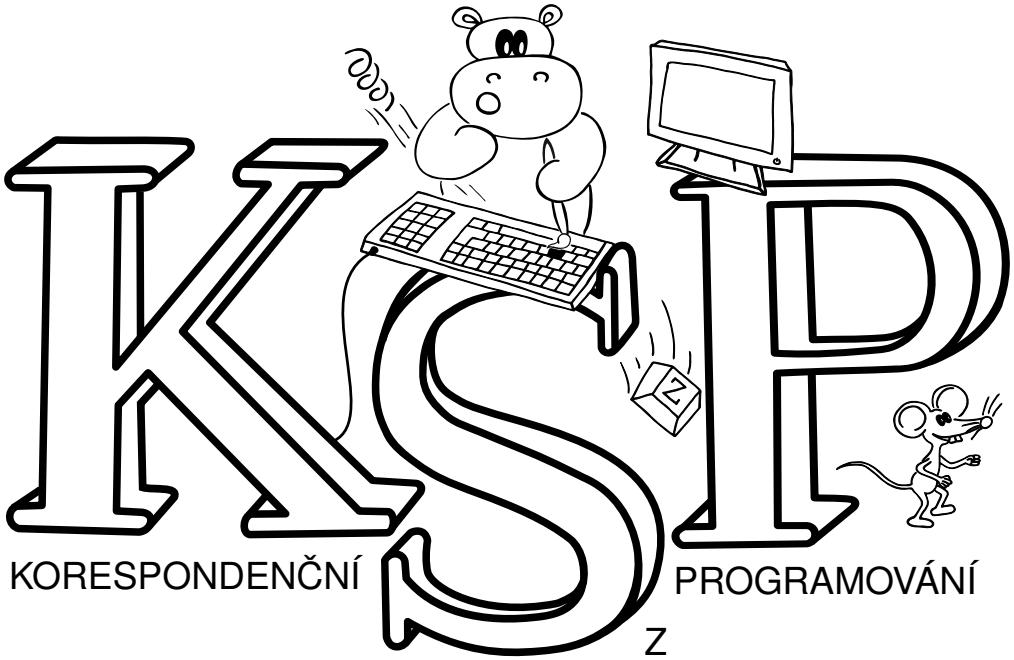
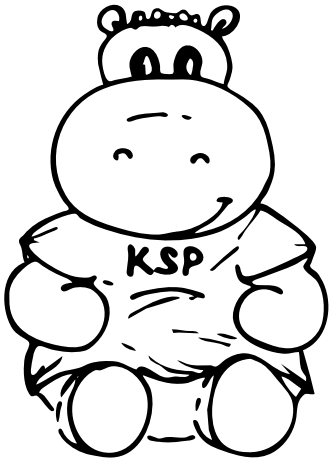
Dokud existují počítače, bude existovat i KSP!





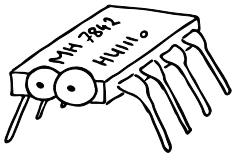
Toužíš po nových vědomostech?

Chceš poznávat nové lidi?

Zajímáš se o počítače?

Láká Tě trocha soutěžení?

Hledáš výzvu pro svou hlavu?



Byla Tvá odpověď alespoň jednou „ano“?
Pak hledáme právě Tebe. Do KSP
se může zapojit každý, tedy i Ty. Otoč list!

Co je KSP?

KSP je Korespondenční seminář z programování. Jak takový seminář funguje? Několikrát za rok vydáváme série obsahující různé úlohy a posíláme je řešitelům.

Ti mají několik týdnů na vymyšlení a odevzdání řešení. My je pak opravíme, okomentované a obodované pošleme zpět a zveřejníme autorská řešení. KSP má dvě kategorie: Z pro začínající řešitele a hlavní pro ty zkušenější, kde číhají záludnější úlohy.

Kdo seminář organizuje?

Organizátoři jsou studenti Matematicko-fyzikální fakulty Univerzity Karlovy v Praze (MFF UK), většinou bývalí řešitelé.

Co najdu v zadání?

Můžeš řešit teoretické a praktické úložky. Vždy je důležité vymyslet postup (návod) jak nalézt řešení, například jak může počítač rychle najít nejkratší cestu z Kocourkova do Prčic.

Součástí zadání jsou i studijní texty, jejichž přečtení Ti dá nástroje k řešení úloh. Kuchařky jsou krátké textíky o různých tématech. Seriál pro změnu probere v průběhu roku jedno téma do hloubky.

Jak úlohy vypadají?

V teoretických úlohách je třeba postup slovně popsat a odevzdat nám ho, my jej pak opravíme a okomentujeme.

V praktických úlohách nejde o popis, ale o výsledek. U open-data úloh si stáhneš vstupní data, která zpracuješ Tebou zvoleným způsobem, nejlépe programem v libovolném programovacím jazyce. U dalších praktických úloh se odevzdává přímo zdrojový kód do vyhodocovacího systému CodEx. V každém případě ihned vidíš, zda je výsledek správný.

Něco nového by nebylo?

Novou specialitou KSP-Z je možnost odevzdat praktické úlohy po termínu – ještě týden po zveřejnění slovních popisů řešení lze odevzdávat úlohy za třetinu bodů. Teprve poté se objeví i zdrojové kódy.

V některých sériích se také můžeš těšit na soutěživé úložky. Při nich nebudeš soupeřit s organizátory, ale s ostatními řešiteli.

Proč mám KSP řešit?

Během řešení KSP se naučíš programovat. To se Ti může v životě hodit, obzvlášť pokud se chceš stát programátorem. ;)

Díky KSP můžeš poznat informatiku v celé její kráse – mocné programy, magické datové struktury… prostě to, co se ve škole nedozvíš. Může to být užitečné nejen při řešení matematické olympiády kategorie P. Navíc nejlepší řešitele zveme na soustředění, kde můžeš poznat nové kamarády.

Také pokud se staneš úspešným řešitelem hlavní kategorie, vezmou Tě na Matfyz bez přijímaček.

Hmmm… soustředění?

Jsou dvě, jarní především pro řešitele KSP-Z a podzimní pro řešitele hlavního KSP. Obě jsou týdenní akcí plnou přednášek a zážitků, kterou určitě stojí za to zažít. ;)

Dostanu i něco hmotnějšího?

Ano, pokud se dostaneš mezi ty nejlepší. Tři nejúspěšnější řešitelé si budou moci vybrat jako odměnu knížku nebo například tričko, hrneček, hrocha.

Vůbec nevím, jak začít…

Inu, žádný učený z nebe nespadl, chce to studovat a zkoušet. ;) Dobrým odrazovým můstkem může být naše Encyklopedie, jejíž součástí jsou i kuchařky včetně úplných základů.

S výběrem Ti může pomoci i bodování – lehčí úlohy bývají většinou za méně bodů.

Napadá mě jen špatné řešení

Tak prostě odevzdej i to. :) Špatné, pomalé nebo neúplné řešení je lepší než žádné. Za nedokonalá řešení u teoretických úloh hlavy rozhodně netrháme. Naopak, pokusíme se Ti poradit, co zlepšit. U praktických úloh zase bývá několik vstupů malých, takže za ně získáš část bodů i s jednodušším řešením.

Co když mi něco není jasné?

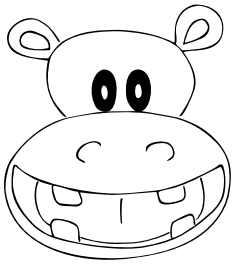
Klidně se nás ptej. Na dotazy k úlohám se nejvíc hodí naše diskusní fórum. Podrobnosti o fungování semináře nalezneš na webu. A budeš-li mít stále nějakou otázku, čteme mail a jsme na Facebooku.

Web: *http://ksp.mff.cuni.cz/*
Mail: *ksp@mff.cuni.cz*
Fórum: *http://ksp.mff.cuni.cz/forum/*
Facebook: *http://facebook.com/ksp.mff/*

Korespondenční Seminář z Programování

28. ročník **KSP** **Červenec 2015**

Milí řešitelé, řešitelky a řešitelčata!



Vítejte ve 28. ročníku KSP, jehož první leták držíte v ruce. Letos bude každá série obsahovat 7–8 úloh, často s lehčími variantami pro začátečníky. Do celkového bodového hodnocení se z každé série započítá 5 nejlépe vyřešených úloh.

Za úspěšné řešení KSP je také možno být přijat na MFF UK bez přijímacích zkoušek. Úspěšným řešitelem se stává ten, kdo získá za celý ročník alespoň 50 % bodů. Za letošní rok půjde získat maximálně 300 bodů, takže hranice pro úspěšné řešitele je 150.

Upozorňujeme letošní maturanty, že termín odevzdání páté série bude pravděpodobně příliš pozdě na to, aby pátou sérii doháněli chybějící body. Diplom úspěšného řešitele ale můžeme v případě potřeby zaslat i dříve, budete-li mít dost bodů.

Každému řešiteli, který v tomto ročníku z každé série dostane alespoň 5 bodů, darujeme KSP propisku, blok, tužku a možná i další překvapení. Navíc každému, kdo vyřeší **alespoň jednu ze dvou nejvíc bodovaných úloh** první série na plný počet bodů, pošleme sladkou odměnu.

Pokud budete mít jakoukoliv otázku, neváhejte se zeptat. Kontaktní adresy najdete v tiráži na konci letáku. Přejeme hodně štěstí!

Termín série: **26. října 2015 v 8:00 SELČ**
Odevzdávání: Přes web na adrese *https://ksp.mff.cuni.cz/submit/*.

Značky úloh:
- Lehčí úloha (či její část) vhodná pro začátečníky
- Těžká úloha pro zkušené
- Praktická úloha odevzdávaná do systému CodEx
- Praktická open-data úloha
- Úloha, kterou lze často řešit algoritmem z kuchařky
- Seriálová úloha

První série dvacátého osmého ročníku KSP

Vyhrocené sousedství

Je jasná, klidná letní noc. Téměř celé město spí, u řeky se prochází nevinné páry a na diskotékách vydrželi už jen ti největší tahouni. Silnice jsou prázdné. Jen občas na nich můžeme potkat zpívající skupinku, oslavující včerejší fotbalový úspěch, nebo taxík vezoucí ty, kdo už domů nezvládají dojít po svých.

Všechen klid tady kazí jen jeden zmateně pojíždějící motorkář a skupina policejních aut snažící se o jeho dopadení. Motorkář nejede nijak rychle, ale moc se nezajímá o protisměry, nemá boty, natož helmu, a už vůbec nehodlá zastavit. Hlídkám zdařile uniká a úspěšně objíždí i všechny zátarasy. Ale jen do chvíle, než dojede do slepé uličky, kde jej policie konečně dopadne.

28-1-1 Jízda na biomotorce — 10 bodů

Představte si, že jedete městem na motorce, jiné než té v příběhu: na biomotorce. Takové, která jezdí na pomeranče. Na jeden pomeranč je schopná ujet celých 100 metrů. Má to ale háček: pomeranče jsou velké a vejde se jich do nádrže jen 10. Naštěstí ale pomeranče ve městě jen tak rostou na stromech.

Mapa je tvořená křižovatkami a ulicemi, které je spojují. Můžeme si ji tedy představit jak neorientovaný ohodnocený graf, v němž navíc každý vrchol má daný počet pomerančů, které v něm rostou.

Napište program, který na vstupu dostane mapu města a najde trasu ze startu do cíle, během které co nejméněkrát projedeme ulicemi (tedy nás zajímá počet přejezdů a ne celková délka trasy). Během cesty můžete brát pomeranče z křižovatek. Nesmíte však překročit limit 10 pomerančů v nádrži. Křižovatky i ulice se mohou na trase opakovat.

Všechny sebrané pomeranče na křižovatce dorostou hned po jejím opuštění a dojetí na jinou křižovatku. Na začátku má motorka prázdnou nádrž, ale můžete ji naplnit pomeranči ze startovní křižovatky.

Formát vstupu: Na prvním řádku vstupu budou čísla N, M, S a C oddělená jednou mezerou, kde N je počet křižovatek ve městě, M počet ulic, S číslo startovní křižovatky a C číslo cílové křižovatky.

Na druhém řádku bude N mezerou oddělených čísel udávající počty pomerančů v jednotlivých křižovatkách. Na dalších M řádcích jsou popsány ulice, každá třemi čísly. První dvě udávají čísla křižovatek, mezi kterými ulice vede, a třetí udává délku ulice v metrech. Všechny ulice jsou obousměrné.

Pro hodnoty na vstupu dále platí:

- $1 \leq N \leq 30\,000$
- $1 \leq M \leq 1\,000\,000$
- Křižovatky jsou číslované od 1 do N.
- Na každé křižovatce leží maximálně 10 pomerančů.
- Ulice jsou dlouhé minimálně 100 a maximálně 1 000 metrů. Délky jsou násobky 100.

Formát výstupu: Na výstup vypište jedno celé číslo udávající délku nejkratší cesty v počtu projetých ulic ze startu do cíle, na které vám nikdy nedojdou pomeranče v nádrži.

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

Motorkář je mírně zakrvácený, vytřeštěný a stále opakuje, že musí utéct. Není mu to ale nic platné, je odveden na stanici a usazen na židli do rohu, kde čeká, než na něj přijde řada.

⋆⋆⋆

Tím motorkářem jsem já. Roztřeseně sedím, hledím do země a naslouchám okolnímu dění.

„Já nevim proč! Prostě najednou odněkud vyskočil, vyrval mi tu pochodeň z ruky, srazil mě na zem a zdrhnul někam do lesa!“ rozčiloval se muž na policejní stanici.

„A nemůžete prostě vzít kus klacku a vyrobit si novou? To nás musíte zdržovat takovýma prkotinama?“ reaguje znuděně policista.

„To nebyla jen tak nějaká pochodeň,“ brání se muž, „to byla speciální žonglovací pochodeň za tisíc korun!“

„Dobře, dobře...“ vzdává se policista, „tak to teda vezmeme znova od začátku.“ „Budeš zapisovat,“ říká kolegovi.

„Jak se incident odehrál?“ ptá se.

„Žongloval jsem v rámci fire show na louce u lesa. Najednou se vedle mě objevil chlap, vzal mi pochodeň, srazil mě na zem a zdrhl. To už jsem vám přece říkal!“ popisuje muž a praští naštvaně pěstí do stolu.

„Můžete pachatele nějak popsat?“ ptá se nenuceně dál policista.

„No... já jsem ho moc neviděl. Byla tma a soustředil jsem se na žonglování,“ vykládá muž, „takový normální chlap, o trochu vyšší než já a byl docela silný.“

„Skvělý! Napiš tam, že hledáme normálního muže vysokého asi 185 centimetrů, co chodí po městě s pochodní,“ vysmívá se policista, „proboha chlape, to na něm nebylo nic neobvyklého?“

„Ne,“ odpovídá muž, „vlastně... měl na ruce něco jako moderní hodinky. Takový technologický náramek – pořád to blikalo.“

Policista se pohrdavě otočí na kolegu: „Máš to?“

„Jo, mám,“ říká kolega.

„Tak děkujeme za nahlášení. V případě jakýchkoliv výsledků ve vašem případu vás ihned budeme informovat. Číslo na vás máme. Na shledanou!“ loučí se s mužem.

Muž se zvedl a trochu nesouhlasně odcházel. Asi tušil, že svou pochodeň již nikdy neuvidí. Jak by taky mohl, když dneska je problém najít ukradené auto, nebo třeba i kamion. A kamion se oproti pochodni sakra dobře hledá.

Ale vlastně ho trochu chápu. Hrátky s ohněm jsou fajn. Když mi bylo pět, tak jsme si s klukama ze sousedství tajně s ohněm hráli. Jen než mi jeden blbeček zapálil kraťasy a způsobil ošklivé popáleniny. Od té doby mám z ohně panickou hrůzu. Ono totiž utíkat před ohněm, který hoří přímo na vás, je čirá marnost.

28-1-2 Zapalování kostek — 8 bodů

Hrajeme následující hru. Máme postavenou pyramidu z dřevěných kostek o K patrech. To jest v prvním patře máme K kostek, na nich stojí $K-1$ kostek, na těch stojí $K-2$ kostek, až na špičce stojí jen jedna kostka.

Dva hráči se střídají v tazích. V jednom tahu hráč vybere jednu kostku v pyramidě a zapálí ji. To zapálí (obě) kostky, které na ní stojí, ty zas zapálí kostky, které stojí na nich a tak dále. Sousední kostky od aktuální nechytnou! Pak hraje druhý hráč. Vyhrává ten, kdo zapálí poslední kostku v pyramidě.

Pro dané K navrhněte vyhrávající strategii pro prvního hráče. To znamená takovou strategii, že ať druhý hráč dělá cokoliv, první vždy vyhraje.

Lehčí varianta (za 3 body): Popište konkrétní strategii pro $K = 4$ a $K = 5$.

„Tak teď vy,“ křikl na mě policista.

Hlídka mě odvádí k výslechovému stolu a sundává mi pouta. Rozklepaně pokládám ruce na stůl, sklopím hlavu a mlčím. Po chvíli se ozve policistův rozzářený hlas.

„Ale, ale... Vás jsme tady měli i včera, že jo? Nějaké sousedské problémy, jestli si správně pamatuji.“

„A-a-ano... S-s-soused mi-mi-mi z-zbořil m-můj ko-ko-komín.“

28-1-3 Bourání komínu I — 12 bodů

Na zahradě stojí V metrů vysoký komín, který chceme zbourat. K tomu máme k dispozici N bomb. Bomba i váží w_i kilogramů a zničí přesně d_i metrů komínu. Komín bouráme postupně odshora. Při boření vždy musíme vynést bombu až na vršek zbytku komínu a tam ji odpálit. Tím se komín sníží o d_i metrů (přitom nesmí vyjít záporná výška, nechceme skončit s jámou).

S nošením bomb se ale chceme co nejméně nadřít. Vynesení i-té bomby na komín vysoký x nás stojí $x \cdot w_i$ jednotek energie. Navrhněte algoritmus, který naplánuje boření komínů tak, abychom celkem použili nejmenší možné množství energie.

Lehčí varianta (za 7 bodů): Řešte případ, kdy všechny bomby dohromady zničí přesně V metrů. Tedy víme, že je všechny musíme použít.

28-1-4 Bourání komínu II — 8 bodů

Stejná úloha jako minulá, ale nyní ji řešte prakticky v CodExu. Stačí ovšem, když místo přesného postupu bourání budete vypisovat pouze minimální počet jednotek energie, kterou je nutno ke zbourání použít.

Formát vstupu: Na vstupu na prvním řádku dostanete dvě čísla V a N – výšku komínu a počet bomb. Na dalších N řádcích jsou vždy dvě čísla w_i a d_i udávající váhu a sílu bomby i. Dále platí:

- $1 \leq V \leq 10\,000$
- $1 \leq N \leq 10\,000$
- $1 \leq w_i \leq 100\,000$
- $1 \leq d_i \leq 10\,000$
- Pro několik prvních vstupů platí, že bomby dohromady zničí přesně V metrů.

Formát výstupu: Na výstup vypište jediné číslo udávající minimální námahu, se kterou je možno komín zbourat.

Ukázkový vstup:	*Ukázkový výstup:*

`12 3`
`4 6`
`2 2`
`12 4`
	`108`

Nejdříve použijeme první bombu, stojí nás $12 \cdot 4 = 48$ jednotek energie, komín po ní bude mít výšku 6. Poté použijeme druhou, stát nás to bude $6 \cdot 2 = 12$, z komínu zbudou 4 metry. Nakonec třetí bombu, k použité energii přičteme

[1] `http://ksp.mff.cuni.cz/viz/codex`

$4 \cdot 12 = 48$, celkem jsme tedy využili 108 jednotek energie a z komínu nic nezbylo.

Tato úloha je praktická a řeší se ve vyhodnocovacím systému CodEx.[1] Přesný formát vstupu a výstupu, povolené jazyky a další technické informace jsou uvedeny v CodExu přímo u úlohy.

„Ano, čtu to tady: Pán nahlásil zboření svého výstavního komínu. Podezřívá z toho svého souseda kvůli dlouhodobým sporům a protože ten jej údajně neprávem obviňuje z otrávení svého psa... Hmmm... Koukám, že jsme panu sousedovi poslali předvolání. Tak nebojte, ono se to brzy vyřeší. Ale teď zpět k vám. Jaké vy jste dneska dělal problémy?“

„Jel na motorce jako blázen, opětovně ignoroval výzvy hlídek k zastavení a po dopadení blekotal nějaké nesmysly o útěku. Možná je pod vlivem drog,“ odpovídá rázně muž z hlídky.

„Tak se na to podíváme. Dělej zase zápis,“ ukazuje na kolegu.

„Proč jste ignoroval všechny výzvy k zastavení?“

„J-já se p-p-potřeboval co-co nejrychleji d-dostat sem... a-abych n-nahlásil, c-co se stalo.“

„Pokračujte,“ vybízí policista s náznakem zvědavosti.

„V-víte, já měl vždycky h-hroznou smůlu n-na své sousedy. Už v první třídě jsem seděl v l-lavici vedle kluka, který mi k-kradl svačiny a tr-trhal o-oblečení. T-to bylo vždycky d-doma problémů, že každý t-týden roztrhnu tričko. J-já ni-nikdy nepřiznal, že to dělá on. J-jsem se b-bál na něj ž-žalovat.“

„Proboha mluvte k věci. Mě zajímá dnešek a ne váš podělanej život.“

„A-ale t-to b-byla na-naprostá pr-prkotina o-oproti tomu, c-co se stalo t-teď,“ pokračuji, jako bych policistu vůbec neslyšel.

Začalo to hned včera, jak jsem vyšel z policejní stanice. Silně pršelo, tak jsem rozevřel deštník a vyrazil. Ze stanice to mám do práce kousek, jen pár bloků. Šlo se mi těžce, protože v ulicích foukal silný vítr a zápasil jsem s rozevřeným deštníkem. Nakonec jsem jej musel naklonit před sebe, abych jím prorážel vzduch. Tím jsem si zablokoval výhled. PRÁSK! Někdo přímo přede mnou vystupoval z auta a já, nevidě před sebe, mu kovovou špičkou deštníku vyryl do dveří pořádnou rýhu.

„Ty šmejde! Tys mi zničil auto! Nemůžeš se doprdele dívat na cestu?“ ozvalo se.

Nadzvedávám deštník, abych viděl, s kým mám tu čest, a mohl se dotyčnému omluvit. Stál tam asi padesátiletý pohublejší muž s vyholenou hlavou, který byl aspoň o pět čísel vyšší než já. Byl to můj soused.

„Ty?! Ty se ještě odvažuješ plést se mi do cesty, po tom, cos mi otrávil psa? To mi teda zaplatíš! To ti jen tak nedaruju!“ pokračuje.

„Já??? Já nikoho neotrávil. A nic vám platit nebudu! Tohle byla vaše chyba. Máte se koukat, jestli tam nejsou lidi, když otevíráte dveře,“ bráním se a s úšklebkem dodávám, „kromě toho vás v dohledné době přijdou vyšetřovat kvůli tomu komínu, který mi v noci někdo zbořil.“

Soused zasupěl a odměřeným zastrašujícím tónem řekl: „Ty mě jako budeš žalovat?“

„Jestli budu? Právě jsem to udělal, protože tohle vám už dál trpět nebudu!“

Soused zrudl, prohlídl si mě očima a potichu supěl: „Tos přehnal chlapečku. Mě tady nikdo žalovat nebude. Rozumíš? Nikdo! A už vůbec ne ty! ... Ty o mě ještě uslyšíš.“

Najednou začal křičet: „TY JEŠTĚ POZNÁŠ, CO JÁ DOKÁŽU A NEBUDE SE TI TO LÍBIT! NA TO SE MŮŽEŠ SPOLEHNOUT!“

Přešel chodník, ukázal směrem ke mně výhružné gesto, agresivně trhl dveřmi a zmizel ve vedlejší budově.

Nevyděsilo mě to. Naopak mi to zvedlo náladu a s pocitem zadostiučinění jsem pokračoval v cestě do práce. Konečně dosáhnu spravedlnosti! V duchu jsem si představoval policejní důstojníky klepající na dveře souseda a jak mi přímo on dává do ruky peníze za způsobenou škodu. Nebo ještě líp, dostává příkaz k vystěhování.

S takto dobrou náladou jsem došel do práce – dokonce přestalo i pršet. To jsem ale ještě netušil, co mě později v ten den čeká...

⋆⋆⋆

Z práce jdu rovnou domů. Po průchodu brankou se ještě jednou smutně podívám na hromádku cihel, která mi zbyla po komínu. On to totiž nebyl jen tak obyčejný komín. Byl to umělecky navržený výstavní komín, který jsme vlastnili už po dvě generace a ke kterému se pojí nejedna vzpomínka. Například když do něj můj mladší bratr zapadl při hře na schovávanou a přes dvě hodiny jsme jej nemohli najít. Nebo méně veselá historka: jak se nám z něj šířila plíseň do vedlejších záhonů a museli jsme se jí celý víkend zbavovat různými chemikáliemi.

28-1-5 Likvidace plísně — 10 bodů

Na zahradě nám vyrostlo N hromádek mnohavrstvé plísně. Té bychom se chtěli co nejrychleji zbavit. K tomu máme plošný postřikovač naplněný přípravkem proti plísním. V jednom kroku můžeme buď použít postřikovač a zničit tak jednu vrstvu plísně z každé hromádky, nebo vzít několik vrstev z jedné hromádky a dát je na jinou (klidně i novou) hromádku. Hromádek takto můžeme vytvořit kolik chceme.

Napište program, který spočítá, v kolika nejméně krocích je možné se zbavit celé plísně.

Formát vstupu: Na prvním řádku vstupu dostanete číslo N udávající celkový počet hromádek plísně. Na druhém řádku dostanete čísla $v_1, \ldots, v_N$ udávající počet vrstev na jednotlivých hromádkách.

- $1 \leq N \leq 10\,000$
- $1 \leq v_i \leq 10\,000$

Formát výstupu: Na výstup vypište jedno celé číslo udávající minimální počet kroků, v jakém je možné se plísně zbavit.

Příklad: Máme-li tři hromádky o počtech 9, 3 a 3, je nejlepší nejdříve dvěma přesuny z první hromádky vyrobit tři po třech plísních, a pak všech pět třemi postřiky vyhubit.

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

Věnoval jsem hromádce cihel další, poslední pohled, když vtom mě popadla zlomyslnost. Proč já mám mít na zahrádce hromadu cihel a ten, kdo ji zničil, pořádek? Ať si soused taky trochu vychutná svou vlastní práci! Vzal jsem několik kusů a hodil je přes plot.

Jednu do růží. Jednu do okurek. Další mezi chryzantémy. Pak jednu do rajčat. Do umělého bazénu a poslední jsem zničil psí boudu. Stejně už toho blbýho čokla nemá. Haha! Takhle už jsem se dlouho nebavil!

Když jsem se vydováděl, šel jsem domů. Po namáhavém dni se natáhl na pohovku, pustil televizi a usnul. Spalo se mi krásně a klidně, ale nevydrželo to dlouho. Probudila mě obrovská rána, která šla z mojí kuchyně. Cože? Nebyl to sen? Nebyl, hned vzápětí se ozvala další, ještě větší!

Celý zmatený spadnu z pohovky na zem a snažím se vzpamatovat. Co se děje? Najednou oknem proletí velký, těžký předmět. Dopadne přímo do zapnuté televize a ta se celá rozpadne.

Ten magor mi hází do domu cihly! Další letěla přes pohovku a roztříštila skleněný stolek přímo přede mnou. Letící střepy mi pořezaly tvář, ruce a pravou nohu. Naštěstí jsem stihl zavřít oči.

Tohle už došlo moc daleko! Tady jde o život! Musím něco udělat dřív, než sem hodí další a stane se něco fatálního.

„Sousede! Dost! Chci se usmířit! Tohle už se nám vymklo z rukou!“ křičím.

Další rána, tentokrát z koupelny.

„Dost! To auto ti zaplatím! Zahradu taky! Jenom už mi prosimtě přestaň ničit barák.“

Nastala chvíle klidu. Z okna se ozve sousedův vyrovnaný hlas: „Ty si vážně myslíš, že penězi se dá něco urovnat po tom, co jsi proved'? Tak pojď ven a zkus to. Čekám na tebe.“ „A ať tě ani nenapadne volat pomoc, pár cihel mi tady pořád zbývá!“ dodal.

Seděl jsem na místě a přemýšlel, co mám dělat. Vzhledem k okolnostem mluvil soused až překvapivě klidně a to mě zneklidňovalo. Takto klidného ho neznám. Nakonec jsem se vyhrabal ze střepů a vydal se do ložnice pro peníze. Šel jsem opatrně, u zdi a při tom se pořád ohlížel po oknech. Vzal jsem peníze a zamířil na chodbu. Všude byl naprostý klid a po sousedovi žádné známky.

„Sousede, jsi tam?“ volám směrem ke dveřím.

Ticho, žádná odpověď. Že by bylo po všem? Že by si uvědomil váhu svých činů a šel domů? Radši se ještě podívám ven, abych měl jistotu.

„Jdu ven!“ volám.

Přistupuji ke dveřím, opatrně beru za kliku a pomalu otvírám. Rozhlížím se. Nikde jej nevidím. Pomalu a tiše dělám krok směrem ven. Druhý. Třetí.

Náhle mi ztuhnou ruce i nohy. Zvláštním způsobem mě zabrní celé tělo a padám na zem. Nemůžu se hnout. Jsem v jedné velké křeči. Nad sebou vidím stát souseda a pomalu se mi udělá temno před očima. Tak takový je to pocit... když vás někdo uzemní paralyzérem.

⋆⋆⋆

Ležím na tvrdé, nepohodlné, studené podložce a pomalu se probírám. Třeští mi hlava. Rozhlížím se a snažím se zjistit, kde to jsem. Jsem v potemnělé místnosti s jedním menším oknem, kterým prosvítá měsíční světlo. Ze všech stran kolem mě jsou mříže. Jsem zavřený v kleci.

Na zdech visí spousta různých středověkých nástrojů. Nůžky mnoha velikostí, kladivo zakončené hřebíky, řemdih, pouta... Na věšáku visí svěrací kazajka a v opačném rohu místnosti stojí... to je opravdu skřipec?

Jsem v mučírně. To nemůže být pravda! To je jenom sen! Hlava mi stále třeští, že nejsem schopný si ani kleknout. Tiše ležím na zemi a čekám, co se bude dít. Asi po půl hodině vchází soused.

„To je ale překvapení! Pán se nám konečně probral!“ zvolal s mírným nadšením v hlase. „To si spolu konečně můžeme užít trochu legrace,“ řekl a usmál se na mě. Pak hned zvážněl a zeptal se přísně: „Víš, proč jsi tady?“

Mlčím. Nejsem schopný slova. Vyčkávám. Najednou se mu v ruce objeví zbraň a střelí mě do paže.

„Na něco jsem se tě ptal!“

„Au!“ chytám se za paži. Nekrvácí. Ale nahmatávám pod kůží zarytou kuličku. Má airsoftku. A nepříjemně silnou!

„Neslyšíš, nebo co?“ střelí mě znova, tentokrát do břicha, „ptal jsem se, jestli víš, proč jsi tady!“

„Jo, tuším!“ chytám se za břicho a vzdychám, „asi kvůli vašemu autu a těm cihlám, co jsem k vám hodil. Omlouvám se! Všechno zaplatím! Jen už do mě prosim nestřílejte!“

„Špatně!“ zaburácel podrážděně a trefil mě do ramene, „za těma chryzantémama,“ rána do holeně, „na kterých ses vydováděl,“ rána pod lopatku, „si hrála moje vnučka!“ završil přímou ranou do čela.

Choulím se v bolestech na zemi. Nedokáži se natočit tak, aby mě nic nebolelo. On zatím odkládá zbraň. Vnučka? On měl nějakou vnučku? Ani jsem nevěděl, že měl ženu nebo děti.

„Dokážeš si představit, co se stane se čtyřletou holčičkou, když na ni dopadne cihla???“ dal hlavu co nejblíž mřížím a potichu a chladně pokračoval, „že nedokážeš? Tak počkej pár hodin a já ti to pomalu a velmi, velmi přesně ukážu.“

Odstoupil několik kroků, sedl si na židli a pozoroval mě svým chladným výrazem. Byl jsem šokován. Otřesen vším, co mi právě řekl, jsem zapomněl na všechny své bolesti a hlavou mi začala probíhat spousta nepříjemných otázek.

Opravdu měl vnučku? Opravdu bych ji přehlédl? Vždyť ta cihla dopadla přímo doprostřed záhonu, tam nikdo nemohl být! A nebo jen chci, aby to tak bylo, a ve skutečnosti jsem v rozrušení nedával pozor, kam co házím? Jak může mít vnučku v domě, kde má mučírnu? Jsme vůbec u něj doma? Jak dlouho jsem byl mimo po zásahu paralyzérem? Nevymyslel si svou vnučku, jen aby teď sledoval mě, užívajícího se pocitem viny? Ale co když si ji nevymyslel? Jak já s tím teď budu žít? Budu vůbec žít? Jak to myslel, že mi to pomalu a přesně ukáže? To mě plánuje něčím rozmáčknout?

Na žádnou z těch otázek jsem neuměl odpovědět a postupně padal do hlubšího a hlubšího pocitu agonie, který jsem doprovázel tichým sípáním. Soused mě neustále bedlivě sledoval. Pak se zvedl, přistoupil ke kleci a hodil mi nějaký ovladač s displejem a tlačítky.

„Na, vem si to. Sleduj displej. Když zasvítí zeleně, musíš zadat správné číslo.“

„Bav se. Musím si teď na tebe něco připravit,“ a odešel.

28-1-6 Úloha z ovladače — 12 bodů

Držíte v ruce ovladač, na jehož displeji se postupně objevují čísla. Jakmile displej zasvítí zeleně, musíte zadat nejmenší z posledních K čísel, které jste viděli. Toto číslo K je pevně dané.

Navrhněte datovou strukturu, která bude umět efektivně zpracovávat následující dvě operace: přidání prvku a vypsání nejmenšího z posledních K přidaných prvků.

Plný počet bodů můžete získat za řešení, které potřebuje průměrně konstantní čas na dotaz. Tedy některé operace struktury můžou být pomalejší, ale posloupnost D operací zabere čas $\mathcal{O}(D)$. Bonusové body můžete získat za řešení, které potřebuje konstantní čas na každý jeden dotaz.

Dělal jsem, jak řekl. Sledoval čísla na displeji. Najednou zasvítil zeleně. Rychle jsem něco zadal a potvrdil. V tu chvíli nade mnou něco zavrzalo a strop klece klesl asi o tři centimetry. Neee! Tak takovou já tady hraji hru. Takhle to myslel! Vždyť já vůbec nevim, co tam mam zadávat!

Během mého panikaření na displeji proběhlo dalších několik čísel a opět zezelenal. Zkusím nedělat nic, třeba to

nezareaguje! Asi čtvrt minuty byl klid. Pak displej zablikal červeně a strop se znova snížil. Tentokrát asi o deset centimetrů.

To je ještě horší! Musím se teda snažit hrát. Vždy chvíli čekám a zkouším zadávat různá čísla. Nikdy jsem se netrefil a strop stále pomalu klesal. To je naprosto beznadějný, takhle nikdy nemám šanci nic uhodnout. A jde to vůbec? Nemá mi to jen dávat falešnou naději na moji záchranu? Padám do hluboké deprese a už prostě jen zadávám nějaká čísla, nevnímaje jaká.

Strop klece se již dostal tak nízko, že jsem si musel lehnout. Už jen pár poklesů a bude po všem. Už jen pár poklesů a nebude mě nic trápit. Strop už je tak blízko, že cítím narezlé železo, které ke mně stále klesá.

Hluboce dýchám. Snažím se každý další pokles nevnímat, ale nejde to. Zavřu oči a odhodím krabičku. Prosím, ať už to skončí! Já chci mít klid! Mříže naposled zavržou a zastaví se přesně ve výšce, že se jich má prsa při každém nádechu dotýkají. Dál se nic neděje. Otevřu oči. Mříže jsou až u mě. Mám tak málo místa, že se sotva můžu pohnout.

Najednou se za mnou zableskne a začnou se ozývat kroky. Už je zas tady! A má s sebou oheň. Já nesnášim oheň! Mám z něj panickou hrůzu! Vydávám vyděšené povzdechy. Představa upálení je pro mě naprosto zničující. To je ta nejhorší možná smrt, co může být!

Není to soused. Po místnosti se prochází neznámý muž asi po třicítce. V ruce drží pochodeň, rozhlíží se všude kolem a pečlivě si prohlíží předměty na zdech. Kdo to je? Je to snad otec sousedovy vnučky, který se mi taky přišel pomstít? Vybírá si teď vhodný mučící nástroj, kterým mi znepříjemní poslední hodiny života?

Sebral ze zdi dlouhé nůžky, kterými by se dalo ustřihnout i celé zápěstí a udělal pár kroků směrem k východu. Najednou se otočil a zadíval se na mě. Zastavil se mi dech. Vystrašeně se dívám jeho směrem a ležím bez nejmenší známky pohyblivosti. Vykročil směrem ke mně. Vyskočil na klec a skrčil se. Stále ty obrovské nůžky držel v ruce. Snažím se nemyslet na to, co s nimi hodlá dělat, ale nejde to. Furt dokola mi hlavou probíhá, jestli víc bolí ustřižené zápěstí, anebo prostřižené stehno.

Muž si nůžky strčil za opasek, koukl se na své moderní hodinky a něco na nich nastavil. Z hodinek vylezla dlouhá žhavá jehla. Tu vzal a začal s ní objíždět mříže. Mě se ani nedotkl, aspoň zatím. Dokončil okruh, škubl za mříže a vyrval je.

28-1-7 Vyříznutý kus mříže — 10 bodů

Ve čtvercové mřížce je nakreslený mnohoúhelník s N vrcholy, které jsou umístěny přesně v mřížových bodech. Navrhněte algoritmus, který na vstupu dostane souřadnice bodů mnohoúhelníka (v pořadí na obvodu) a spočítá, kolika mřížovými body prochází jeho strany.

Stále nehybně a vyčerpán ležím. Bojím se jakkoliv zareagovat. „Uteč!“ pošeptá mi, zvedne pochodeň a začne odcházet.

Nevěře, co se právě stalo, se pomalu zvedám a podezřívavě se dívám na muže. Jen aby to nebyla nějaká hra, dávající mi falešnou naději na svobodu. Vtom do dveří vejde soused.

„C-cože? Kdo jsi a co tady děláš?“ napůl překvapeně a napůl vyděšeně křičí na muže a začne jej ohrožovat pěstí.

Muž jej tvrdě odstrčí ke zdi a utíká ven. Soused hned vyrazí za ním a natahuje po něm ruku. Vtom za rohem zablýskne ostré světlo a soused začne neuvěřitelně řvát. Řval, jakoby mu tloukli hřebíky do kolen. Byl to takový úzkostný a dávivý řev, jaký jsem nikdy předtím neslyšel.

Já přestal na cokoliv čekat a zamířil k východu. Má záchrana bylo to jediné, co mě zajímalo.

Dobelhal jsem se do vedlejší místnosti. Tam se v rohu v bolestech svíjel zakrvácený soused. Chyběla mu půlka paže a z rány stříkala tmavě rudá krev. Prošel jsem místností, jak rychle to jen šlo, a začal hledat východ z domu.

Venku byla naprostá tma. U branky bylo zaparkované auto a vedle motorka. Po neznámém muži ani stopy. Bez rozmýšlení jsem sedl na motorku a ujížděl pryč. V hlavě mi zůstala jen jedna myšlenka a tou byl ÚTĚK!

Karel Tesař

28-1-8 Programování podle Darwina — 15 bodů

V letošním seriálu se budeme věnovat přírodou inspirovaným algoritmům, jakými jsou například *evoluční algoritmy* či *neuronové sítě*. Téma je velmi široké a obsahuje v sobě velkou spoustu postupů, ze kterých my se budeme věnovat jen těm základním a často používaným.

Úvod do evolučních algoritmů

Proč se vlastně informatici snaží přírodou inspirovat? Jednak určitě hraje roli motivace vyzkoušet si něco neobvyklého, ale jedním z hlavních důvodů je, že tradičními (exaktními) informatickými postupy mnoho problémů neumíme vůbec řešit. Přitom příroda má v zásobě spoustu poměrně silných a obecných technik pro řešení problémů, nepodobných čemukoli, na co jsme v informatice zvyklí. A při pohledu na výsledky se zdá, že fungují docela dobře. Patří mezi ně třeba právě proces evoluce (který je vlastně takovým algoritmem na hledání nejschopnějších forem života).

Nabízí se tedy otázky: „Mohly by nám tyto techniky nějak pomoci při řešení těžkých algoritmických problémů?“, „Dají se jednoduše popsat, nebo dokonce naprogramovat?“, „Dokážeme naprogramovat mravence, kteří by společně namísto stavění mraveniště hledali cestu v grafu?“, „Může se i počítač pomocí simulace neuronů něco naučit?“ a tak dále. Ukazuje se, že odpovědí na většinu těchto otázek je: „Ano, je to možné!“

To všechno vypadá skvěle! Ale možná si teď někteří z vás pro sebe řekli: „Budu já tomu rozumět? Já biologii moc neumím.“ Tak toho se přesně nemusíte bát. Všechny algoritmy, kterým se během seriálu budeme věnovat, se přírodou pouze inspirují. To znamená, že sledují nějaké její základní chování, a pak si jej vysvětlí svým informatickým způsobem. Tedy prakticky žádné biologické znalosti nejsou potřeba.

Evoluční algoritmy – část 1

Prvních několik dílů seriálu se budeme věnovat *evolučním algoritmům*. V tomto díle si konkrétně popíšeme a naučíme se používat vůbec první typ: takzvaný *genetický algoritmus*. Řekneme si, čím je motivován, podrobně popíšeme jeho hlavní části a pokusíme se pomocí něj vyřešit pár problémů. Do otázek ohledně toho, proč by takový algoritmus vůbec měl mít šanci fungovat, zatím nebudeme příliš zabíhat – ty si necháme na příště.

S genetickým algoritmem poprvé přišel John Holland v roce 1970. Genetický algoritmus je inspirován myšlenkou evoluce. V přírodě platí pravidlo „silnější přežijí,“ což znamená, že nejsilnější jedinci obstojí v konkurenci ostatních, reprodukují se a zvládnou tak přenést své geny do dalších generací. Tím v každé další generaci dostáváme lepší jedince, protože nám zůstanou geny jen těch, kteří dokázali přežít.

Dále budeme předpokládat, že výkonost jedince ovlivňují pouze jeho geny a nic jiného (tomu se v biologii říká *darwinismus*). Tento předpoklad znamená, že při reprodukci jsou potomci a jejich vlastnosti závislí pouze na genech rodičů a ne na jejich životních zkušenostech. Nyní se pojďme podívat, jak vypadá nějaký informatický genetický algoritmus.

Genetický algoritmus sestává z populace *jedinců*, funkce ohodnocující výkonost těchto jedinců (*fitness funkce*) a tří hlavních genetických operátorů pro manipulaci s jedinci: *selekce*, *křížení* a *mutace*. V dalším textu si tyto jednotlivé části podrobněji popíšeme.

Jedinec je tvořen posloupností genů, která představuje nějaké řešení našeho problému. Je důležité, aby tato posloupnost měla danou fixní délku. Zatím navíc budeme předpokládat, že tato posloupnost je binární (skládá se pouze z 0 a 1). Pro binární soustavu dokážeme jednoduše popsat další operátory.

Fitness funkce ohodnocuje výkonost (kvalitu) jednotlivých jedinců. Jinými slovy říká, jak jsou jedinci dobří pro řešení našeho problému. Zpravidla vyhodnocuje tak, že vyzkouší, jak dobře genetický kód jedince řeší zadaný problém a ohodnotí jej reálným číslem.

Selekce

Pomocí *selekce* vybíráme, které jedince použijeme pro vytvoření další populace. Bereme přitom v úvahu fitness jedinců, ale zároveň ponecháváme i určitou míru náhody. My si uvedeme dva druhy selekcí: *ruletovou selekci* a *turnajovou selekci*.

Ruletovou selekci si představíme jako opravdovou ruletu. Máme kruh rozsekaný na různě velké části, kde každá odpovídá jednomu jedinci. Velikost části kruhu je přímo úměrná fitness jedince. Do rulety pak hodíme kuličku a vybereme toho jedince, v jehož části kulička skončí. Tento proces opakujeme tolikrát, kolik jedinců potřebujeme vybrat. Nevadí nám, pokud jednoho jedince vybereme vícekrát.[2]

V praxi je ruletová selekce počítána tak, že se vygeneruje náhodné číslo od 1 do součtu všech fitness a podle toho se vybere příslušný jedinec. Z toho důvodu je pro ruletovou selekci nutné, aby fitness funkce byla vždy kladná.

Pak ale existuje ještě *turnajová selekce*, u které hodnoty fitness funkce mohou být libovolné. Ta funguje tak, že vezme dva náhodné jedince a z nich vybere toho s lepší fitness. To opět zopakuje tolikrát, kolik chceme vybrat jedinců. (Turnajová selekce nemusí vždy vybírat jen lepšího ze dvou jedinců, ale klidně obecně nejlepšího z k jedinců.)

Křížení

Křížení je jedním z nejdůležitějších genetických operátorů. To vezme dva jedince a nějakým způsobem je zkombinuje. Nejčastěji se používá takzvané *jednobodové křížení*, které vybere náhodný bod, tam jedince rozpůlí a prohodí jejich druhé části. Obdobně také funguje *dvoubodové křížení*, které náhodně zvolí dva body a pak prohodí tu část jedinců, která je mezi nimi.

Také se může použít křížení, které se pro každý bit zvlášť rozhodne, zda jej prohodí nebo ne. Takové křížení se ale pro některé problémy moc nepoužívá. Později si sami můžete vyzkoušet, které z křížení vám bude fungovat lépe.

Křížení se neaplikuje na všech jedincích, ale probíhá s pravděpodobností p_k, která se obvykle pohybuje od 0,7 do 0,9. Křížení se často považuje za hlavní pohon genetických algoritmů. Na druhou stranu existuje mnoho verzí a odvození, které křížení vůbec nezahrnují.

Mutace

Mutace je operátor, který náhodně změní jednoho jedince. To jest každý bit s nějakou malou pravděpodobností změní. Tato pravděpodobnost je často volena kolem $1/d$, kde d je délka jedince. Taková volba pravděpodobnosti způsobí, že během mutace v průměru prohodíme právě jeden bit jedince. Mutace se na jedince aplikuje s pravděpodobností p_m, která se obvykle volí v rozmezí 0,001 až 0,05.

Poslední pojem, který si definujeme, je *generace*. Generací je myšlena populace, která existuje v jedné iteraci. Na začátku máme generaci 0, z té je pak pomocí selekce, křížení a mutace vytvořena generace 1, z té pak generace 2 a tak dále. Předchozí generace vždy celá umírá a dále se používá pouze nová generace.

Pseudokód algoritmu

Nyní jsme si představili všechny důležité části genetického algoritmu, tak se pojďme podívat, jak dohromady fungují. Zde je pseudokód algoritmu.

1. Vygeneruj náhodných n jedinců velikosti d do generace 0 a spočítej jejich fitness.
2. $t = 0$
3. Opakuj následující:
4. Pomocí selekce vyber m jedinců z generace t.
5. Na každého z těchto m jedinců aplikuj křížení s pravděpodobností p_k.
6. Na každého dále aplikuj mutaci s pravděpodobností p_m.
7. Spočítej fitness výsledných jedinců a nejlepších n prohlaš za generaci $t + 1$.
8. $t = t + 1$

Jelikož využíváme náhodu, vyplatí se algoritmus pustit několikrát za sebou a ze všech běhů vzít ten nejlepší výsledek. Při každém běhu začínáme s jinak nagenerovanou počáteční populací, a tedy můžeme získat i jinak dobré řešení.

A to je celé. Poslední, co zbývá říct, je, kdy se algoritmus zastaví. To může být buď ve chvíli, kdy vyvineme jedince reprezentujícího optimální řešení problému (tj. s maximální možnou fitness, pokud ji známe) nebo po určitém počtu iterací. Často se také zohledňuje počet vyhodnocení fitness funkce, protože právě ta bývá tou časově nejnáročnější částí algoritmu. Ta se ale v našem případě pustí v každé iteraci právě m-krát, takže výpočet budeme řídit počtem iterací a velikostí populace. ($m \geq n$, ale často $m = n$)

Elitismus

Než si algoritmus vyzkoušíme, zmíníme ještě poslední věc. V algoritmu, tak jak jsme jej popsali, se lehce může stát, že během přesunu na další generaci přijdeme o nejlepšího jedince – můžeme jej nevybrat, může se špatně zkřížit a může zmutovat. Z tohoto důvodu se do algoritmu přidává ještě další vlastnost, která se jmenuje *elitismus*. Ta funguje tak, že do výběru pro generaci $t + 1$ přidáme ještě $p_e \cdot n$ nejlepších jedinců z generace t. Tím určitě zachováme doposud nejlepší geny. Hodnota p_e se obvykle volí maximálně 0,1. Nemůžeme brát moc velkou část, protože pak by nám celá populace postupně konvergovala jen k jednomu aktuálně nejlepšímu jedinci.

[2] V přírodě by to sice nešlo, ale my si to v informatice klidně můžeme dovolit a jednoho jedince si nakopírovat, kolikrát chceme.

Nyní si algoritmus pojďme vyzkoušet. Protože genetické algoritmy obsahují mnoho parametrů, které se musí správně nastavit, aby dobře fungovaly, ladí se nejdříve na jednoduchých problémech. Na takových, na kterých je dobře vidět, jak algoritmus funguje, a pro které umíme efektivně vyhodnocovat fitness funkci. My se pokusíme navrhnout genetický algoritmus, který se bude snažit vyvinout posloupnost samých jedniček.

Pro takový problém je fitness funkce jednoduchá – prostě jen spočítá, kolik jedniček jedinec obsahuje. Selekce, křížení a mutace fungují přesně tak, jak je popsáno výše. Zbývá vyladit parametry: velikost populace n, počet iterací t a hodnoty p_k, p_m, p_e. Vaším úkolem teď bude si různé kombinace těchto parametrů vyzkoušet a zjistit, jak se algoritmus chová.

Úkol 1 [6b]: Pomocí genetického algoritmu vyviňte posloupnost samých jedniček. Tedy začněte s populací náhodných jedinců a pomocí genetického algoritmu se snažte vyvinout jedince, který je složen pouze z jedniček.

Vyzkoušejte to pro velikosti jedince $d = 20, 100, 500$. Zkuste různé velikosti populace a různé kombinace pravděpodobností. Jak se algoritmus chová?

Sledujte, jak se během výpočtu mění maximální a průměrná fitness generací.

Vyzkoušejte si také napsat nějaký vlastní způsob křížení nebo mutace. Jak to bylo úspěšné? Tato křížení a mutace by neměly nijak záviset na znalosti řešeného problému. Cílem je odladit operátory, které by pak mohly fungovat i pro jiné, už ne tak jednoduché problémy.

Vyzkoušejte také, jak je algoritmus úspěšný, pokud má vyvinout jedince, kde se 0 a 1 střídají. (Přičemž je jedno, čím se začne.) Mělo by stačit změnit fitness funkci. Funguje váš algoritmus stále stejně dobře?

Během řešení můžete použít naše šablony genetického algoritmu ze stránky `http://ksp.mff.cuni.cz/viz/evoluce`.

Odevzdávejte soubor typu ZIP s popisem řešení, průběhem algoritmu a pokud chcete, tak i se zdrojovým kódem. V popisu rozeberte, co jste zkoušeli a jaké jste použili parametry, aby algoritmus byl co nejlepší.

Průběhem algoritmu je myšlen textový soubor, kde na každém řádku budou mezerou či tabulátorem oddělené hodnoty: číslo generace, hodnota průměrné fitness, hodnota maximální fitness. Nemusíte logovat každou generaci, stačí každá desátá.

Tím jsme si vyzkoušeli, jak se genetický algoritmus ladí na jednoduchém problému. Často při vymýšlení nového operátoru či dokonce celého algoritmu se hodí jej nejdříve vyzkoušet a odladit na něčem takto jednoduchém. To se především týká operátorů, které jsou nezávislé na řešeném problému. O takové operátory se snažíme, protože je pak můžeme aplikovat i na složitější problémy.

Někdy ale můžeme znalost řešeného problému využít a přímo ji zahrnout do genetických operátorů, jako jsou křížení nebo mutace. To na jednu stranu může značně urychlit výpočet, na druhou stranu nás to ale může v řešení zahnat někam do suboptimálního řešení, ze kterého se nebudeme moci dostat.

To vše si vyzkoušíme na problému sedmi loupežníků. Skupinka sedmi loupežníků vyloupila vesnici a získala z ní dohromady d předmětů, každý z nich ohodnotila nějakou cenou. Vaším úkolem je tyto předměty rozdělit na 7 hromádek tak, aby jejich součet byl co nejbližší. Konkrétně tak, aby rozdíl nejhodnotnější a nejméně hodnotné hromádky byl co nejmenší.

Než se pustíme do řešení, musíme vyřešit několik problémů: Jak budeme kódovat jedince? Jak v tomto kódování bude fungovat křížení a mutace? Jak zvolit fitness funkci?

Jedinci budou délky d a budeme je kódovat čísly $0, 1, \ldots, 6$. Pokud na i-té pozici máme číslo 4, tak to znamená, že i-tý předmět přidělíme do hromádky 4. Křížení může fungovat stejně jako s binárními jedinci a mutace změní číslo na náhodnou hodnotu od 0 do 6 namísto překlopení bitu. To, jak dobře tyto operátory budou fungovat, je jiná otázka.

A co s fitness funkcí? V tomto případě máme za úkol minimalizovat rozdíl největší a nejmenší hromádky. Náš genetický algoritmus se ale snaží fitness funkci f maximalizovat a ne minimalizovat.

U turnajové selekce problém můžeme vyřešit jednoduše – prostě použijeme hodnoty $-f(x)$ namísto $f(x)$. Co ale dělat, pokud chceme použít ruletovou selekci? Tam všechny fitness navíc musí být kladná čísla. Máme několik možností, jak toho dosáhnout. Často se používá hodnota $1/(f(x)+1)$ namísto $f(x)$. Nebo hodnota $A-f(x)$ pro vhodně zvolené A tak, že výsledné hodnoty určitě budou kladné. Musíme ale dát pozor, aby A nebylo příliš velké, protože pak by výsledné hodnoty byly příliš blízko u sebe a z pohledu ruletové selekce byly „skoro stejné“.

Tím jsme si poradili se všemi problémy a tedy genetický algoritmus můžeme zkusit použít.

Úkol 2 [9b]: Pomocí genetického algoritmu řešte problém sedmi loupežníků pro data, která naleznete na stránce se šablonami. Data jsme vygenerovali troje: lehká, střední a těžká. Doporučujeme je řešit postupně. Tj. až si budete myslet, že máte dost dobré řešení pro lehká data, zkuste, jak vám algoritmus funguje pro střední, atd.

Na prvním řádku dat jsou dvě celá čísla: počet loupežníků (vždy 7) a počet nakradených věcí D. Na druhém řádku je D mezerou oddělených čísel udávajících váhy jednotlivých věcí.

Opět zkoušejte různé kombinace parametrů. Také si vyzkoušejte, jak nejlépe převést fitness funkci na maximalizační – můžete vymyslet i vlastní způsob nebo nějaké modifikace způsobů popsaných výše.

Nalezněte co nejlepší řešení daného problému. Můžete použít i vlastní genetické operátory, které libovolně využívají znalost problému a provádějí křížení nebo mutace „cíleně“ na specifických částech jedinců.

V ZIPu odevzdejte nejlepší vyvinuté řešení společně s popisem, jak jste jej dosáhli a proč si myslíte, že takový postup funguje. Také můžete přidat záznam průběhu řešení (jako v minulé úloze).

Při řešení obou úloh můžete upravit námi vytvořenou šablonu v jazycích C++, Java, ..., anebo použít svou vlastní.

Naše kódy obsahují základní verzi genetického algoritmu se všemi jeho částmi. Navíc logují, jak se během výpočtu mění průměrná a maximální fitness, a ukládají doposud nejlepšího vyvinutého jedince.

Karel Tesař

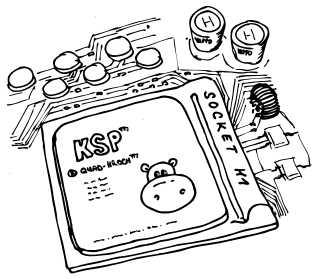
Recepty z programátorské kuchařky: Základní algoritmy

Tato naše kuchařka je nejzákladnější ze základních a je určena hlavně pro začínající řešitele. To však neznamená, že zkušenější řešitelé do ní nahlédnout nemůžou – třeba na nějakou konkrétní programátorskou techniku, kterou by si potřebovali osvěžit.

V první části kuchařky se seznámíme hlavně se základními principy programování, uchovávání dat v počítači a základy rychlé manipulace s nimi. Po přečtení této části bychom měli být schopní převést své myšlenky z hlavy na papír či do počítače a měli bychom vědět, proč je námi zvolený postup rozumný.

Druhá část nás poté seznámí se základními postupy, jak řešit určité konkrétní problémy. Naučíme se například, jak rychle vyhledávat v uspořádané posloupnosti hodnot, nebo jak si pomocí předpočítání usnadnit řešení těžké úlohy.

Většinu klíčových částí se pokusíme též ukazovat v podobě zdrojového kódu ve dvou různých jazycích (v nízkoúrovňovém C, kde je zápis blízký tomu, jak počítač doopravdy pracuje, a v Pythonu, ve kterém se píše o něco příjemněji). Nebudeme ale probírat základy syntaxe těchto jazyků, ty si případně můžete nastudovat jinde.[3] Pokud žádný z těchto jazyků neumíte, nezoufejte. KSP můžete řešit i bez toho, stačí když svá řešení důkladně slovně popíšete (konkrétní jazyk se pak můžete naučit až během dalších sérií).



Část první: Základní pojmy

Algoritmus a program

Pod tajemným slovem *algoritmus* se skrývá jen jiný výraz pro postup. Můžete si to představit jako příkaz od maminky „Běž do krámu, kup chleba, a když budou mít měkké rohlíky, tak jich vem tucet“.[4]

Takovýto příkaz klidně můžeme nazvat algoritmem, ačkoliv to bude asi znít nezvykle – pojem algoritmus se totiž používá hlavně ve světě počítačů. Je to tedy nějaká posloupnost základních příkazů, která řeší nějaký problém. Výběr konkrétního programovacího jazyka rozhoduje o tom, jaké základní příkazy budeme mít k dispozici. Většinou jsou ale skoro stejné.

Mezi základní příkazy patří:

- Manipulace s daty v paměti (uložení či načtení hodnoty, detailněji v další kapitole).
- Provedení nějakého numerického výpočtu $(+, -, *, /)$.
- Vyhodnocení určité podmínky a odpovídající větvení programu: *Pokud platí A, tak proveď B, jinak proveď C.* Přitom B i C mohou být klidně celé *bloky kódu*, tedy libovolně mnoho dalších základních příkazů.
- Opakování nějakého příkazu: *Dokud platí A, dělej B.* Takové konstrukci říkáme *cyklus* a podobně jako u podmínky může být B blok kódu, který se celý opakuje.
- Vstup a výstup programu (typicky vstup od uživatele z klávesnice či načtení vstupu ze souboru; výstup pak může znamenat vypsání výsledku na obrazovku nebo třeba zapsání dat do souboru).

Z těchto základních stavebních kamenů se skládá každý algoritmus. Programem potom rozumíme realizaci algoritmu v nějakém konkrétním programovacím jazyce.

U složitějších programů se pak často setkáte s problémem, že budete mít nějakou posloupnost příkazů, která se bude na spoustě míst programu opakovat, což zbytečně prodlužuje a znepřehledňuje kód.

Řešením tohoto problému je použití *funkcí*. Funkci si můžeme představit jako nějakou pojmenovanou část programu (s vlastní pamětí), kterou můžeme opakovaně použít tím, že ji v různých částech programu *zavoláme*. Funkci při zavolání předáme parametry (například seznam čísel), které se dostanou do její vnitřní paměti.

Funkce pak na základě obdržených parametrů může provádět nějaké operace, při kterých pracuje se svojí vnitřní pamětí (mluvíme o *lokální* paměti, změny v ní se neprojeví nikde mimo funkci). Na konci nám funkce může vrátit nějaký výsledek. Pokud funkce během svého běhu změní i nějaká data v *globální* paměti, či provede nějakou globální operaci (například výpis textu na monitor), mluvíme pak o funkci s *vedlejšími efekty* (neboli side-efekty).

Konkrétním příkladem může být funkce, která nám spočítá odmocninu ze zadaného čísla. Ta dostane jako svůj parametr číslo, uvnitř si provede nějaký výpočet, o který se jako uživatel funkce nemusíme starat, a jako výstup nám vrátí spočtenou odmocninu.

Reprezentace dat v počítači

Celkem často si v průběhu výpočtu našeho algoritmu potřebujeme pamatovat nějaké hodnoty. K tomu nám programovací jazyky dávají nástroj s názvem *proměnná*. Ta představuje jakési pojmenované místo v paměti (přihrádku), do kterého si můžeme data ukládat a pak je odtud zase načítat.

Typickým příkladem může být počítání součtu čísel, která nám uživatel zadá na vstupu. Na začátku nejdříve do nějakého místa v paměti uložíme hodnotu 0. Poté postupně, jak nám uživatel zadává čísla, tuto proměnnou pokaždé přečteme, k její hodnotě přičteme nově zadané číslo a výsledek opět uložíme na stejné místo.

Takovéto použití jedné proměnné je velmi jednoduché (tak jednoduché, že ho takto podrobně do řešení KSPčka nepište, není to potřeba), ale také celkem omezené. Co kdybychom si chtěli pamatovat třeba celou zadanou posloupnost čísel? Mohlo by nám stačit vyrobit si spoustu různě pojmenovaných proměnných, ale nejde to lépe? Jde.

Jednotlivé proměnné se mohou kombinovat do složitějších konstrukcí, které obecně nazýváme *datovými strukturami*. Zkusíme si ty nejzákladnější představit.

[3] `http://ksp.mff.cuni.cz/study/odkazy.html`

[4] A jako slušně vychovaní se tedy vydáte do krámu a koupíte tucet chlebů, protože měli měkké rohlíky :-)

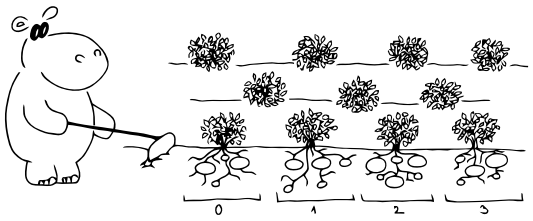
Pole

První datovou strukturou, kterou si představíme a která se na výše nastíněnou situaci náramně hodí, je *pole*. To představuje spoustu přihrádek (proměnných) naskládaných v paměti za sebou, ke kterým typicky přistupujeme přes jeden společný název pole a jejich pořadové číslo neboli index (jako `NazevPole[0], NazevPole[1], ...`).[5]

Ve většině základních jazyků je pole jen *statické*, tedy v okamžiku jeho vytváření musíme počítači říct, jak ho chceme velké. Některé vyšší jazyky ale nabízejí i pole, které se dynamicky zvětšuje, takovou konstrukci si ukážeme ve druhé části kuchařky.

Abychom nebyli omezeni jen jedním rozměrem, můžeme si klidně vyrobit pole dvourozměrné (případně obecně n-rozměrné). Dvourozměrné pole je vlastně tabulka hodnot, nazýváme ji také někdy *matice*, a může se nám hodit například při reprezentaci různých map (plán bludiště) nebo, jak uvidíme níže, pro reprezentaci dalších datových struktur.

U pole již má smysl přemýšlet, jak dlouho bude která operace trvat. Díky tomu, že jsou jednotlivé prvky v poli naskládané pevně za sebou, když se počítače zeptáme na obsah přihrádky `pole[42]`, přesně ví, na kterém místě v paměti se její obsah nachází, a proto nám hodnotu vrátí ihned.



Tomu budeme říkat *operace v konstantním čase* a budeme značit, že trvá čas $\mathcal{O}(1)$. Efektivitu programu totiž nepočítáme v sekundách (protože každý z nás má asi jinak rychlý počítač), ale v počtu základních operací, které musí program řádově vykonat. Více o časové složitosti si můžete přečíst v kuchařce o složitosti,[6] nejdříve však doporučujeme dočíst tuto kuchařku.

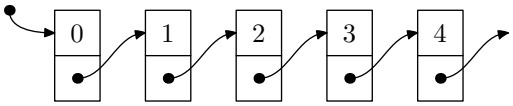
Přidání nového prvku na konec pole také zvládneme v konstantním čase. Problém je přidání nového prvku někam doprostřed (což se nám typicky stane, pokud budeme chtít udržovat hodnoty v poli seřazené a zároveň do něj vkládat nové). V takovém případě se totiž všechny prvky za vkládaným musí posunout o jednu pozici dál, aby se vkládaný prvek vešel na své místo. Taková operace tedy může pro pole délky N prvků trvat řádově až N kroků, což zapisujeme jako $\mathcal{O}(N)$ a říkáme, že je to vzhledem k N *lineární časová složitost*.

To je docela značná nevýhoda oproti struktuře, kterou si ukážeme za chvíli. Určitě ale pole nezavrhujme. Je to základní datová struktura, která nalezne použití ve spoustě programů, a jak si ve druhé části kuchařky ukážeme, můžeme ho použít třeba k rychlému hledání hodnoty metodou *binárního vyhledávání*. Nyní ale již slibovaná další datová struktura.

Spojový seznam a ukazatele

Pole jsme měli v paměti určené jenom tím, že počítač věděl, kde je jeho začátek a kolik místa v paměti zabírají jeho prvky. Při dotazování na konkrétní index pak podle indexu a podle velikosti prvků počítač přesně věděl, kam do paměti se má podívat, aby našel námi požadovaný prvek (to vše zvládl v konstantním čase). Jednotlivé prvky si tedy vůbec nemusely pamatovat, kde se nachází jejich sousedi, protože všechny prvky seděly v paměti za sebou.

Představme si ale teď situaci, kdy by si každý prvek ještě pamatoval pozice sousedů. Pak bychom mohli mít prvky libovolně rozházené v paměti a jen by se na sebe vzájemně odkazovaly (první prvek by tvrdil, že druhý je na pozici X, druhý by tvrdil, že třetí je na pozici Y, a tak dále).

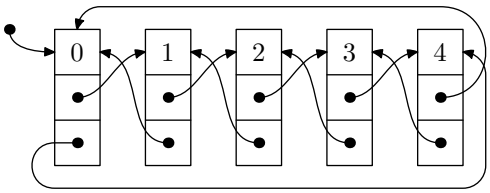


K lepšímu pochopení tohoto principu je důležité si vysvětlit, co to je *ukazatel* (nebo také *odkaz* či anglicky *pointer*). Každé místo v paměti počítače má své číselné označení, kterému říkáme *adresa*. Když si vytváříme nějakou pojmenovanou proměnnou, ta se vlastně odkazuje na určité místo v paměti a na tomto místě v paměti je její hodnota.

Co kdyby ale hodnota proměnné byla adresa nějakého jiného místa v paměti? Pak takové proměnné říkáme *pointer* a umožňuje nám vytvářet výše popsanou strukturu rozházených prvků v paměti.

Spojový seznam je tedy určený svým prvním prvkem (máme v jedné proměnné pointer na tento prvek, který se často nazývá *kořen*, protože z něj „vyrůstá“ zbytek struktury) a poté u každého dalšího prvku máme za sebou uloženou hodnotu tohoto prvku a odkaz (pointer) na další prvek. Odkazy mezi prvky mohou být i obousměrné, mohou vést dokola (poslední ukazuje na první) či mohou dokonce tvořit nějakou složitější strukturu (pak to ale již nebude čistý spojový seznam).

Pokud pointer nemá nikam ukazovat, realizuje se to odkázáním tohoto pointeru na adresu `NULL`. To skoro doslovně říká „Neukazuji nikam“.



Co nám takto vystavěná struktura umožňuje v porovnání s polem? Přístup na konkrétní prvek v ní stojí lineárně času, protože ho musíme „odkrokovat“ od prvního prvku (na který máme pointer), tedy musíme udělat až $\mathcal{O}(N)$ kroků. Pokud bychom však pointer na daný prvek už nějak měli, samozřejmě na něj můžeme přistoupit v konstantním čase.

Naopak přidávání prvků na konkrétní místo (i jejich odebírání) máme v podstatě zadarmo a spojový seznam můžeme rozšiřovat, dokud na něj máme v počítači paměť. Ve chvíli, kdy chceme přidat nový prvek za prvek, na který máme pointer, jen šikovně přepojíme ukazatele. Pokud předtím ukazatele vedly $A \to B$, teď povedou $A \to C \to B$ (a při odebírání naopak).

Zde můžete vidět ukázku pointerů a spojových seznamů v jazyce C, kde jsou tyto věci mnohem více nízkoúrovňové (ale zato rychlejší):

[5] Pozor, ve světě počítačů se velmi často indexuje od nuly, tedy první prvek má v tomto případě index 0.

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

```c
#include <stdio.h>
#include <stdlib.h>
// Příkazy výše načetly do programu
// standardní knihovny a funkce z~nich.

// Struktura pro prvek obsahující dopředné
// i zpětné odkazy. Zkráceně tomuto typu
// budeme říkat "tprvek".
typedef struct prvek tprvek;
struct prvek {
    int hodnota;
    tprvek *dalsi;
    tprvek *predchozi;
};

// Vytvoří nový prvek:
tprvek *novy(int i) {
    tprvek *aktualni =
        malloc(sizeof(tprvek));
    aktualni->dalsi = NULL;
    aktualni->predchozi = NULL;
    aktualni->hodnota = i;
    return aktualni;
}

// Odstraní prvek a vrátí pointer na další
// prvek (vrácení pointeru se hodí při
// odstraňování kořene):
tprvek *odstran(tprvek *aktualni) {
    if (aktualni->predchozi != NULL)
        aktualni->predchozi->dalsi =
            aktualni->dalsi;
    if (aktualni->dalsi != NULL)
        aktualni->dalsi->predchozi =
            aktualni->predchozi;

    tprvek *pomocna = aktualni->dalsi;
    free(aktualni);
    return pomocna;
}

// Vloží a vrátí pointer na nový prvek:
tprvek *vloz_za(tprvek *aktualni, int i) {
    tprvek *pomocna = aktualni->dalsi;
    aktualni->dalsi = novy(i);

    if (pomocna != NULL)
       pomocna->predchozi = aktualni->dalsi;

    aktualni->dalsi->dalsi = pomocna;
    return aktualni->dalsi;
}

// Použití:
int main(void) {
    tprvek *koren = novy(1);
    tprvek *aktualni = vloz_za(koren, 2);

    aktualni = koren;
    while (aktualni != NULL) {
        printf("%d\n",aktualni->hodnota);
        aktualni = aktualni->dalsi;
    }

    return 0;
}
```

Zde je ukázka spojových seznamů v Pythonu, kdybychom si je podobně jako v C chtěli naprogramovat sami (Python totiž obsahuje spoustu základních struktur již hotových, podívejte se na modul jménem `collections`):

```python
class Prvek:
    def __init__(self, hodnota):
        self.hodnota = hodnota
        self.dalsi = None
        self.predchozi = None

class Spojak:
    def __init__(self):
        self.koren = None

    def Vypis(self, aktualni):
        if aktualni is not None:
            print aktualni.hodnota
            self.Vypis(aktualni.dalsi)

    def VlozPo(self, prvek, zaPrvek = None):
        if zaPrvek is not None:
            prvek.dalsi = zaPrvek.dalsi
            prvek.predchozi = zaPrvek
            zaPrvek.dalsi = prvek
        if prvek.dalsi is not None:
            prvek.dalsi.predchozi = prvek
        if self.koren is None:
            self.koren = prvek

    def Odstran(self, prvek):
        if prvek.predchozi is not None:
            prvek.predchozi.dalsi = \
                prvek.dalsi
        if prvek.dalsi is not None:
            prvek.dalsi.predchozi = \
                prvek.predchozi

# Použití:
prvekA = Prvek("A")
prvekB = Prvek("B")
prvekC = Prvek("C")
prvekD = Prvek("D")

seznam = Spojak()

seznam.VlozPo(prvekB)
seznam.VlozPo(prvekD, prvekB)
seznam.VlozPo(prvekC, prvekD)
seznam.VlozPo(prvekA, prvekC)
seznam.Odstran(prvekC)

seznam.Vypis(seznam.koren)
```

Fronta a zásobník

S použitím spojových seznamů (nebo v jednodušším případě dokonce i polí) můžeme zkonstruovat dvě velmi užitečné datové struktury, frontu a zásobník.

Fronta funguje tak, jak si ji asi každý z nás představuje: ten, kdo se do fronty postaví první, ten také první přijde na řadu. Můžeme si ji také představit jako trubku, do které na jedné straně sypeme nějaké věci a na druhé je odebíráme. Anglicky je též nazývaná *FIFO* (*„First In, First Out“*).

Praktickou realizaci uděláme jednoduše spojovým seznamem. Budeme si držet dva ukazatele, jeden na začátek seznamu, druhý na konec. Když se objeví nový prvek, který do fronty budeme chtít vložit, přidáme ho na konec, zatímco při odebírání z fronty využijeme druhého ukazatele a vezmeme prvek ze začátku.

Druhou velmi podobnou datovou strukturou je *zásobník*. Jak už ale plyne z anglického názvu *LIFO* (*„Last In, First Out“*), funguje spíše jako plný šuplík: Nahoru na něj přidáváme nové prvky, a když chceme nějaký odebrat, vezmeme

také zvrchu. To znamená, že první se na řadu dostane naposledy vložený prvek.

Implementace je velmi obdobná jako u fronty, jen bude ukazatel pouze jeden a bude ukazovat jenom na jeden konec spojového seznamu.

Knihovny

Tyto základní struktury už jsou často předpřipravené jako součást nějakých *knihoven* v daném jazyce. Knihovna je většinou sbírka nějakých navzájem souvisejících funkcí, které již někdo sepsal a které si můžeme do našeho programu načíst a používat. Ukázku načtení knihoven můžete vidět například ve výše zmíněném kódu v jazyce C.

Je ale velmi důležité rozumět tomu, jak knihovní funkce vnitřně fungují. Protože jedině když budeme vědět, co je jak rychlé a efektivní, budeme schopni psát rychlé programy.

Teď již víme, jak reprezentovat nejzákladnější datové struktury v počítači, ale mohlo by se nám hodit zastavit se ještě chvíli u dalších struktur. Tentokrát je už budeme studovat trochu teoretičtěji.

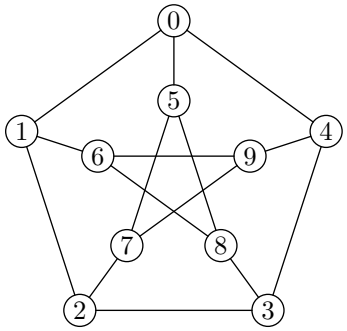
Stromy a grafy v informatice

Grafy

S nějakými grafy jste se již možná potkali, ale tento pojem je bohužel docela přetěžovaný. Jedním jeho významem jsou „koláčové grafy“ a jiné další diagramy znázorňující nějaký poměr (ať už to jsou výsledky voleb, nebo poměr lidí, kteří sledovali v televizi Večerníček).

Další význam můžeme nalézt v analytické matematice, kde se potkáme s grafy průběhu nějakých funkcí. My však nemáme na mysli ani jedno z výše zmíněných, teď se budeme bavit o *kombinatorických grafech*.

Grafem tedy máme na mysli nějakou množinu objektů, říkejme jim *vrcholy*, a nějaké vztahy mezi nimi. Tyto vztahy nazýváme *hranami* a jsou vyjádřené dvojicemi vrcholů, mezi kterými vedou. Ukázku takového grafu vidíme třeba na následujícím obrázku.



Jako praktickou ukázku grafu si můžeme například představit silniční síť nějakého státu: vrcholy budou města a hrany budou silnice, které mezi nimi vedou.

Občas se můžete setkat s pojmem *souvislý* graf. Ten znamená jen to, že mezi každými dvěma vrcholy existuje nějaká cesta. Pokud tomu tak není, je graf *nesouvislý* a dá se rozložit na několik menších grafů, které již souvislé jsou a říká se jim *komponenty souvislosti*.

Samotný graf poté můžeme doplnit tím, že si v každém vrcholu nebo na každé hraně budeme pamatovat nějakou hodnotu (například cenu nejlevnějšího benzínu ve městech a délku v kilometrech na silnicích). Pamatování si hodnot ve vrcholech je docela obvyklá technika a nemá speciální název, ale pokud budeme mít graf, který si pamatuje hodnoty na hranách, budeme o něm mluvit jako o *ohodnoceném grafu*.

Další možnou úpravou je, že každá hrana povede jen jedním směrem (jednosměrné silnice), takovým grafům říkáme *orientované* (pokud pak v orientovaném grafu chceme silnici oběma směry, prostě do něj přidáme dvě hrany, jednu v každém směru).

Poslední, co nám schází k praktickému použití grafů, je naučit se, jak je reprezentovat v počítači. Existuje několik možností (n bude značit počet vrcholů, m počet hran):

- **Seznam sousedů** – vrcholy grafu budeme mít uložené v poli a u každého vrcholu budeme mít (spojový) seznam čísel dalších vrcholů, do kterých z aktuálního vrcholu vede hrana. Zabírá místo $\mathcal{O}(n+m)$ a hodí se pro řídké grafy (tedy grafy, kde je m řádově stejné jako n).
- **Matice sousednosti** – tabulka $n \times n$, kde na souřadnicích $[i, j]$ je jednička (případně jiná hodnota, v případě ohodnoceného grafu), pokud z i do j vede hrana, a nula, pokud tam hrana není (u neorientovaných grafů je navíc matice symetrická – je jedno, jestli vezmeme $[i, j]$ nebo $[j, i]$). Hodí se pro husté grafy, kde $m \sim n^2$.
- **Matice incidence** – řádky reprezentují vrcholy, sloupce hrany. V každém sloupci jsou právě dvě jedničky – indexy vrcholů, mezi kterými hrana vede. Zabírá však $\mathcal{O}(mn)$ a její použití bývá dost neohrabané, takže je většinou lepší dát přednost jiné reprezentaci grafu. Je však dobré o ní vědět.

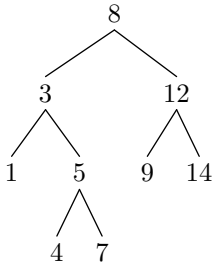
Grafy jsou velmi široké téma. Můžeme hledat jejich minimální kostry, můžeme v nich hledat nejkratší cesty či skrze ně pouštět pod tlakem vodu. Více o nich si tedy můžete přečíst v některé z našich specializovaných grafových kuchařek, které odkazujeme z našeho kuchařkového rozcestníku.[7]

Stromy

Možná si říkáte, co má informatika u všech elektronů společného s lesnictvím? Kupodivu celkem mnoho a bez stromů bychom se v leckterém případě jen těžko obešli. Informatické stromy sice nejsou většinou tak zelené, mají ale, na rozdíl od svých dřevnatých sourozenců, mnoho jiných pěkných vlastností.

Strom je vlastně speciálním případem souvislého grafu, který neobsahuje žádnou kružnici (cyklus). To znamená, že mezi každými dvěma vrcholy stromu existuje právě jedna cesta.

Díky této vlastnosti můžeme nějaký zvolený vrchol prohlásit za *kořen* a strom za něj pomyslně zavěsit (tak, že strom roste od kořene směrem dolů), této operaci se říká *zakořenění*. Pak můžeme mluvit o tom, že z kořene směrem dolů (informatické stromy mají tradičně kořen nahoře) vyrůstají nějaké *podstromy*.



[7] `http://ksp.mff.cuni.cz/study/cooks/`

Pokud je strom zakořeněný, můžeme v něm mluvit o *hloubce* každého vrcholu, neboli o jeho vzdálenosti od kořene. Hloubka celého stromu je pak nejdelší ze vzdáleností od kořene k nějakému z *listů* (tak říkáme vrcholům, které již nemají žádné *syny*, tedy vrcholy, které by z nich vyrůstaly). Podle hloubky poté můžeme vrcholy stromu uspořádat do jednotlivých *hladin*.

Velmi často používáme stromy, které jsou nějak pravidelné. Příkladem jsou třeba *binární stromy*, které mají v každém vrcholu maximálně dva syny (říkáme jim *levý a pravý podstrom*). Reprezentovat se dají buď obecně jako každý jiný strom (v každém vrcholu spojový seznam podstromů), nebo velmi pěkně i v poli.

Stačí si pomyslně doplnit binární strom na *úplný* (to je takový, který má všechny své hladiny plné) a pak ho od kořene směrem dolů po hladinách očíslovat (kořen dostane číslo nula, jeho synové čísla jedna a dva, další hladina čísla tři až šest, atd.).

Můžeme si všimnout, že pokud si v takovém očíslování vezmeme jakýkoliv vrchol s číslem (indexem) i, tak jeho synové jsou právě vrcholy s indexy $2i+1$ a $2i+2$. Do pole níže je zapsaný binární strom z obrázku výše.

index	0	1	2	3	4	5	6	7	8	9	10
hodnota	8	3	12	1	5	9	14	–	–	4	7

Jak plyne z očíslování, pro úplný binární strom je uložení v poli efektivní a neplýtváme místem. Pokud ale strom úplný nebude, zůstanou nám v poli volná místa. Uložení v poli se tedy vyplatí jen pro stromy, které se od úplných příliš neliší.

Speciálním případem binárních stromů jsou pak ještě *binární vyhledávací stromy*. Jsou to normální binární stromy, pro něž navíc platí, že ať si vezmeme libovolný vrchol, budou hodnoty vrcholů v jeho levém podstromě menší než hodnota tohoto vrcholu, a hodnoty v jeho pravém podstromě naopak větší.

V takovém stromě pak zvládneme snadno vyhledávat. Budeme ho postupně procházet od kořene a v jednotlivých vrcholech budeme porovnávat hledanou a aktuální hodnotu a podle toho sestupovat do správného podstromu. Podobná technika je detailněji popsaná ve druhé části kuchařky, v kapitolce *Rozděl a panuj*.

Na složitější datové struktury stavějící na těchto základních (haldy, intervalové stromy, ...) se můžete podívat do některé z našich dalších kuchařek, na jejichž přehled jsme vás už odkázali o kapitolu výše.

Část druhá: Programátorské techniky

Tato část by měla sloužit jako rychlý přehled a ukázka různých technik, které se dají použít při řešení úloh z KSPčka, nebo při programování obecně.

Rekurze

Rekurze je velmi důležitá programátorská technika. V podstatě znamená definování nějaké věci (ať už je to nějaký objekt či postup výpočtu) pomocí sebe sama.

Rekurzivně může být například zadána nějaká datová struktura. Například stromy jsou pěkným příkladem rekurzivně definované datové struktury – každý vrchol stromu může mít syny, a každý z těchto synů je sám o sobě strom (tedy i osamocený vrchol bez synů je stromem).

Prakticky je to realizováno tak, že každý vrchol má svou hodnotu a pak ještě seznam ukazatelů vedoucích na další případné podstromy. S ukazateli jsme se již potkali a s jejich pomocí jsme si postavili spojový seznam. A přesně tak, spojový seznam je také ve své podstatě rekurzivní datová struktura.

Mimo rekurzivních datových struktur se ale často potkáváme i s rekurzivním postupem výpočtu nějakého programu, nejčastěji realizovaným ve formě funkce, která volá sama sebe (většinou s jinými parametry, jinak by to asi nemělo smysl), takové funkci se říká *rekurzivní funkce*.

U rekurzivních funkcí je nejdůležitější věc definovat nějakou *koncovou podmínku*, tedy podmínku, při níž už se rekurze zastaví. Jinak by se totiž mohlo stát, že by rekurze běžela donekonečna.

Přesněji rekurze by se i tak v nějakou chvíli zastavila, ale skončila by chybou, protože by jí došla paměť – každá volání funkce si totiž ukousne kus paměti (musí si pamatovat, kam se po skončení má vrátit) a pokud má rekurzivní funkce ještě nějaké lokální proměnné, musíme si někde uložit všechny lokální proměnné funkcí, z kterých jsme se doposud nevrátili.

Rekurzivní funkce a převod na nerekurzivní cyklus

Typickým příkladem rekurzivní funkce je výpočet Fibonacciho čísel. Ta jsou definována tak, že $f_0 = 1$, $f_1 = 1$ a n-té Fibonacciho číslo je součtem dvou předchozích ($f_n = f_{n-1} + f_{n-2}$). To nám dává posloupnost čísel 0, 1, 1, 2, 3, 5, 8, 13, ... pokračující donekonečna. Pokud toto přepíšeme do programového kódu, tak dostáváme následující zápisy:

V jazyce C:

```
int fib(int n) {
        if (n==0) return 0;
        else if (n==1) return 1;
        else return fib(n-1) + fib(n-2);
}
```

V Pythonu:

```
def fib(n):
        if n==0:
                return 0
        elif n==1:
                return 1
        else:
                return fib(n-1) + fib(n-2)
```

Jak vidíme, je přepis celkem přímočarý. Pokud by se nám však rekurze v nějakém případě nelíbila, můžeme se každé rekurze zbavit. Rekurzivní volání totiž můžeme šikovně přepsat na nějaký cyklus se *zásobníkem*.

Pak jen v cyklu odebíráme prvky ze zásobníku, dokud není prázdný, a za každé rekurzivní zavolání do zásobníku přidáme parametry, se kterými bychom naši funkci volali. Tímto

postupem převedeme každou rekurzivní funkci na nerekurzivní.

Ještě doplníme poznámku, že ve většině programovacích jazyků každé volání funkce stojí nějaký čas, sice malý, ale když se volání provádí opakovaně, tak se to už nasčítá. Pro reálnou implementaci je tedy nejlepší pokusit se rekurzi převést na nerekurzivní volání, pokud to nějak rozumně jde.

Občas to jde dokonce i jednodušeji a bez zásobníku. Podívejte se na alternativní variantu výpočtu Fibonacciho čísel níže a rozmyslete si, co dělá.

V jazyce C:

```
int fib2(int n) {
        if (n==0) return 0;
        else if (n==1) return 1;

        int a = 0; int b = 1;
        for(int i = 2; i<=n; i++) {
                int c = a + b;
                a = b;
                b = c;
        }
        return b;
}
```

V Pythonu:

```
def fib2(n):
        if n==0:
                return 0
        elif n==1:
                return 1

        a = 0; b = 1
        while n>1:
                (a, b) = (b, a+b)
                n-=1
        return b
```

Jak vidíte, je i tato funkce elegantní a navíc běhá mnohem rychleji, než její rekurzivní varianta. Tato funkce běhá v $\mathcal{O}(n)$, kdežto rekurzivní varianta počítala stejné věci mnohokrát dokola (zkuste si nakreslit nějaký strom volání předchozí funkce, případně se podívat dopředu do kapitoly Předpočítané mezivýsledky).

Rekurzivní varianta tedy běžela až v čase $\mathcal{O}(2^n)$, což je pro velká n mnohem pomaleji než $\mathcal{O}(n)$ (avšak šla by celkem snadno zachránit, aby běžela také v $\mathcal{O}(n)$, zkuste si rozmyslet jak).

Backtracking

S rekurzí silně souvisí i pojem *backtracking*, česky by se snad dalo říci „metoda pokusu a omylu“. Tímto pojmem označujeme proces, kdy postupně zkoušíme všechny možnosti, jak vyřešit nějaký problém.

Metoda pokusu a omylu se tento proces nazývá proto, že pokud již nemůžeme pokračovat dál (třeba v případě, že v bludišti dojdeme do slepé uličky), vrátíme se kus zpět a zkusíme jinou (zatím nevyzkoušenou) možnost. Takto postupně zkusíme každou možnost, a buď nalezneme námi hledané řešení, nebo se vrátíme až na výchozí pozici a zjistíme, že řešení neexistuje.

Backtracking bývá často realizován pomocí rekurze, ukážeme si to na příkladu hledání rozkladu zadané částky na mince o hodnotách 5 Kč a 3 Kč (všimněte si, že v takto omezeném peněžním systému nejde složit třeba částka 7 Kč). Naše funkce dostane jako parametr zbývající částku a zkusí rekurzivně provést rozklad na jednotlivé mince:

V jazyce C:

```
bool rozloz(int castka) {
        // Koncová podmínka rekurze
        if (castka == 0) return true;
        else if (castka < 0) return false;
        else if (rozloz(castka-5)) {
                printf(" 5 Kc");
                return true;
        } else if (rozloz(castka-3)) {
                printf(" 3 Kc");
                return true;
        } else return false;
}
```

V Pythonu:

```
def rozloz(castka):
        if castka == 0:
                return True
        elif castka < 0:
                return False
        elif rozloz(castka-5):
                print " 5 Kc"
                return True
        elif rozloz(castka-3):
                print " 3 Kc"
                return True
        else:
                return False
```

V každém kroku zkusíme nejdříve použít pětikorunovou minci a zavoláme se na zbylou částku, a když náš rozklad nevyjde, zkusíme v tomto kroku použít ještě tříkorunu. Takto se rozhodujeme v každém kroku rekurze a případně se vracíme z neúspěšných větví výpočtu a zkoušíme další možnosti.

Takovým postupem ale vyzkoušíme až exponenciálně mnoho možností ($\mathcal{O}(2^n)$), což není moc rychlé. Proto je doporučováno se backtrackování raději vyhnout, nebo ho nějak chytře vylepšit. Je však dobré o backtrackování vědět, protože existují problémy, které efektivněji řešit neumíme.

Rozděl a panuj

Jednou ze základních technik je rozdělení složitějšího problému na menší části, které opět můžeme rozdělit na menší a tak dále, dokud se nedostaneme k problémům tak malým, že je už umíme triviálně vyřešit.

Binární vyhledávání v poli

Představte si, že máme seřazené pole n prvků a chceme zjistit, jestli se v něm nachází prvek s hodnotou k. Určitě můžeme projít celé pole v lineárním čase (tím, že budeme brát jeden prvek za druhým a kontrolovat, zda je roven hodnotě k), ale to je zbytečně pomalé a nevyužívá toho, že máme pole seřazené.

Můžeme totiž začít s velkým problémem a ten postupně zmenšovat na stále menší a menší. Nejdříve hledáme k v celém poli. Podíváme se na jeho prostřední prvek:

- Pokud je roven k, jsme hotovi.
- Je-li větší než k, víme, že se k musí nacházet nalevo od něj. Můžeme tedy hledat znovu, ale tentokrát se omezit jen na levou polovinu pole.

- Analogicky, je-li menší než k, můžeme hledat jen v pravé polovině.

Když tímto postupným dělením problémů na menší dojdeme až k poli o velikosti jednoho prvku, stačí tento prvek jenom porovnat, dál už se pole nepokoušíme rozdělovat.

Jelikož se nám každým krokem problém zmenší na polovinu, tak se maximálně po $\log n$ krocích dostaneme na pole velikosti jedna. Říkáme, že algoritmus má *logaritmickou časovou složitost*, píšeme $\mathcal{O}(\log n)$.[8]

Prakticky postup provádíme tak, že si udržujeme levý a pravý okraj aktuálně zpracovávaného úseku a postupně je k sobě přibližujeme.

Ukázka hlavní smyčky v C:

```
int pole[] = {1,2,5,7,12,16,42};
int hledane = 8;
int L = 0, R = 6;
int x;

do {
    int prostredni = (L+R)/2;
    x = pole[prostredni];
    if (x == hledane)
        printf("Pole obsahuje hledane\n");
    else if (x < hledane)
        L = prostredni + 1;
    else
        R = prostredni;
} while (L < R && x != hledane);

if (x != hledane)
    printf("Hledane neni v poli\n");
```

Ukázka v Pythonu jako funkce vracející index prvku nebo -1, pokud hledané číslo nenalezne:

```
def bin_vyhled(pole, hledane, L=0, R=None):
    if R is None:
        R = len(pole)
    while L < R:
        prostredni = (L+R)//2
        x = pole[prostredni]
        if x < hledane:
            L = prostredni + 1
        elif x > hledane:
            R = prostredni
        else:
            return prostredni
    return -1

# Zavolání:
print bin_vyhled([1,2,5,7,12,16,42], 8)
```

Další aplikace

Další typickou aplikací postupu rozděl a panuj je například třídění posloupnosti pomocí *Mergesortu*. Ten v základu funguje tak, že každou posloupnost, kterou dostane k setřídění, rozdělí na poloviny a každou z nich setřídí rekurzivním zavoláním sebe sama. Zanořování se zastaví ve chvíli, kdy třídíme posloupnost délky jedna (ta už je z podstaty setříděná). Pak jen v každém kroku ze dvou setříděných menších posloupností vyrobí jejich sléváním setříděnou posloupnost dvojnásobné délky.

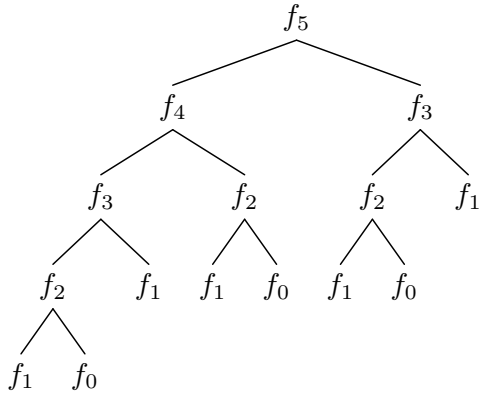
Více se o metodě Rozděl a panuj můžete dozvědět ve stejnojmenné kuchařce.[9]

Předpočítané mezivýsledky

Motivací k této kapitole je následující motto: „Proč počítat něco vícekrát, když nám to stačí spočítat jednou a zapamatovat si to?“.

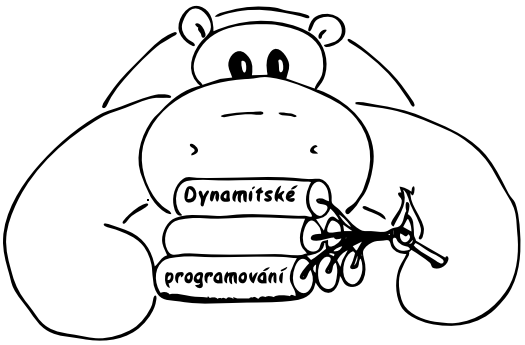
Velmi často se totiž setkáváme s tím, že něco počítáme stále dokola. Jako příklad si můžeme vzít naši rekurzivní implementaci počítání Fibonacciho čísel z kapitolky Rekurze.

Když se podíváme na výpočet čísla `fib(5)`, vidíme, že pro něj voláme `fib(4)` a `fib(3)`, `fib(4)` volá `fib(3)` a `fib(2)`, `fib(3)` volá `fib(2)` a `fib(1)` a tak dále. Všimli jste si, kolikrát se nám tyhle výpočty opakují? Některá Fibonacciho čísla spočteme totiž zbytečně mnohokrát.



Kdybychom si je namísto opakovaného počítání někde pamatovali, mohli bychom pak odpověď na dotaz na již vypočtené číslo vytáhnout jako králíka z klobouku v konstantním čase. Zavedením jednoho globálního pole, ve kterém si tyto hodnoty pro jednotlivá n budeme pamatovat, nám sníží časovou složitost z $\mathcal{O}(2^n)$ na pěkných $\mathcal{O}(n)$. Takovému postupu se obecně říká *dynamické programování*.

Dynamické programování



Nejprve uveďme na pravou váhu výraz „dynamické“ v názvu. Nevystihuje tak úplně podstatu této techniky a jeho historické pozadí je celkem složité, avšak dnes je tento název již tak zažitý, že se už pravděpodobně nezmění.

Slovo „dynamické“ částečně odkazuje na to, že se dynamicky (za běhu programu) postupně staví řešení jednodušších problémů, která jsou následně použita pro řešení složitějších. Jeho hlavní podstatou je tedy ukládání a opětovné použití již jednou vypočtených údajů.

[8] Pokud není řečeno jinak, znamená pro nás v informatice značka log *dvojkový logaritmus*, což je funkce opačná k funkci 2^n a roste o hodně pomaleji než funkce lineární. Pro velká n platí: $1 < \log n < n$ a například $\log 2 = 1, \log 8 = 3, \log 1024 = 10$.

[9] `http://ksp.mff.cuni.cz/viz/kucharky/rozdel-a-panuj`

Hodí se na úlohy, které se dají dělit na podúlohy, které jsou si podobné a mohou se opakovat. Výsledky takovýchto podúloh si poté ukládáme a při dotazu na stejnou podúlohu vrátíme jen uložený výsledek a výpočet již neprovádíme.

Pro další prohloubení znalostí můžete na našem webu nahlédnout do další kuchařky, tentokrát nesoucí (překvapivě) název Dynamické programování.[10]

Prefixové součty

Velmi často se nám hodí si ještě před samotným výpočtem předpočítat a uložit nějaké hodnoty, které poté použijeme.

Představme si například problém nalezení souvislého úseku s největším součtem v nějaké posloupnosti kladných i záporných čísel. Že to není úplně jednoduchý příklad, si ukažme na následující posloupnosti:

$$1, -2, 4, 5, -1, -5, 2, 7$$

Máme zde dvě ryze kladné souvislé posloupnosti, každou se součtem 9 ($4, 5$ a $2, 7$). Ale přesto je výhodnější vzít i nějaké záporné hodnoty a vytvořit tak souvislou posloupnost o součtu 12 (zkuste ji nalézt).

Mohlo by nás napadnout, že prostě zkusíme vzít všechny možné začátky a všechny možné konce. To nám dává $\mathcal{O}(n^2)$ možných posloupností (máme n možných začátků a ke každému z nich řádově n možných konců), pro každou posloupnost si spočteme součet (to zvládneme v $\mathcal{O}(n)$) a budeme si pamatovat ten největší nalezený. Celý náš postup tak trvá $\mathcal{O}(n^3)$.

To není pro takhle jednoduchou úlohu zrovna ten nejpěknější čas, zkusme ho zlepšit. Ukážeme si, jak vypočítat součet libovolné posloupnosti v konstantním čase. Celý princip je vlastně až kouzelně jednoduchý, ale zároveň velmi mocný. Na začátku výpočtu si do pomocného pole P stejné délky jako posloupnost na vstupu (té říkejme S) uložíme takzvané *prefixové součty:* i-tý prefixový součet je součet prvních $i+1$ prvků S, neboli $P[i] = S[0] + S[1] + \ldots + S[i]$.

Pro náš ukázkový případ a pro vstupní pole označené S by to dopadlo takto:

i	-1	0	1	2	3	4	5	6	7
$S[i]$		1	-2	4	5	-1	-5	2	7
$P[i]$	0	1	-1	3	8	7	2	4	11

Pole prefixových součtů umíme získat v linerárním čase – prostě jen od začátku procházíme vstupní pole, počítáme si průběžný součet a ten zapisujeme.

Součet libovolného úseku $a \ldots b$ pak získáme v konstantním čase jako prefixový součet od začátku do indexu b minus prefixový součet od začátku do indexu a. Zapsáno programově to pak je:

```
soucet = P[b] - P[a-1];
```

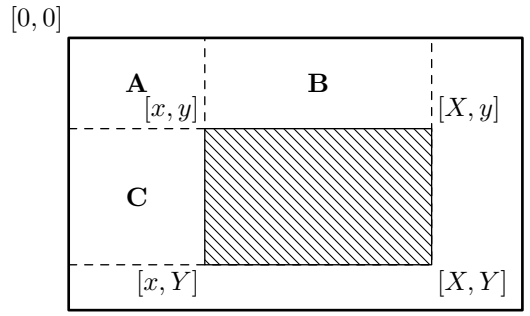
To nám umožňuje snížit čas potřebný na řešení této úlohy na $\mathcal{O}(n^2)$. To je už lepší čas, prozradíme však, že tuto úlohu lze řešit dokonce v lineárním čase, ale to je již nad rámec této kuchařky.

Dvourozměrné prefixové součty

Prefixové součty se dají zobecnit i do více rozměrů, ale princip je vždy stejný. Například dvourozměrné prefixové součty u matice fungují tak, že si předpočítáme součty podmatic začínajících levým vrchním políčkem a končící na indexu $[x, y]$.

Z toho je vidět, že prefixový součet zpravidla obsadí stejně velký prostor jako původní data, v tomto případě tedy budeme mít matici hodnot prefixových součtů končících na zadaných souřadnicích. Jak ale získat součet nějaké podmatice, která se nachází někde „uprostřed“ naší matice?

Použijeme stejný princip jako u jednorozměrného případu: Přičteme větší část, kterou chceme započítat, a odečteme od ní části, které započítat nechceme. Pro případ podmatice začínající vlevo nahoře na pozici $[x, y]$ a končící napravo dole na $[X, Y]$ to ilustruje následující obrázek:



Nejdříve přičteme celý prefixový součet končící na pozici $[X, Y]$. Tím jsme ale započítali i části A, B a C z obrázku, které započítat nechceme. Tak odečteme prefixové součty končící na indexech $[X, y]$ a $[x, Y]$. Ale pozor, teď jsme odečetli jednou $A + B$ a jednou $A + C$, tedy část A (prefixový součet končící na pozici $[x, y]$) jsme odečetli dvakrát, musíme ji proto ještě jednou přičíst.

Celý vzorec tedy vypadá takto:

```
soucet = P[X,Y] - P[X,y] - P[x,Y] + P[x,y];
```

Tento princip přičítání a odečítání se dá zobecnit i pro libovolné vyšší rozměry, ale chce to již trošku představivosti, co se má přičíst a kolikrát. Říká se tomu také *princip inkluze a exkluze* a najde použití nejen u vícerozměrných prefixových součtů.

Vyvážení délky předvýpočtu a hlavního výpočtu

Správně vyvážit, kolik času můžeme věnovat na předvýpočet a kolik času na hlavní výpočet, je velice důležitá věc a spousta i zkušenějších řešitelů v tom občas chybuje. Přitom to při troše počítání není vůbec nic složitého.

Jako první je potřeba vědět, kolikrát nám předvýpočet během běhu programu pomůže. Předvýpočtem si totiž vybudujeme za nějaký čas určitou datovou strukturu, pomocí které pak dokážeme rychle odpovídat na zadané dotazy.

Označme si počet takovýchto dotazů, které program za běhu dostane, jako Q. Buď to může být hodnota přímo ze zadání typu „Zkonstruujte datovou strukturu pro n hodnot, která zvládne rychle odpovídat na dotazy daného typu, a očekávejte řádově m dotazů“, nebo se může jednat o nějaký interní dotaz v rámci běhu programu (příklad interního dotazu je třeba výše uvedený algoritmus na hledání souvislé podposloupnosti s co největším součtem, který se za běhu ptal na součty nějakých úseků).

Dále si označme jako $\mathcal{O}_p$ čas, který nám zabere předvýpočet a jako $\mathcal{O}_q$ čas, který nám ušetří každý předvypočítaný dotaz. Celkový čas, který ušetříme, je pak vlastně $Q \cdot \mathcal{O}_q$. Pokud je tento čas řádově větší než $\mathcal{O}_p$, pak má předvýpočet smysl.

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

Naopak nemá smysl trávit předvýpočtem řádově více času, než by trval samotný výpočet bez použití předpočítaných hodnot.

Jako příklad uvažujme problém o velikosti n, u kterého máme tři možnosti, které můžeme zvolit. Můžeme buď předvýpočet úplně vynechat a na každý dotaz odpovídat v čase $\mathcal{O}(n)$, nebo provést předvýpočet v čase $\mathcal{O}(n \log n)$ a poté odpovídat na každý dotaz v čase $\mathcal{O}(\log n)$, nebo provést předvýpočet v čase $\mathcal{O}(n^2)$ a pak odpovídat v čase $\mathcal{O}(1)$ na dotaz.

Kdy se nám co hodí?

- Pokud bychom dostali jen jeden dotaz, nemá smysl si cokoliv předpočítávat a odpovíme jednou v čase $\mathcal{O}(n)$.
- Pokud bude dotazů řádově n, má smysl použít první předvýpočet. Pak budeme mít čas na předvýpočet i na samotný výpočet $\mathcal{O}(n \log n)$, což je optimum.
- Naopak pokud by dotazů bylo řádově n^2 nebo víc, tak se nám již první předvýpočet nevyplatí, dostali bychom se totiž na čas $\mathcal{O}(n^2 \log n)$. Zde se hodí použít druhý, delší předvýpočet a pak se dostat na časovou složitost $\mathcal{O}(n^2 + n^2 \cdot 1) = \mathcal{O}(n^2)$.

Hladové algoritmy

Věřte nebo ne, ale i počítač se někdy cítí hladový. Po namáhavé práci mu můžeme dopřát to potěšení, aby si ukousl co největší kus dat. A ukážeme, že někdy je to i ku prospěchu. Řeč bude o *hladových algoritmech*.

Takovými algoritmy rozumíme ty, které hledají řešení celé úlohy po jednotlivých krocích a splňují následující dvě podmínky:

- V každém kroku zvolí lokálně nejlepší řešení.
- Provedené rozhodnutí již nikdy neodvolává (tedy nebacktrackuje).

Lokálně nejlepší řešení je takové, které v aktuálním kroku vybere tu možnost, která nám na tomto místě nejvíce pomůže (bez jakéhokoliv ohledu na globální stav). Může to být třeba nejvyšší hodnota, nebo nejkratší cesta v grafu.

Pokud ale od hladového algoritmu chceme, aby nám našel globálně nejlepší řešení, musí naše úloha splnit předpoklad, že si výběrem lokálně nejlepšího řešení nezhoršíme to globální. Tento předpoklad se nedá formulovat obecně a je nutné se nad ním zamyslet zvlášť u každé úlohy.

Příklady hladových algoritmů

První hladovou úlohou bude (jak jinak) automat na jídlo vracející mince. Automat by měl vracet peníze nazpět tak, aby vrátil daný obnos v co možná nejmenším počtu mincí. Pro náš měnový systém (máme mince hodnot 1, 2, 5, 10, 20 a 50 Kč) lze tuto úlohu řešit hladovým algoritmem – v každém kroku algoritmu vrátíme tu největší minci, kterou můžeme (tedy pro vrácení 42 Kč to bude $42 = 20 + 20 + 2$ Kč).

Měnové systémy většiny států jsou postavené tak, aby fungovaly takto pěkně, neplatí to ale obecně. Zkusme si vrátit 42 Kč, pokud bychom měli jen mince hodnoty 20, 10 a 4 Kč. Správným řešením je $42 = 20 + 10 + 4 + 4 + 4$ Kč, hladový algoritmus by ale zkusil vrátit $42 = 20 + 20 + \ldots$ a tady by selhal.

Dále se velmi často dají hladovým algoritmem řešit nějaké úlohy přidávání nebo odebírání skupin prvků. Typickým příkladem je třeba rozvržení naplánovaných přednášek do učeben. Seřadíme si začátky přednášek podle času a postupně bereme jednu za druhou a umísťujeme je do volných učeben s nejnižším číslem.

Tím jsme si určitě nic nerozbili, protože v nějaké učebně přednáška být musí. Určitě budeme potřebovat tolik učeben, kolik je maximálně přednášek v jeden čas, a díky tomu si umístěním přednášky do nějaké učebny nezablokujeme místo pro jinou přednášku, jelikož nám vždy zbude dostatek volných učeben.

Kdybychom ale naopak měli pevně zadaný počet učeben a chtěli jsme do nich umístit co možná nejvíce přednášek, tak se již nejedná o úlohu řešitelnou hladovým algoritmem, v takovém případě je potřeba zvolit nějaký chytřejší postup.

Závěr

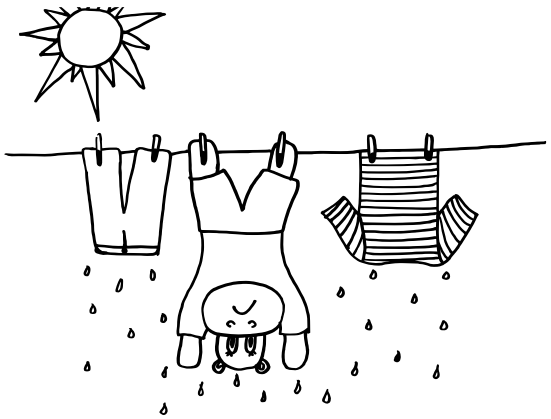
Doufám, že jste si z tohoto rozsáhlého textu odnesli nějaké nové znalosti a poznatky, které vám pomohou nejen v řešení KSP.

Pokud jste začínajícími řešiteli, zkuste s pomocí kuchařky vyřešit několik lehčích úloh a jejich řešení poslat – nově nabyté znalosti je totiž nejlepší co nejdříve protrénovat. Nic si nedělejte z toho, pokud napoprvé nevyřešíte všechno, s postupným zkoušením se budou vaše znalosti jen zlepšovat. Zkušenější řešitelé možná v kuchařce nalezli nějaké ujasnění pojmů, či si některé techniky osvěžili.

A pokud tento text považujete za dobrý, budeme jen rádi, pokud ho doporučíte svým kamarádům a spolužákům, kteří chtějí s programováním začít.

Úvodním kurzem vaření podle kuchařky vás provedl

Jirka Setnička



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

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: `0E:D9:B6:E5:6F:B0:51:D9:66:EB:E9:29:E4:58:AB:5F:99:D6:FD:A3`.