnoty první poloviny nějakého předtím stejnorodého úseku, čímž ho nutně rozdělíme na dvě nové části s různými hodnotami.

Všem ostatním úsekům před koncem prefixu pak pouze přehodíme hodnoty, tedy žádný neodstraníme, a na ty dále za nikterak nezměníme. Skončíme tedy jen s přidáním jednoho přechodu. Je vidět, že má smysl uvažovat jen o úsecích končících v místech již existujících přechodů.

Algoritmus je již nyní snadný. Vytvoříme si pomocnou proměnnou, která si bude pamatovat typ posledního úseku. Na začátku ji nastavíme na hodnotu prvního prvku v poli. Následně pro každý další prvek pole uděláme jednu ze dvou věcí.

Pokud je jeho hodnota stejná jako v pomocné proměnné, tak pokračujeme normálně dál. Pokud je jiná, tj. nastal zlom, tak zahlásíme otočení až do posledního indexu (včetně) a pomocnou proměnnou aktualizujeme hodnotou prvku, který právě zpracováváme; tedy hodnotou která je těsně za zlomem.

Takhle dojdeme až na konec, kde ještě zkontrolujeme, zda je poslední hodnota 1. Když není, tak zahlásíme operaci nad celým polem, tj. otočení všech placek.

Algoritmus zjevně potřebuje pouze jeden lineární průchod plackami. Časová složitost je tedy $\mathcal{O}(N)$, kde N je počet placek.

U paměové složitosti záleží na tom, jak si ji zavedeme. Pokud si povolíme číst vstup postupně a stejně tak i vypisovat výstup, tak nám stačí konstantní množství paměti. V průběhu si totiž stačí pamatovat onu pomocnou proměnnou. Pokud však použijeme staromdní definici, tak si musíme pamatovat celý vstup najednou, což automaticky znamená i prostorovou složitost $\mathcal{O}(N)$.

Petr Houška

29-1-2 Kupecké počty

Každý kupec ví, kolik zaplatil, tedy se domluví, napíšou všechny částky na papír a spočítají z nich průměr. To je částka, kterou každý z nich chce nakonec zaplatit. Někteří zaplatili víc, jiní méně. V seznamu kupců tedy od výše každé investice odečteme průměr.

Tím dostaneme pro každého kupce buďto kladné číslo – o tolik zaplatil do projektu víc, a tedy tolik má od ostatních dohromady dostat – nebo záporné číslo – o tolik zaplatil méně, a tedy tyto peníze nějak rozdělí svým kolegům.

Můžou nám vyjít i nuly. Nulový kupec zaplatil právě tolik, kolik zaplatit měl, a nemusí se tedy s nikým vyrovnávat. Vyřadíme jej ze seznamu, neboť takových kupců by klidně mohlo být docela dost a zbytečně by kazili časovou i paměťovou složitost zbytku algoritmu.

Kupci se mezi sebou budou vyrovnávat tak, že se vždy potká nějaký dlužník (to je ten, co má platit) s nějakým věřitelem (to je ten, co má peníze dostat) a zaplatí si nějakou částku.

Kolik si můžou zaplatit? Označíme-li celkovou výši dluhu jako D a celkovou výši pohledávky jako P, můžou se vyrovnat nejvýše o $\min(D, P)$. Kdyby si chtěli snad předat víc peněz, buď dojde k předlužení, nebo k přeplacení, což zadání zakazuje.

Jaké je nejmenší množství transakcí, které by teoreticky mohlo proběhnout? Jedna transakce má vždy dvě strany, ovlivní tedy dva kupce. Jednou transakcí tedy dojde k vynulování nejvýše dvou kupců. Je-li tedy kupců celkem N, je nejmenší množství transakcí k vyrovnání $\lceil \frac{N}{2} \rceil$.

Zadání nám tedy dovoluje provést N transakcí. To je ale velmi milé, protože každou transakcí dokážeme alespoň jednoho kupce vynulovat. Stačí když převedeme nejvyšší možnou částku: pokud $\min(D, P) = D$, právě jsme vynulovali dlužníka; pokud $\min(D, P) = P$, vynulovali jsme věřitele.

Rýsuje se nám tedy jednoduchý algoritmus:

Nejprve si spočítáme, kolik měl každý zaplatit, a rozdělíme vstup na dluhy a pohledávky, to vše v čase ($\mathcal{O}(N)$). Poté, dokud budou nějaké pohledávky a dluhy zbývat, vybereme libovolný dluh a libovolnou pohledávku (třeba první ze seznamu) a převedeme maximální možnou částku. Když už žádné pohledávky a dluhy nejsou, máme hotovo a můžeme se jít klouzat.

Proč to funguje? Každým převodem vynulujeme alespoň jednoho kupce, tedy s konečným množstvím převodů budou vynulováni všichni. Též z toho plyne, že převodů provedeme nejvýše $N - 1$, takže jsme splnili zadání.

Jak je to rychlé? Předzpracování trvá $\mathcal{O}(N)$. Každý krok pak trvá konstatní čas: výběr dluhu, výběr pohledávky i samotné vyrovnání.

Paměťová složitost zahrnuje uložení několika proměnných, vstupu velikosti N a dvou seznamů o celkové délce taktéž N, celkem tedy $\mathcal{O}(N)$.

Někteří řešitelé tvrdili, že je potřeba vstup setřídit, nedokázali ale přijít s žádným rozumným argumentem, proč by to mělo být potřeba. Je to lákavé, ale můžeme netřídit a díky tomu si nezavléct do složitosti logaritmus ($\mathcal{O}(N \log N)$ místo $\mathcal{O}(N)$, a to za to stojí, ne?

Jiní se mě zase snažili přesvědčit, že po každé transakci musíme proběhnout celé pole a vyházet z něj ty, kdo už platili. To však není vůbec potřeba. Vždyť jediné dva, kterých se to týká, jsme právě měli v ruce. Takovým postupem se dalo dosáhnout až (pro tuto úlohu jistě impozantní) složitosti $O(N^2)$.

Též se objevilo několik řešení, kde řešitelé porušovali podmínky dané v zadání, například vesele přepláceli, případně si dokonce pořídili banku, která všechno zprostředkovala. Někteří se tohoto prohřešku dopustili skrytě, jiní to dokonce drze deklarovali. Pro odvahu postavit se organizátorům a hrdě řešit jinou úlohu jsem však neměl valného pochopení a uděloval pouze body útěchy.

Úlohu jsme zadali úmyslně s požadavkem na maximálně dvojnásobný počet převodů. Důvod je prostý. Optimální řešení totiž získáme tak, že bychom našli rozdělení množiny kupců do co nejvíce podmnožin tak, aby součet v každé z podmnožin byl nulový. Nicméně už jenom otázka, jestli vůbec existuje podmnožina, jejíž součet je nulový, je NP-úplný problém (anglicky Subset Sum Problem).[6] Jistě pochopíte, že po vás nemůžeme chtít vymyslet řešení takové úlohy v polynomiálním čase.

Jan „Moskyto“ Matějka

29-1-3 Střídání zbraní

Na začátku si seřadíme gardisty podle výšky od nejnižšího po nejvyššího. Představme si, že si v tomto pořadí chodí vybrat kopí. Označme si m_i počet kopí kratších než i-tý gardista (počítáme od nuly). První gardista si může vybrat ze všech kopí, které je schopen používat, má tedy m_0 možností.

Druhý gardista je schopen používat m_1 různých kopí. Ale vybrat si může pouze z $m_1 - 1$, protože jedno z nich už si vzal první gardista (první gardista je menší než druhý, tedy kopí, které si vzal, by mohl používat i druhý, ten tak o jednu možnost přišel).

Analogicky třetí gardista dokáže používat m_2 kopí, ale dvě z nich už si zabrali první dva gardisté, má tedy na výběr z $m_2 - 2$ možností. Obecně i-tý gardista si může vybrat z $m_i - i$ kopí.

Počet možností, které má i-tý gradista, nezávisí na tom, jaká kopí si vybrali předchozí. Tady je vidět, proč je důležité gardisty brát v pořadí od nejmenšího. Kdyby byl první gardista vyšší než druhý, počet kopí, které zbudou na druhého gardistu, by závisel na tom, jak vysoké kopí si vzal první.

Takhle víme, že pro každou z m_0 možností výběru prvního gardisty existuje právě $m_1 - 1$ možností volby druhého. Celkem tedy je $m_0 \cdot (m_1 - 1)$ možností, jak si může vybrat první dvojice.

Obecně když provádíme několik výběrů po sobě a počet možností, ze kterých v každém kroku vybíráme, nezávisí na volbách v předchozích krocích, je celkový počet možností prostě součinem počtů možností v jednotlivých krocích. Tomuto principu se říká *pravidlo kombinatorického součinu*.

Dostáme tedy celkem

$$m_0 \cdot (m_1 - 1) \cdot (m_2 - 2) \cdot \ldots \cdot (m_N - N)$$

(kde N je počet gardistů) možností, jak si gardisté mohou rozdělit kopí.

[6] `https://en.wikipedia.org/wiki/Subset_sum_problem`

Dlužno podotknout, že celkový počet možností samozřejmě nezávisí na tom, v jakém pořadí si gardisté vybírají kopí. Jen když si představujeme, že si vybírají v pořadí od nejmenšího, je o dost snažší možnosti spočítat.

Teď v podstatě stačí dosadit do vzorečku. K tomu bychom ale nejdřív potřebovali spočítat jednotlivá m_i. To je celkem jednoduché. Na začátku si kromě gardistů i kopí seřadíme vzestupně podle délky. Teď budeme postupně procházet jednotlivé gardisty a průběžně si udržovat index největšího kopí menšího než aktuální gardista (tento index odpovídá právě m_i).

Když přejdeme na následujícího gradistu, tento index zvyšujeme, dokud nenarazíme na kopí vyšší než on. Takto spočítáme všechna m_i jedním průchodem přes obě pole v lineárním čase. Podrobněji ve vzorovém programu. Jde o podobný princip jako při slévání setříděných posloupností.

Celková časová složitost je $\mathcal{O}(N \log(N))$, protože tolik času strávíme tříděním a zbytek stihneme v lineárním čase. Paměťová složitost bude lineární.

Program (Python 3):
`http://ksp.mff.cuni.cz/viz/29-1-3.py`

Většina z vás na základní myšlenku přisla. Co jste často opomíjeli, je zdůvodnit, proč řešení funguje. Z výše uvedeného vzorečku není bez komentáře vůbec vidět, že by měl platit. Po přečtení spousty řešení nebylo ani jasné, proč vlastně vstup třídí, když se poté nikdy nezmínily, k čemu to využijí, či dokonce že bez třídění by vzoreček neplatil. Zdůvodnění jsme tentokrát považovali za poměrně důležitou součást úlohy (už proto, že algoritmus samotný je poměrně jednoduchý) a patřičně strhávali body.

Také jste se málokdy zamysleli, co se stane, když neexistuje žádná možnost, jak gardistům kopí rozdat (například některá kopí jsou vyšší než všichni gardisté). V takovém případě se může stát, že některé ze závorek $m_i - i$ vyjdou nulové či dokonce záporné.

V tomhle případě platí, že pokud některá z nich vyjde záporná, bude mezi nimi i nějaká nulová (rozmyslete si), tedy dosazením do vzorečku dostanete správný výsledek: nulu. Ale není to vůbec zřejmé a slušelo by se nad tím v řešení zamyslet.

Ještě se zmíníme o jednom detailu, který je asi přijatelné na úrovni KSP příliš neřešit: výsledný počet možností může být velmi velké číslo. Počítáme součin N čísel velkých až N (v nejhorším případě jsou všechna kopí kratší než všichni gardisté), tedy výsledek může být velký řádově až N^N, což se kromě nejmenších vstupů do běžných celočíselných typů nevejde.

S tím se musíme nějak vypořádat v závislosti na tom, čeho přesně chceme dosáhnout. Pokud by nám výsledek například stačil přibližně či řádově, nejjednodušším řešením je použít nějaký typ s plovoucí čárkou (float, double), který umí uchovávat velmi velká čísla, byť s omezenou přesností, a zacházet s nimi v konstantním čase a prostoru.

Druhou možností je počítat s velkými čísly poctivě přesně. Ale to už obecně nezvládneme v konstantním čase: číslo velké řádově K zabere v paměti místo $L = \mathcal{O}(\log K)$ a vynásobení dvou takovýchto čísel zvládneme poměrně jednoduchým algoritmem v čase zhruba $\mathcal{O}(L^{1.5})$ (jde to i rychleji). V našem případě $L = \mathcal{O}(\log N^N) = \mathcal{O}(N \log N)$, tedy jedno násobení stihneme v čase $\mathcal{O}(N^{1.5} \log^{1.5} N)$, počítáme N součinů, takže celková časová složitost vzroste na $\mathcal{O}(N^{2.5} \log^{1.5} N)$, paměťová na $\mathcal{O}(N \log N)$.

Filip Štědronský

29-1-4 Zběsilý útěk

Úloha byla velmi jednoduchá, až triviální. Prostě pro každou úsečku projdu všechny hustolesy, zjistím, se kterým se protíná, spočítám dráhu, kterou urazím hustolesem a kterou řídkolesem, obojí vydělím příslušnou rychlostí a výsledné časy sečtu.

Takové řešení má časovou složitost $\mathcal{O}(U \cdot H)$, kde U je počet úseček a H je počet hustolesů. Tyto hodnoty mají být řádově stejné, aneb $U = \mathcal{O}(N)$, $H = \mathcal{O}(N)$, celková složitost je tedy $\mathcal{O}(N^2)$. Není to moc? Je to moc. Áha. Úloha nebyla tak jednoduchá a triviální, jak by se na první pohled mohla zdát. Přesto i na triviálním řešení šlo nasbírat nemálo bodů.

Připomeňme si techniku zametání roviny přímkou.[7] Projdeme celou oblast například zdola nahoru a poznamenáme si, kde začíná a kde končí který hustoles. Na to si musíme seznam hustolesů setřídit, což zabere $O(H \log H)$. Seznam začátků a konců hustolesů si označíme jako SZKH.

Tahle metoda se nám hodí při procházení lesem. Na začátku stojím v bodě y_0, spočítám si, které hustolesy procházejí přímkou o souřadnici $y = y_0$, a budu tedy znát jejich seznam. Bude se hodit, aby byl setříděný, zabere nám to jen $O(H \log H)$.

Pak posunu přímku do y_1. Abych ji nemusel počítat celou odznova, podívám se do SZKH a zpracuju všechny události, které jsou mezi y_0 a y_1. A celou dobu se přitom dívám, jestli se na přímce náhodou neobjevil nějaký hustoles se souřadnicí mezi x_0 a x_1, a pokud ano, tak jej hned podrobím zkoumání, jestli náhodou nebyl proťat právě zpracovávanou úsečkou.

Pak posunu stejným způsobem přímku do y_2 (a zpracovávám druhou úsečku), pak do y_3 … Postupně tak zpracuju všechny úsečky na vstupu.

Implementačně se pro účely ukládání stavu na přímce bude patrně hodit použít nějaký vyvažovaný vyhledávací strom, třeba červenočerný. Složitost zpracování každé úsečky pak bude $\mathcal{O}(K \log H)$, kde K je počet kroků, které je potřeba udělat.

Tím jsme si ale vůbec nepomohli, protože úsečka zrovna mohla vést z levého okraje území na pravý, takže stejně musíme vyzkoušet všechny hustolesy, i když víme, že protnout má nejvýše jeden.

Neuvědomili jsme si ale, že vůbec nemusíme zkoušet všechny hustolesy mezi x_0 a x_1, ale často jen malý úsek. Nechť aktuální vodorovná přímka má souřadnici y_α a nejbližší následující má souřadnici y_β. Zkoumaná úsečka mezi body (x_0, y_0) a (x_1, y_1) protne tyto dvě přímky na souřadnicích $x_\alpha = x_0 + \frac{(x_1 - x_0)(y_\alpha - y_0)}{y_1 - y_0}$ a $x_\beta = x_0 + \frac{(x_1 - x_0)(y_\beta - y_0)}{y_1 - y_0}$.
Co se stane v obdélníku, který jsme si vytyčili souřadnicemi $(x_\alpha, y_\alpha, x_\beta, y_\beta)$? Především uvnitř něj nemůže ležet žádná vodorovná hrana žádného hustolesa. Kdyby v našem obdélníku snad chtěla ležet horní nebo dolní hrana nějakého

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

hustolesa, musela by být vložena též v SZKH, ale pak bychom se naším obdélníkem vůbec nemohli zabývat, protože mezi y_α a y_β by ještě existovalo nějaké y_υ, takže by y_α a y_β nebyly hned po sobě jdoucí polohy zametací přímky, tedy máme spor, kterým jsme dokázali, že tento obdélník je horních a dolních hran hustolesů zcela prost.

Pokud tedy nějaký hustoles v obdélníku leží, vždy obsáhne celou jeho výšku, a tedy se vždy musí protnout s úsečkou jdoucí po jeho úhlopříčce. Stejně tak obráceně, pokud nějaký hustoles v obdélníku neleží, tak se rozhodně mezi souřadnicemi y_α a y_β nemůže protnout se zkoumanou úsečkou.

Kromě hustolesů tedy budeme mít na naší zametací přímce ještě bod, který bude označovat průsečík zkoumané úsečky s aktuální přímkou, a při každém posunutí přímky s ním pohneme na správné místo a zkontrolujeme, jestli jsme tím náhodou netrefili hustoles.

Co když ale úsečka vede shora dolů, takže musíme projít všechny polohy zametací přímky? Jednou by to nevadilo, ale mohly by takto vést třeba všechny úsečky a pak bychom si se složitostí vůbec nepomohli. Stále totiž každý posun zametací přímky zabere $\mathcal{O}(\log H)$ času, takže bychom si zhoršili složitost na $\mathcal{O}(U \cdot H \log H)$. To nám je platné jak mrtvému zimník.

Všimneme si tedy, že při posouvání zametací přímky semtam počítáme pořád dokola totéž – její stav.

Pořídíme si tedy datovou strukturu, která dokáže nějak rozumně uložit všechny možné polohy zametací přímky, a zároveň nebude přehnaně velká. Pokud jsme ochotni obětovat až $\mathcal{O}(H^2)$ paměti, stačí si postupně uložit všechny stavy zametací přímky. Jenomže na vygenerování $\mathcal{O}(H^2)$ dat potřebujeme také $\mathcal{O}(H^2)$ času, takže jsme zase nahraní.

Ne tak docela. Ukládáním všech stavů zametací přímky bychom zase ledacos ukládali redundantně, takže si pořídíme nějaký vhodný druh komprese. Každý hustoles budeme reprezentovat dvěma seznamy ukazatelů: seznamem levých sousedů a pravých sousedů. Seznam levých sousedů říká pro každý interval souřadnice y nejbližší sousední hustoles směrem vlevo. Analogicky vypadá seznam pravých sousedů.

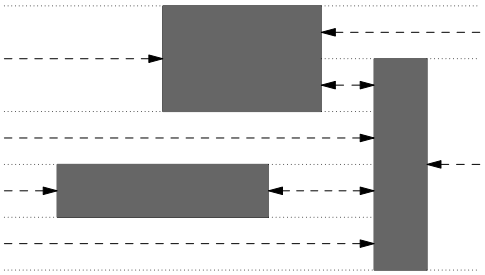
Seznamy postavíme tak, že projdeme zametací přímkou přes všechny události v SZKH. Na začátku není v oblasti žádný hustoles, ale vyrobíme si dva virtuální se souřadnicemi $x_\lambda = -\infty$, $x_\pi = \infty$.

Když potkáme událost „začátek hustolesa“, podíváme se, *mezi které dva* jiné hustolesy jej máme přidat. (Vždycky tam budou alespoň ty dva virtuální.) Do seznamu levých sousedů pravého hustolesa a do seznamu pravých sousedů levého hustolesa tedy přidáme odkaz na právě přidávaný hustoles, do seznamu levých sousedů právě přidávaného hustolesa vložíme odkaz na levý hustoles a do seznamu pravých sousedů právě přidávaného hustolesa vložíme odkaz na pravý hustoles.

Když potkáme událost „konec hustolesa“, podíváme se taktéž, mezi kterými dvěma jinými hustolesy byl. Seznamy levých a pravých sousedů odebíraného hustolesa uzavřeme, do seznamu levých sousedů pravého hustolesa přidáme levý hustoles a do seznamu pravých sousedů levého hustolesa přidáme pravý hustoles.

Jak dlouho to trvá? Každý posun zametací přímkou potřebuje $\mathcal{O}(\log H)$ času kvůli aktualizaci jejího stavu. K tomu vyrobíme 4 nebo 2 nové odkazy, což je konstanta, která logaritmem nehne. Celkem nás bude vytvoření datové struktury stát $\mathcal{O}(H \log H)$ času, což je stále stejně jako počáteční potřeba setřídění.

Jak velká bude zkonstruovaná datová struktura? To se dá spočítat z algoritmu, kterým ji vyrábíme. Každý hustoles vygeneruje v SZKH jednu událost začátku a jednu událost konce hustolesa. Zpracováním těchto dvou událostí vznikne v datové struktuře právě 6 nových odkazů. Datová struktura tedy spotřebuje celkem $\mathcal{O}(H)$ paměti. To vypadá nadějně.



Na vstupech, které generovalo naše odevzdávátko, už tato úprava běžela dostatečně rychle na získání plného počtu bodů, protože drtivá většina zadaných úseček byla v porovnání s velikostí oblasti velmi krátká. Při zpracování každé úsečky totiž stačilo projít několik málo odkazů – prázdných oblastí, přes které zrovna procházela, i když byla třeba poměrně dlouhá.

Stále se nám sice může stát, že budeme při zpracování každé úsečky projít přes $\mathcal{O}(H)$ odkazů, takže jsme si pomohli zpátky na $\mathcal{O}(U \cdot H)$ v nejhorším případě.

Bylo by fér spočítat, že v průměrném případě na tom budeme líp, nicméně počítat statistiku nad úsečkami a obdélníky v rovině je poněkud ošemetné, jak naznačuje kupř. Bertrandův paradox, tak si to raději odpustíme.

Program s řešením v čase $\mathcal{O}(N^2)$ (C):
`http://ksp.mff.cuni.cz/viz/29-1-4-slow.c`

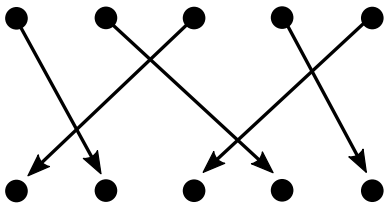
Program s rychlejším řešením v čase až $\mathcal{O}(N^2 \log N)$ (C):
`http://ksp.mff.cuni.cz/viz/29-1-4-fast.c`

Jan „Moskyto“ Matějka

29-1-5 Lučištníci

Na vstupu mějme N lučištníků. Zleva doprava si je očíslujme od 1 do N, stejně jako cíle, na které se míří. Platí, že cíl s číslem k se nachází naproti střelci k. Jako c_k si označme číslo cíle lučištníka k.

Mnoho z vás se snažilo najít nejlepší řešení tímto způsobem: pro každého lučištníka našlo počet drah, s nimiž se ta jeho kříží, a následně postupně odstraňovalo lučištníky s největším možným počtem křížení. Takový postup ale obecně nefungoval. Například v zadání na obrázku byste mohli odstranit i druhého lučištníka, který do správného řešení očividně patří, ačkoliv se jeho dráha kříží se dvěma dalšími drahami.



Uvedeme si řešení běžící v čase $\mathcal{O}(N^2)$ a to následně zlepšíme. Zaveďme pojem *nekřížící se skupina* pro takovou podmnožinu střelců, že jejich dráhy se navzájem nekříží.

Pro každého lučištníka k teď chceme najít co největší nekřížící se skupinu A_k, pro níž platí, že k do ní náleží (tj. bude střílet) a jinak se skládá jen ze střelců nalevo od k. Všimněte si, že v této skupině míří na nejpravější cíl právě lučištník k.

Projdeme teď lučištníky zleva a pro každého budeme chtít spočítat číslo d_k, což je velikost skupiny A_k.

Jistě $A_1 = \{1\}$ a tedy $d_1 = 1$. Pro $k > 1$ vytvoříme skupinu A_k tak, že prodloužíme nějakou předešlou skupinu (každá větší skupina je nutně rozšířením nějaké předešlé skupiny: stačí si uvědomit, že odebráním střelce nejvíce napravo dostaneme kratší skupinu). Jistě chceme vzít takovou s co největším počtem střelců; stále ale musí platit, že se dráhy nekříží. Pokud dáme dohromady všechna pravidla, vyjde nám, že $d_k = \max d_i + 1$, kde $i < c$ a $c_i < c_k$. Právě druhé pravidlo zajistí, že dráha k-tého střelce není v konfliktu s ostatními drahami.

Po doběhnutí cyklu pak nejvyšší d_k označuje maximální počet lučištníků, kteří mohou střílet najednou. Pokud chceme znát konkrétní čísla střelců, stačí algoritmus jednoduše rozšířit: vždy po vypočtení nového d_k si zapamatujeme číslo střelce, rozšířením jehož skupiny vznikla A_k. Na konci cyklu pak najdeme nejvyšší d_k a od něj tyto zpětné odkazy projdeme a vypíšeme.

Zbývá určit složitost. U k-tého lučištníka musíme projít všech $k-1$ předchůdců, dohromady nám to tedy zabere $\mathcal{O}(1 + 2 + \ldots + (N-1)) = \mathcal{O}(\frac{N(N+1)}{2}) = \mathcal{O}(N^2)$ času. Paměťová složitost je $\mathcal{O}(N)$, u každého střelce si ukládáme pouze konstantní počet hodnot.

Existuje však lepší řešení. Nejprve si pro jakoukoliv nekřížící se skupinu lučištníků definujme *pravý okraj*, což je číslo cíle, kam míří lučištník nejvíce napravo. To je zároveň nejvyšší c_i skupiny.

Může se stát, že máme více stejně velkých skupin, a chceme je rozšířit o lučištníka, který stojí napravo od ostatních střelců ve skupinách. Pak upřednostňujeme skupinu s nejnižším pravým okrajem (c_k nového lučištníka musí být větší, než c_k ostatních).

Zaveďme posloupnost $m_i, i \in \{0, \ldots, N\}$. Pokud si vezmeme všechny nekřížící se skupiny velikosti i, m_i bude nejmenší z jejich pravý okrajů. V případě, že skupina velikosti i neexistuje, pak $m_i = \infty$. Platí, že $m_{i-1} \leq m_i$: rozmyslete si, že ze skupiny s i střelci lze vytvořit skupinu s $i-1$ střelci takovou, že pravý okraj zůstane stejný. Pokud pro zadané lučištníky a jejich terče dokážeme vypočítat posloupnost m_i, je maximálním počtem lučištníků nejvyšší takové i, že $m_i < \infty$.

A jak tedy m_i získat? Opět projdeme střelce zleva doprava; na začátku zvolíme $m_0 = -1$, pro $k > 0$ bude $m_k = \infty$. Pro k-tého střelce potom vylepšíme posloupnost takto: protože posloupnost m_i je celou dobu neklesající, dokážeme najít takové l, že $m_l < c_k \leq m_{l+1}$.

Všimněte si, že přidáním k-tého lučištníka nevylepšíme m_i, kde $i \leq l$: pro skupiny velikosti i jsme už nalezli menší pravé okraje. Můžeme však zlepšit m_{l+1}: představte si, že ze skupiny o $l+1$ lučištnících s pravým okrajem m_{l+1} odstraníme střelce nejvíce napravo a přidáme k-tého lučištníka – tím m_{l+1} snížíme, nebo necháme nezměněné. Pro $i > l+1$ toto udělat nemůžeme, museli bychom odstranit dva a více lučištníků a velikost skupiny by se změnila.

Ačkoliv i zde uděláme N kroků, najít správné l lze pomocí binárního vyhledávání, následně už jen upravíme m_{l+1}. Celkový čas je tedy $\mathcal{O}(N \log N)$.

Na závěr důležité pozorování: pokud $c_0, c_1, \ldots, c_N$ zapíšeme jako posloupnost, odpovídá nalezené řešení nejdelší rostoucí podposloupnosti v c_k. Není těžké převést tyto úlohy mezi sebou: ostatně většina řešitelů s plným počtem bodů si této spojitosti všimla.

Kuba Maroušek

Program s řešením v $\mathcal{O}(N^2)$ (Python 3):
`http://ksp.mff.cuni.cz/viz/29-1-5-pomale.py`

Program s řešením v $\mathcal{O}(N \log N)$ (Python 3):
`http://ksp.mff.cuni.cz/viz/29-1-5-rychle.py`

29-1-6 Štítové kouzlo

Nabízí se triviální algoritmus, který vyzkouší všechny možnosti. Jenomže toto nebude fungovat vůbec rychle. Pojďme se podívat proč.

Jednotlivé kroky rozšiřování štítu můžeme zapsat do posloupnosti $\{L, P\}^{n-1}$, kde n je počet všech hradeb. Nechť pak v této posloupnosti L reprezentuje rozšíření štítu vlevo a P rozšíření štítu vpravo. V této definici můžeme jednoduše získat počáteční úsek hradby spočítáním všech L (musíme začít dostatečně vpravo, aby rozšiřovaní vlevo mělo smysl), takto každá posloupnost jednoznačně určuje postup rozšiřování štítu.

Počet všech takových posloupností činí 2^{n-1}. Vyzkoušení všech možností by nám tedy trvalo vždy $\Omega(2^n)$ času. Jak se z toho vybrodit?

Hladové řešení nefunguje. Vybírat vždy nejvyšší úsek hradby je již vyvráceno příkladem v zadání. Stejně tak nás zradí rozšiřování směrem, kde je větší součet stojících hradeb. Ukažme si to na jednoduchém příkladu. Mějme následující hradby:

$$10\ 10\ \overline{10}\ 0\ 0\ 0\ 0\ 0\ 100$$

Zde se hladový algoritmus rozhodne rozšiřovat štít vpravo. Jenže než se dostane k nejpravějšímu úseku, oba úseky nalevo ztratí příliš mnoho hodnoty. Nakonec zachrání jen $(100-5)+(10-6)+(10-7) = 102$ hradeb. Kdybychom nejprve ochránili část hradeb vlevo a až potom se vydali vpravo, zachráníme jich $10+9+(100-7) = 112$, tedy mnohem více.

Hlavní potíž při zkoumání všech možností spočívá ve skutečnosti, že takto spoustu společných postupů mnohokrát zbytečně znovu počítáme. Třeba tyto dvě varianty aktuálních štítů $\overline{A\ B\ C\ D}\ E$ a $A\ \overline{B\ C\ D\ E}$ vychází ze štítu $A\ \overline{B\ C\ D}\ E$, který při zkoumání všech možností celý přepočítáme dvakrát. Takový problém řeší princip *dynamického programování*.[8]

Označme $\check{S}(a,b)$ nejlepší možný součet výšek hradeb v úseku od a do b, který chrání náš štít v kroce $k = b-a$. Dále si označme $V(k,i)$ výšku i-tého nechráněného úseku hradby v k-tém kroce, jejíž hodnota se bude rovnat $\max(0, h[i]-k)$.

Všimněme si, že $\check{S}(a,b)$ se rovná buďto $\check{S}(a,b-1)+V(k,b)$, anebo $\check{S}(a+1,b)+V(k,a)$, podle té z možností poskytující vyšší součet chráněných hradeb. Tedy vzorec pro $\check{S}(a,b)$ je $\max\left\{\check{S}(a,b-1)+V(k,b), \check{S}(a+1,b)+V(k,a)\right\}$.

Nyní můžeme spočítat $\check{S}(0,n-1)$, jenž odpovídá optimálnímu rozšíření štítu nad celou hradbou. Můžeme postupovat

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

podle definice rekurzí, kde si již spočítaná $\check{S}(a,b)$ zapamatujeme do tabulky, abychom je nepočítali znova.

Můžeme postupovat i jiným způsobem, a to spočítat všechna $\check{S}(a,a+k)$ iterativně po „vrstvách“, tedy nejprve všechna $\check{S}(a,a)$, potom všechna $\check{S}(a,a+1)$, …, až nakonec získáme hledané $\check{S}(0,n-1)$. Při takovém způsobu počítání nám stačí si pamatovat předchozí vrstvu, tabulka tedy není nutná.

Jestliže bychom navíc chtěli znát postup, jak jsme štít rozšiřovali, budeme si kromě $\check{S}(a,b)$ pamatovat i $D(a,b)$ říkající, přidáním kterého prvku (nebo ze kterého směru) byl nejlepší štít mezi a a b rozšířen. Projitím $D(0,n-1)$ potom získáme postup rozšiřování štítu v opačném pořadí.

Nyní stačí spočítat časovou a paměťovou složitost tohoto algoritmu. Všech dvojic a,b je $\mathcal{O}(n^2)$, každou spočítáme nejvýše jednou. Na vypočítání každého $\check{S}(a,b)$ spotřebujeme konstantní čas. Celková časová složitost je tedy $\mathcal{O}(n^2)$. Jestliže nám stačí znát pouze nejlepší součet zachráněných hradeb, vystačíme si s $\mathcal{O}(n)$ paměti při použití iterativní metody. Jinak je paměťová složitost shodná s časovou složitostí $\mathcal{O}(n^2)$.

Program (C):
`http://ksp.mff.cuni.cz/viz/29-1-6.c`

Václav Končický

29-1-7 Stromy kolem nás

Úkol 1: Příklady stromů

Děkujeme za pěkné příklady stromů: rodokmen (já a všichni mí biologičtí předkové), řeka se svými přítoky, adresářová struktura na disku, nebo třeba všechny možnosti, jak se může vyvíjet partie deskové hry. Přidáme k nim strukturu webových stránek (do sebe zanořené elementy HTML), programů v programovacích jazycích a nádavkem ještě vztahy mezi větnými členy či větami v souvětí. Stačí? :)

Úkol 2: Strom z postfixového výpisu

Úkol byl formulován trochu nepořádně: pokud o vrcholech stromu vůbec nic nevíme, tvar stromu se z jeho postfixového výpisu určit nedá. Pokud nám ale někdo řekne, kolik má mít který vrchol synů, už to možné je. U stromů výrazů je tato podmínka triviálně splněna: čísla jsou listy, operace mají právě dva syny. Pro jednoduchost budeme nadále předpokládat, že pracujeme se stromem výrazu.

Postfixový zápis budeme procházet zleva doprava a postupně ho překládat na stromy: pokud jsme z `1234++*` přečetli `1234+`, máme zatím hotové jednovrcholové stromy `1` a `2` a třívrcholový strom s kořenem `+`, pod nímž jsou listy `3` a `4`. Tyto stromy postupně slepíme do jednoho velkého, ale z toho, co jsme zatím přečetli, dosud není jasné jak.

Budeme si tedy pamatovat nějakou posloupnost rozpracovaných stromů, říkejme jí *alej*. Kdykoliv ze vstupu přečteme číslo, vytvoříme jednovrcholový strom s tímto číslem a přidáme ho na konec aleje. Přečteme-li naopak nějakou operaci, odpojíme z konce aleje dva stromy, spojíme je pod nově založený kořen s danou operací a výsledek opět zapíšeme na konec aleje. Obecně objeví-li se na vstupu zápis vrcholu stupně s, spojujeme posledních s stromů z aleje.

Časová složitost tohoto algoritmu je zřejmě lineární. Kdybychom chtěli poctivě dokázat korektnost, pak třeba takto: Uvažme průběh prohledávání stromu do hloubky. Právě stojíme v nějakém vrcholu v a chystáme se ho opustit, tedy do postfixového zápisu toto v vypsat. Vrcholy na cestě z kořene do v jsme už objevili, ale na vypsání teprve čekají. Vše vlevo od této cesty jsme už vypsali, vše napravo naopak ani neobjevili.

O dekódovacím algoritmu pak platí následující invariant: při zpracování v se v aleji nachází posloupnost podstromů visících vlevo od cesty do v (v pořadí shora dolů) následovaných podstromy ležícími pod v (zleva doprava). Jakmile algoritmus uvidí v, podstromy správně poslepuje a invariant bude nadále platit.

Za odměnu vám ukážeme ještě jedno, mírně magické řešení: Postfixový zápis otočíme, čímž se z něj stane prefixový (až na opačné pořadí synů). Ten můžeme poskládat do stromu jednoduchým rekurzivním algoritmem: pokud narazíme na číslo, vytvoříme z něj jednovrcholový strom a skončíme. Pokud na operaci, dvakrát se rekurzivně zavoláme a získané dva stromy spojíme pod společný kořen. Hotovo, $\mathcal{O}(n)$. Formální důkaz složitosti by se vedl stejně jako u minulého algoritmu.

Úkol 3: Nejednoznačný infixový zápis

Prosté, milý Watsone: stačí uvážit zápis `1+2+3`. Ten můžeme uzávorkovat dvěma způsoby: `(1+2)+3` a `1+(2+3)`. Pokaždé vyjde jiný strom. Prefixově tyto stromy zapíšeme `++123` a `+1+23`, postfixově `12+3+` a `123++` – vše bez závorek, z předchozího úkolu už přeci víme, že nejsou třeba.

Úkol 4: Průměr stromu

Jak zadání naznačuje, k měření délky nejdelší cesty se skutečně bude hodit něco počítat během DFS. Kdykoliv se budeme vracet z podstromu s kořenem v, spočítáme dvě čísla: $h(v)$ udávající hloubku podstromu a $\ell(v)$ – maximální délku cesty v podstromu.

Výpočet $h(v)$ už známe ze zadání: v listu platí $h(v)=0$, ve vnitřním vrcholu se syny $s_1,\ldots,s_k$ musí být $h(v)=\max(h(s_1),\ldots,h(s_k))+1$.

Jak tedy s $\ell(v)$? Rozmysleme si možné polohy vrcholu v vzhledem k této cestě:

1. v je list – tehdy je celý podstrom jednovrcholový, pročež $\ell(v)=0$
2. v vůbec na cestě neleží – tehdy jsme cestu již započítali do některé z délek $\ell(s_1)$ až $\ell(s_k)$.
3. v je koncovým vrcholem cesty – cesta vede z v dolů do nejvzdálenějšího listu, takže měří $h(v)$.
4. v je „zlomem“ cesty – cesta přichází zespoda z některého s_i a zase odchází dolů do nějakého jiného s_j. První část určitě vede z listu, takže měří $h(s_i)$, pak následuje hrana s_iv, hrana vs_j a závěrečná část do listu, délky $h(s_j)$. Vrcholy s_i a s_j samozřejmě volíme tak, abychom použili největší a druhé největší $h(s_i)$.

Pro vrcholy s jedním synem může nastat druhá a třetí možnost, tedy $\ell(v)=\max(\ell(s_1),h(v)+1)$. Je-li synů více, třetí možnost je vždy horší než čtvrtá, takže spočítáme $\ell(v)=\max(\ell(s_1),\ldots,\ell(s_k),m_1+m_2+2)$, kde m_1 a m_2 je první a druhé největší $h(s_i)$.

Nejdelší cestu v celém stromu pak najdeme v $\ell(v)$ kořene.

Našimi výpočty strávíme v každém vrcholu lineární čas s počtem synů, což odpovídá složitosti samotného DFS. Celý algoritmus tedy poběží v lineárním čase.

Průměr stromu podruhé

Předveďme ještě jeden způsob, jak změřit nejdelší cestu. Strom zakořeníme v libovolném vrcholu a. Pak najdeme

nejhlubší vrchol v (to už umíme jedním DFS). Tento vrchol prohlásíme za nový kořen a opět najdeme nejhlubší vrchol w. Tvrdíme, že cesta vw je jedna z nejdelších cest.

Tento algoritmus je evidentně lineární, ale není zřejmé, jestli funguje. Pojďme to dokázat sporem: předpokládejme, že existují nějaké stromy, na nichž algoritmus nefunguje. Vyberme z nich nějaký strom T s nejmenším počtem vrcholů (tomu se obvykle říká *minimální protipříklad*). T má alespoň 3 vrcholy – menší stromy algoritmus jistě zvládne.

Bez újmy na obecnosti můžeme předpokládat, že vrchol a není list – v opačném případě místo a vybereme jeho souseda, čímž jsme chování algoritmu nezměnili.

Nyní si všimneme, že v i w jsou listy (jinak by šlo cestu do nich ještě prodloužit). Sousedé těchto vrcholů, označme si je v' a w', listy určitě nejsou.

Proto si ze stromu T sestrojíme strom T' otrháním všech listů. V T' určitě leží vrcholy a, v' a w'. A jelikož přes listy žádné cesty nevedou, náš algoritmus spuštěný v T' z vrcholu a dojde nejprve do v' a pak do w'. Jelikož T byl minimální protipříklad, v T' musí algoritmus vydat správný výsledek, takže cesta $v'w'$ je nejdelší.

Vezměme nyní nějakou nejdelší cestu P ve stromu T. Ta určitě vede z listu do listu, takže odtržením listů získáme nějakou cestu v T', jež jistě nemůže být delší než nejdelší cesta $v'w'$. Proto pro délky cest platí $|P| \leq 2+|v'w'|$. Pravá strana nerovnosti je ovšem rovna délce cesty vw, takže vw je také nejdelší. Hotovo.

Úkol 5: Strážníci s drony

Mějme nějaký podstrom s kořenem v a přemýšlejme, jak může být hlídaný. Buďto k ohlídání všech jeho hran postačí strážníci, které jsme rozmístili uvnitř podstromu – tehdy mu budeme říkat *samostatný*. Nebo potřebujeme, aby dovnitř dronem doletěl nějaký strážník zvenku (což nutně znamená, že stojí o jednu hladinu nad v; z vyšších pozic by možná doletěl do v, ale už ne dovnitř podstromu) – tehdy hovoříme o hlídání *s výpomocí*.

Navíc u samostatných podstromů potřebujeme rozlišovat, jak vysoko z nich dron vyletí. Tomuto číslu budeme říkat *síla hlídání* a všimneme si, že se pohybuje od 0 do 2. Sílu poznáme podle toho, na jaké nejvyšší hladině se nachází strážník: strážník v kořeni (na 0. hladině) dává sílu 2, strážník na 1. hladině sílu 1, na 2. hladině sílu 0; od 3. hladiny dál nelze hlídat bez výpomoci. Navíc dodefinujeme sílu -1 pro hlídáni s výpomocí.

Nyní budeme pro každé v počítat čísla $h_i(v)$, kde $i \in \{-1, 0, 1, 2\}$. Budou vyjadřovat minimální cenu za ohlídání síly i. Podobně jako u hlídání bez dronů můžeme tato čísla spočítat při návratech z vrcholů, jen rozbor případů bude trochu chlupatější.

Nejprve uvažujme ohlídání síly 2. Tehdy v kořeni leží strážník (snad bychom měli říci, že stojí, ale dron se dá jistě ovládat i vleže), takže musíme započítat cenu $c(v)$ za umístění strážníka. Podstromy ležící pod jednotlivými syny kořene pak mohou být ohlídané libovolně silně včetně síly -1. Platí tedy:

$$h_2(v) = c(v) + \sum_s \min_{i \in \{-1,0,1,2\}} h_i(s),$$

přičemž suma sčítá přes všechny syny vrcholu v.

Dobrá, snížíme sílu na 1. V kořeni jistě strážník není, ale musí být v alespoň jednom vrcholu na 1. hladině. Podstrom pod tímto vrcholem má tedy sílu 2, díky čemuž jsou ohlídány i všechny hrany mezi 0. a 1. hladinou (dron do nich doletí). Ve zbývajících podstromech zakořeněných na 1. hladině si tudíž můžeme vybrat libovolnou sílu od 0 do 2 (-1 nelze). Proto:

$$h_1(v) = \min_s \left(h_2(s) + \sum_{s' \neq s} \min_{i \in \{0,1,2\}} h_i(s') \right).$$

Počítat přímo podle tohoto vztahu by ale bylo moc pomalé: zkoušíme všechny dvojice synů a těch může být až kvadraticky mnoho. Proto si spočítáme sumu

$$\sum_s \min_i h_i(s)$$

přes všechny syny s a pak postupně zkusíme každý člen $\min_i h_i(s)$ odečíst a přičíst místo něj $h_2(s)$. To už jde lineárně s počtem synů.

Snižujeme dále: síla 0. Na 0. a 1. hladině nejsou strážníci, takže každá hrana mezi 0. a 1. hladinou musí být hlídaná zespoda (víme, že nevyužíváme výpomoc shora). Proto podstrom pod každým synem s musí být hlídaný silou alespoň 1. Ale nemůže to být ani víc než 1, protože pak by celková síla vyšla 1. Z toho plyne:

$$h_0(v) = \sum_s h_1(s).$$

Konečně zbývá případ se silou -1, tedy s výpomocí shora. Všechny hrany mezi 0. a 1. hladinou jsou hlídané shora a na 0. ani 1. hladině nestojí strážníci (jinak bychom výpomoc nepotřebovali). Vrcholy na 1. hladině tedy mohou být hlídané silou 0 nebo 1. Proto:

$$h_{-1}(v) = \sum_s \min(h_0(s), h_1(s)).$$

To nám dává kompletní recept na výpočet všech $h_i(v)$ během prohledávání do hloubky. V každém vrcholu tím trávíme čas lineární s počtem synů, takže jsme DFS nezpomalili.

Teď už se stačí jen podívat na hodnoty h_0, h_1 a h_2 v kořeni a vzít z nich minimum (h_{-1} neuvažujeme: už nezbývá, kdo by nám vypomohl). Uff.

Úkol 6: Excentricity

Uvažujme nějaký list ℓ a označme jeho souseda s. Nejdelší cesta z ℓ určitě vede přes s, takže excentricita ℓ je o 1 větší než excentricita s. Stačí tedy odstranit všechny listy, stanovit excentricity zbývajících vrcholů a pak dopočítat excentricity listů.

Můžeme snadno upravit algoritmus ze zadání na hledání centra stromu. Při otrhávání listů si zapisujeme, v jakém pořadí jsme je odebírali. Až nám zbude centrum, spočítáme jeho excentricitu (třeba algoritmem na výpočet hloubky stromu spuštěným v původním stromu z centra). Pak listy v opačném pořadí připojujeme zpět a dopočítáváme jim excentricity.

Všechny tři části algoritmu jistě běží v lineárním čase.

Program (Python):
`http://ksp.mff.cuni.cz/viz/29-1-7-ukol6.py`

Martin „Medvěd“ Mareš

Výsledková listina první série dvacátého devátého ročníku KSP

	řešitel	*škola*	*ročník*	*sérií*	*H1-1*	*H1-2*	*H1-3*	*H1-4*	*H1-5*	*H1-6*	*H1-7*	*série*	*celkem*
0.					8	11	12	10	10	10	15	58,0	58,0
1.	Pavel Turek	GTomkovaOL	4	6	8		11	1	10	10	11	52,2	52,2
2.	Richard Hladík	GOAMarLaz	4	21	6	8	12		10	10	15	51,6	51,6
3.	Lukáš Rozsypal	GÚstavníPH	4	4	8		9,5	6	4		15	48,3	48,3
4.	Martin Picek	GJirsíkaČB	2	1	8	11	11		3		9	47,0	47,0
5.	Jakub Pelc	G UherBrod	3	6	8		12	10			15	45,0	45,0
6.	Tomáš Domes	MendelG_OP	4	2	8	6	10			1	15	44,1	44,1
7.	Rajmund Hruška	GPošKošice	4	1	6,5	5	3	2	9		9	43,0	43,0
8.	Matouš Mařík	G_Krumlov	4	1	3	6,5	6,5	1			10	40,7	40,7
9.	Peter Grajcar	GMetodovaBA	3	1	5	6,5	7,5	4	3			40,0	40,0
10.	Filip Geib	G MMH LM	3	3	6	3,5	8,5		3		4	36,9	36,9
11.	Roman Bujdák	G JM Galanta	3	1	6	9,5	6,5		2	1		31,5	31,5
12.	Jonáš Fiala	GJungmanLT	4	6	6	9	9	1				28,8	28,8
13.	Filip Masár	PiarGNitra	3	1	6,5	6,5	8					27,4	27,4
14.	Petr Gebauer	GMělník	3	2	6,5	6,5	7					26,8	26,8
15.	Miroslav Hrabal	GTomkovaOL	3	3	6,5		9,5	0			6	25,6	25,6
16.	Václav Pavlíček	SPSE_Pard	1	6	6,5			1	3	1	8	25,5	25,5
17.	Michal Kodad	SPŠ_Smíchov	1	5		10,5	2,5				6	23,3	23,3
18.	Kristián Jacik	GSRandyJN	4	1	5,5	3	0		1		3	22,6	22,6
19.	Lukáš Caha	GZborovPH	3	1	5,5	3				0,5	2	19,7	19,7
20.	Ondřej Gonzor	G Brandýs	0	1	7	5			2			18,8	18,8
21.	Radek Olšák	MensaG	2	1	5,5		9					18,4	18,4
22.	Pavel Turinský	G Brandýs	4	10	6						11	16,4	16,4
23.	Ondřej Krsička	GJarošeBO	1	1	6,5	1	2		2			16,3	16,3
24.	Anna Řechtáčková	GJarošeBO	4	1	8				5			15,9	15,9
25.	Jindřich Dítě	VOSPŠŽďár	1	1	6,5		6,5					15,6	15,6
26.	Ondřej Cach	SPSE_Pard	1	1	5,5		0,5	2	2			15,4	15,4
27.	Anna Hollmannová	GSRandyJN	0	1	3	3	0		3			13,4	13,4
28.	Daniel Skýpala	GTomkovaOL	−1	1	8				3			12,5	12,5
29.	Vojtěch Hudec	G_ČTřebová	3	3	5,5		1		2			12,1	12,1
30.	Vojtěch Lengál	GZborovPH	3	1			8,5					11,0	11,0
31.	Jan Kaifer	GKepleraPH	1	3	6,5						3	10,5	10,5
32.	Stanislav Lukeš	GPísnickáPH	4	11			10,5					10,3	10,3
33.–34.	Adam Dřínek	GNAlejíPH	3	1	8							8,0	8,0
	František Kmječ	G Brandýs	1	3	8							8,0	8,0
35.	Jiří Löffelmann	GLitoměřPH	3	4	5			1				7,9	7,9
36.	Jan Neumann	GNAlejíPH	3	2	7							7,7	7,7
37.–39.	Jakub Dobrý	GMikulášPL	3	4	7							7,6	7,6
	Tomáš Raunig	GHlu	2	1	6,5							7,6	7,6
	Anna Šebestíková	GČeskáČB	2	2	6,5							7,6	7,6
40.	Michael Kozel	GZborovPH	3	1	7							7,5	7,5
41.–42.	Jan Jeníček	GNAlejíPH	1	1	6							7,4	7,4
	Kryštof Mitka	ZŠUniverzum	0	1	6							7,4	7,4
43.	Jakub Jirkal	GJungmanLT	2	1	7							7,2	7,2
44.	Přemysl Šťastný	GŽamberk	4	13	7		1					7,1	7,1
45.	Jakub Spišák	G VBN Prie	4	1	2		0,5			0,5		7,0	7,0
46.	Erik Kučák	GHorMichal	4	1	2			1				6,7	6,7
47.	Michal Töpfer	G DrJPekMB	4	11	5						4	6,6	6,6
48.	Jonáš Havelka	GJírovcČB	1	2				1				2,2	2,2

matfyz

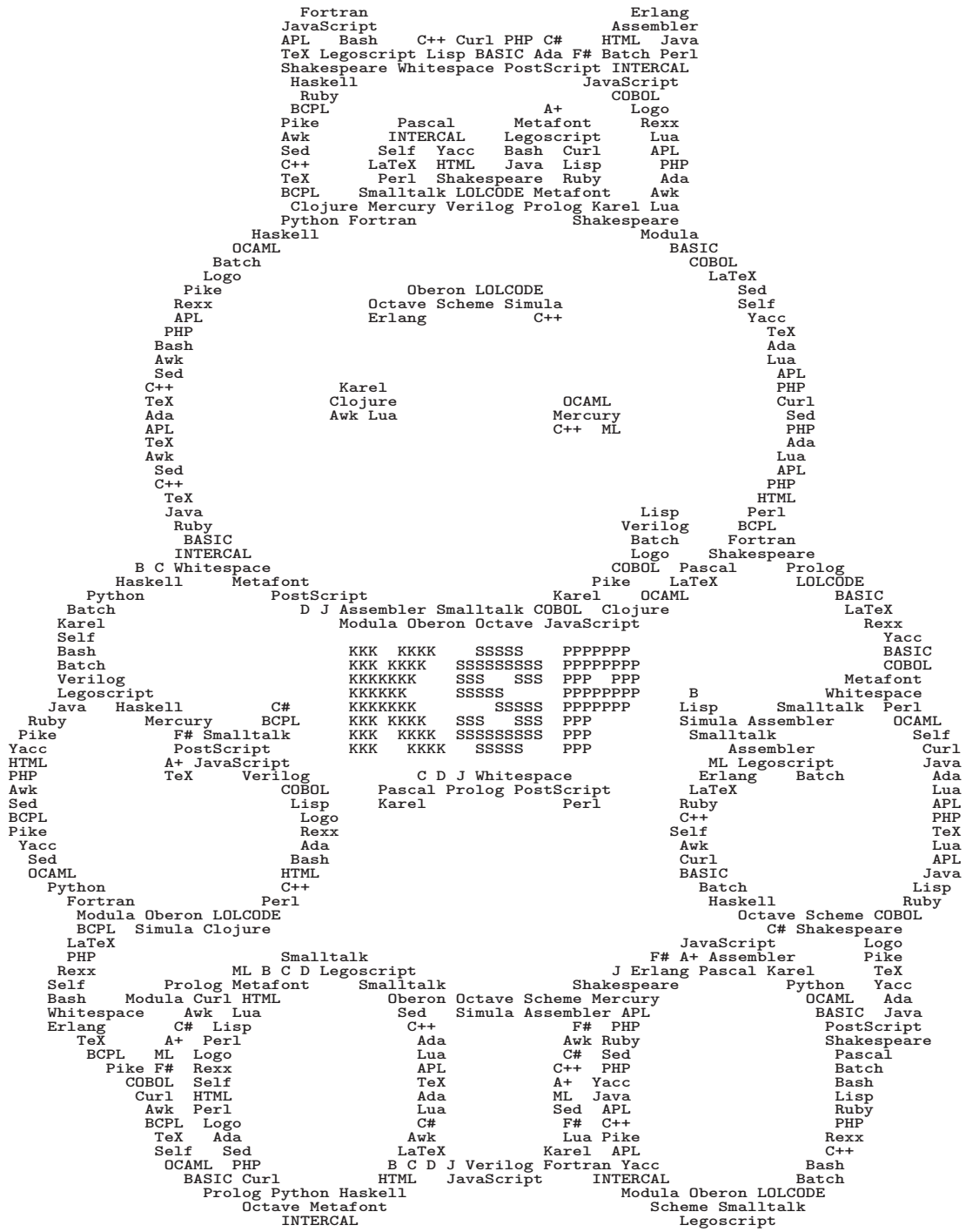
KSP pro vás připravují studenti Matematicko-fyzikální fakulty Univerzity Karlovy.

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`.


Fortran Erlang
JavaScript Assembler
APL Bash C++ Curl PHP C# HTML Java
TeX Legoscript Lisp BASIC Ada F# Batch Perl
Shakespeare Whitespace PostScript INTERCAL
Haskell JavaScript
Ruby COBOL
BCPL A+ Logo
Pike Pascal Metafont Rexx
Awk INTERCAL Legoscript Lua
Sed Self Yacc Bash Curl APL
C++ LaTeX HTML Java Lisp PHP
TeX Perl Shakespeare Ruby Ada
BCPL Smalltalk LOLCODE Metafont Awk
Clojure Mercury Verilog Prolog Karel Lua
Python Fortran Shakespeare
Haskell Modula
OCAML BASIC
Batch COBOL
Logo LaTeX
Pike Oberon LOLCODE Sed
Rexx Octave Scheme Simula Self
APL Erlang C++ Yacc
PHP TeX
Bash Ada
Awk Lua
Sed APL
C++ Karel PHP
TeX Clojure OCAML Curl
Ada Awk Lua Mercury Sed
APL C++ ML PHP
TeX Ada
Awk Lua
Sed APL
C++ PHP
TeX HTML
Java Lisp Perl
Ruby Verilog BCPL
BASIC Batch Fortran
INTERCAL Logo Shakespeare
B C Whitespace COBOL Pascal Prolog
Haskell Metafont Pike LaTeX LOLCODE
Python PostScript Karel OCAML BASIC
Batch D J Assembler Smalltalk COBOL Clojure LaTeX
Karel Modula Oberon Octave JavaScript Rexx
Self Yacc
Bash KKK KKKK SSSSS PPPPPPP BASIC
Batch KKK KKKK SSSSSSSSS PPPPPPPP COBOL
Verilog KKKKKKK SSS SSS PPP PPP Metafont
Legoscript KKKKKK SSSSS PPPPPPPP B Whitespace
Java Haskell C# KKKKKKK SSSSS PPPPPPP Lisp Smalltalk Perl
Ruby Mercury BCPL KKK KKKK SSS SSS PPP Simula Assembler OCAML
Pike F# Smalltalk KKK KKKK SSSSSSSSS PPP Smalltalk Self
Yacc PostScript KKK KKKK SSSSS PPP Assembler Curl
HTML A+ JavaScript ML Legoscript Java
PHP TeX Verilog C D J Whitespace Erlang Batch Ada
Awk COBOL Pascal Prolog PostScript LaTeX Lua
Sed Lisp Karel Perl Ruby APL
BCPL Logo C++ PHP
Pike Rexx Self TeX
Yacc Ada Awk Lua
Sed Bash Curl APL
OCAML HTML BASIC Java
Python C++ Batch Lisp
Fortran Perl Haskell Ruby
Modula Oberon LOLCODE Octave Scheme COBOL
BCPL Simula Clojure C# Shakespeare
LaTeX JavaScript Logo
PHP Smalltalk F# A+ Assembler Pike
Rexx ML B C D Legoscript J Erlang Pascal Karel TeX
Self Prolog Metafont Smalltalk Shakespeare Python Yacc
Bash Modula Curl HTML Oberon Octave Scheme Mercury OCAML Ada
Whitespace Awk Lua Sed Simula Assembler APL BASIC Java
Erlang C# Lisp C++ F# PHP PostScript
TeX A+ Perl Ada Awk Ruby Shakespeare
BCPL ML Logo Lua C# Sed Pascal
Pike F# Rexx APL C++ PHP Batch
COBOL Self TeX A+ Yacc Bash
Curl HTML Ada ML Java Lisp
Awk Perl Lua Sed APL Ruby
BCPL Logo C# F# C++ PHP
TeX Ada Awk Lua Pike Rexx
Self Sed LaTeX Karel APL C++
OCAML PHP B C D J Verilog Fortran Yacc Bash
BASIC Curl HTML JavaScript INTERCAL Batch
Prolog Python Haskell Modula Oberon LOLCODE
Octave Metafont Scheme Smalltalk
INTERCAL Legoscript