Korespondenční Seminář z Programování

36. ročník **KSP** **Září 2023**

Milí řešitelé, řešitelky a řešitelčata!

Dostává se k vám první číslo hlavní kategorie 36. ročníku KSP.

Letos se můžete těšit v každé z pěti sérií hlavní kategorie na **4 normální úlohy**, z toho alespoň jednu praktickou opendatovou. Dále na **kuchařky** obsahující nějaká zajímavá informatická témata, hodící se k úlohám dané série. Občas se nám také objeví **bonusová X-ková úloha**, za kterou lze získat X-kové body. Kromě toho bude součástí sérií **seriál**, jehož díly mohou vycházet samostatně.

Autorská řešení úloh budeme vystavovat hned po skončení série. Pokud nás pak při opravování napadnou nějaké komentáře k řešením od vás, zveřejníme je dodatečně.

Odměny & na Matfyz bez přijímaček

Za úspěšné řešení KSP můžete být **přijati na MFF UK bez přijímacích zkoušek.** Úspěšným řešitelem se stává ten, kdo získá za celý ročník (hlavní kategorie) alespoň 50 % bodů. Za letošní rok půjde získat maximálně 300 bodů, takže hranice pro úspěšné řešitele je 150. Maturanti pozor, pokud chcete prominutí využít letos, musíte to stihnout do konce čtvrté série, pátá už bude moc pozdě. Také každému řešiteli, který v tomto ročníku z každé série dostane alespoň 5 bodů, darujeme KSP propisku, blok, nálepku na notebook a možná i další překvapení.

Termín série: **neděle 5. listopadu 2023 ve 32:00 (tedy další ráno v 8:00)**

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

Značky úloh: Lehčí úloha (či její část) vhodná pro začátečníky; Praktická open-data úloha; Úloha, u které doporučujeme začíst se do kuchařky; Seriálová úloha

Odměna série: **Sladkou odměnu** si vysloužíte, kdo **odešlete originální řešení úlohy Nejlepší programovací jazyk**.

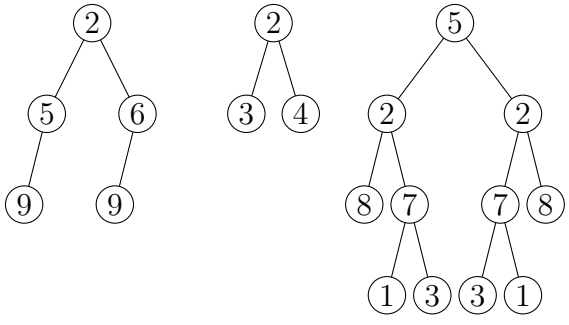
První série třicátého šestého ročníku KSP

36-1-1 Strom v zrcadle — 9 bodů

„Zrcadlo, zrcadlo, řekni mi, kdo je na světě nejsymetričtější?“ nakrucoval se pěkně košatý král binárních stromů před zrcadlem ve své lesní říši. Všemi svými listy i vnitřními vrcholy věřil, že zrcadlo samozřejmě označí jeho jako největšího a zajisté i nejsymetričtějšího vládce stromů.

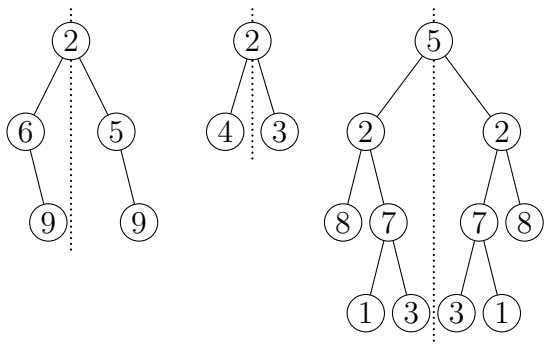
Problém byl v tom, že zrcadlo bylo pravdomluvné… a vážně si nebylo jisté ani v tom, jestli je král strom vůbec symetrický. Pomůžete zrcadlu?

V zrcadle se odráží obraz *zakořeněného binárního stromu* a v každém jeho vrcholu se nachází nějaká hodnota. Stromy mohou vypadat třeba jako na následujícím obrázku.



Můžeme si představit, že máme strom uložený v paměti, a to tak, že v každém vrcholu máme *ukazatele* na levého syna, pravého syna a na otce. Každý z těchto ukazatelů může být i prázdný, pokud vrchol daného syna nemá (prázdný ukazatel na otce pak bude jen v *kořeni* stromu). Pokud některým pojmům o stromech nerozumíte, můžete nahlédnout do přiložené kuchařky základních algoritmů.

Pro takto zadaný strom by nás zajímalo, jestli můžeme strom ozrcadlit a dostat tak stejný strom – strom se stejným tvarem i stejnými hodnotami ve vrcholech. Ozrcadlení znamená, že pro každý vrchol otočíme pořadí synů (prohodíme levého syna za pravého syna, včetně všeho navázaného pod nimi). Když ozrcadlíme stromy z prvního obrázku, dostaneme následující stromy.



Vidíme, že první strom se po ozrcadlení liší tvarem i hodnotami ve vrcholech, druhý má sice stejný tvar, ale liší se hodnotami ve vrcholech a jen třetí lze ozrcadlit a získat tak stejně vypadající strom. Jen o třetím stromu tedy můžeme prohlásit, že je symetrický.

Zrcadlo bohužel nemá moc velkou výpočetní paměť, umí si pamatovat jen konstantně mnoho údajů. Pozor, zásobník rekurze se také počítá do paměti, takže si určitě nemůžeme dovolit rekurzivní funkci na procházení stromu.

Vymyslete a popište algoritmus, který za těchto paměťových omezení bude umět pro již načtený strom v paměti v co nejrychlejším čase rozhodnout, zdali je symetrický. Porovnání dvou hodnot ve vrcholech zvládneme v čase $\mathcal{O}(1)$.

Toto je teoretická úloha. Není nutné ji programovat, odevzdává se pouze slovní popis algoritmu. Více informací zde: `http://ksp.mff.cuni.cz/viz/tinfo`

36-1-2 Taneční — 11 bodů

Král pořádá veliký ples, na kterém se sešlo mnoho úžasných tanečníků a tanečnic. Právě probíhá dámská volenka – dámy i pánové stojí naproti sobě ve dvou stejně početných řadách: N dam a N pánů. Pro jednoduchost jsou obě řady očíslované popořadě čísly 1 až N.

Dámy se už před začátkem dámské volenky dohodly na tom, kterého z pánů si každá dáma vybere pro následující tanec – každá si vybrala jiného, takže všechny dámy i všichni pánové jsou pro tento tanec zadaní.

Jak začne hrát hudba, tak se dámy mohou vydat ke svým vybraným partnerům. Problém je ale v tom, že se dráhy dam mohou křížit, například když si první dáma vybere druhého pána a druhá dáma toho prvního. A to by mohlo způsobit faux-pas.

Některé dámy tak budou muset chvíli počkat. Rády by ale věděly, kolik nejvýše dam bude moci vyrazit hned, jak začne hrát hudba, aby se dráhy žádných dvou z nich nekřížily. Pomůžete jim?

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

Formát vstupu: Na prvním řádku vstupu bude číslo N udávající počet dam a počet pánů. Na dalších N řádcích pak bude vždy jedno číslo od 1 do N udávající, kterého pána si vybrala i-tá dáma.

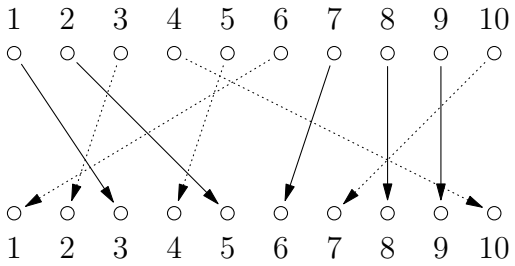
Formát výstupu: Na první řádek výstupu vypište jedno číslo K udávající počet dam, které mohou vyrazit bez toho, aniž by se jejich dráhy křížily. Na dalších K řádcích pak uveďte čísla těchto dam. Pokud existuje více způsobů, jak může vyrazit K dam najednou, vyberte libovolný z nich.

Ukázkový vstup:

```
10
3
5
2
10
4
1
6
8
9
7
```

Ukázkový výstup:

```
5
1
2
7
8
9
```



V příkladu výše by ještě stejné dobré řešení bylo vybrat dámy 1, 5, 7, 8 a 9 nebo 3, 5, 7, 8 a 9, všechny ostatní možnosti nedovolí vybrat více než čtyři dámy najednou.

36-1-3 Výšlap — 12 bodů

Kristýna se vydala na týdenní turistiku do Jeseníků. Jeseníky si můžeme představit jako 2D mřížku $R \times S$ políček, kde na každém políčku je výška terénu.

A jak probíhá Kristýnin den? Nejprve se probudí a vyleze na nějaký kopec, kde si dá oběd (ten má již uvařený a nese si ho s sebou). Po obědě sleze na vyhlídnuté místo, kde si rozloží spacák a bude před spaním sledovat hvězdy.

Dneska má k obědu borůvkové knedlíky, což je do žaludku těžké jídlo, a proto se rozhodla naplánovat dnešní výšlap následujícím způsobem:

- Ráno se probudí na místě, kde včera večer ulehla.
- Poté, plná energie, vyleze na kopec. Bude se opakovaně pohybovat na sousední políčka (políčka v Jeseníkách sousedí stranou, roh nestačí) a vždy se musí pohnout na vyšší, než to na kterém stojí.
- V nejvyšším bodě výšlapu si dá borůvkové knedlíky.
- Poté, plná knedlíků, půjde rovnou dolů. S knedlíky se špatně chodí do kopce, a proto každé další pole musí být nižší než předchozí.

Kristýna by chtěla poznat co nejvíce krás Jeseníků, a proto by její výšlap měl být co nejdelší (co do počtu polí). Políčko je možné navštívit během výšlapu cestou nahoru i dolů, pak se do délky započítává každé jeho navštívení. Pokud je nejdelších výšlapů více, pak vyberte ten s nejvyšší navštívenou výškou (dá se tam fotit pěkné panorama).

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

Formát vstupu: Na prvním řádku dostanete dvě čísla R a S – počet řádků a sloupců mapy Jeseníků. Na druhém řádku dostanete čtyři čísla S_x, S_y, C_x, C_y – souřadnice startu a cíle výšlapu (souřadnice x udává sloupec, souřadnice y řádek; indexujeme od nuly s tím, že políčko v levém horním rohu má souřadnice $(0, 0)$). Následuje R řádků, kde na každém z nich je S čísel popisujících Jeseníky – číslo udává výšku terénu na daném políčku.

Formát výstupu: Na první řádek vypište délku nejdelšího výšlapu L. Potom vypište L řádků, na každém X_i, Y_i – souřadnice i-tého políčka výšlapu. První políčko musí být S_x, S_y a poslední C_x, C_y. Pokud je nejdelších výšlapů více, pak vypište ten s nejvyšší navštívenou výškou. Pokud je stále výšlapů více, vypište libovolný z nich. Zaručujeme, že nějaký výšlap vždy existuje.

Ukázkový vstup:

```
4 3
1 0 1 3
1 1 1
1 6 1
9 7 1
1 1 1
```

Ukázkový výstup:

```
6
1 0
1 1
1 2
0 2
1 2
1 3
```

36-1-4 Mediánový filtr — 12 bodů

Byl pozdní večer a astronom pracující s výstupy z JWST[1] si po pátém pádu jeho oblíbeného grafického programu uvědomil, že složený obrázek z teleskopu o rozlišení dosahujícím gigapixelové velikosti asi takhle jednoduše neodšumí. Posadil se tedy ke klávesnici a začal přemýšlet o tom, jak napsat vlastní program.

Má čtvercový obrázek o velikosti $N \times N$ pixelů a rozhodl se použít *mediánový filtr* o velikosti $K \times K$ na odstranění impulzního šumu. Uvažujme pro jednoduchost, že obrázek je jen ve stupních šedi (v rozsahu od nuly do nějakého vysokého čísla) a K bude vždy liché (aby čtverec $K \times K$ měl vždy jasně určený střed).

Mediánový filtr funguje tak, že pro každý pixel obrázku umístí do tohoto pixelu střed čtverce a z až K^2 hodnot ve čtverci (méně hodnot tam může být například na okrajích obrázku) vybere tu prostřední, neboli *medián*, a tu zapíše do výsledného obrázku. Tím dojde třeba k odstranění náhodně „přepálených“ pixelů.

Počítejte s tím, že $1 \ll K \ll N$ (K je řádově větší než malá konstanta a zároveň řádově menší než N). Počítat mediánový filtr pro každý pixel obrázku tak nedává moc smysl, protože by výpočet běžel moc dlouho. Tato úloha cílí na řešení v čase určitě lepším než $\Theta(N^2K^2)$.

3	3	4	5	1	1	2	1	1	2
4	4	5	6	2	2	2	1	1	3
5	5	0	6	5	4	4	2	2	4
6	6	1	2	(8)	3	3	2	2	2
7	7	2	2	2	2	5	5	3	6
7	7	6	6	5	5	9	2	7	6
6	6	5	5	4	3	8	7	6	6

V příkladu výše vidíte mediánový filtr pro $K = 5$, který právě počítá, co bude v novém obrázku na pozici zakroužkovaného pixelu. Má okénko s 25 hodnotami, které když si seřadíme, tak to jsou: 0, 1, 2, 2, 2, 2, 2, 2, 2, 2, 3, 3, 4, 4, 5, 5, 5, 5, 5, 6, 6, 6, 6, 8, 9. Medián (a tedy výsledná hodnota) je číslo 4.

Toto je teoretická úloha. Není nutné ji programovat, odevzdává se pouze slovní popis algoritmu. Více informací zde: `http://ksp.mff.cuni.cz/viz/tinfo`

36-1-5 Nejlepší programovací jazyk, část I. — 1 bod

Jak bylo vyzkoumáno už dávno,[2] dobrý programovací jazyk lze vytvořit přidáním samých dobrých instrukcí. Pojďme tedy spolu uvařit ten nejlepší programovací jazyk.

Upozornění: Toto **není** kuchařková úloha.

Náš programovací jazyk počítá s celými čísly (jak je už z názvu zřejmé, jsou lepší než čísla necelá). Jediná paměť, kterou má k dispozici, je *zásobník*. Ten si můžete představit jako posloupnost čísel uspořádanou shora dolů. Nová čísla přidáváme na vrchol zásobníku, odebíráme opět z vrcholu.

Celá čísla jsou 64-bitové znaménkové integery (tedy rozsah od -2^{63} až po $2^{63}-1$). Přetečení způsobí ukončení programu chybou. Zásobník má prozatím přesně nespecifikovaný maximální počet prvků (alespoň 100 000), při přeplnění dojde k ukončení programu chybou. Taktéž existuje prozatím blíže nespecifikovaný limit na čas výpočtu.

Program se skládá z konečné sekvence instrukcí zapsaných do souboru. Kromě zásobníku ještě máme instrukční ukazatel *IP*, což je číslo příští vykonávané instrukce. Začíná na 0, tedy na první instrukci, a když přesáhne číslo poslední instrukce, tak program skončí. Ale než se dostaneme k programování, musíme nejdřív sehnat ty nejlepší instrukce.

Úkolem každého z vás bude odevzdat popis jedné vámi vymyšlené instrukce a její očekávaný zápis. Pokud nespecifikujete jinak, instrukce přičte k *IP* jedničku. Také nám dejte vědět, jestli chcete svou instrukci zadat anonymně, nebo jestli se chcete podepsat – v takovém případě vás u ní pak uvedeme jako autora.

Příklad: instrukce `+` sečte dvě čísla na vrcholu zásobníku a nahradí je jejich součtem.

Instrukce rozhodně nemusí být tak přímočaré jako sčítání, nebojte se popustit uzdu fantazii. Požadujeme pouze, aby bylo chování instrukce jednoznačné a šlo ji rozumně implementovat. Pokud si nejste jistí, napište nám. Instrukce může být netriviální, ale upozorňujeme, že výpočetně příliš náročné instrukce mohou zabrat příliš času vykonávání programu, který je omezený.

Pro inspiraci se můžete také podívat na úlohu 35-4-1 Golfový turnaj.[3]

Druhou část úlohy očekávejte v druhé sérii. A také prosím o této úloze **nediskutujte na Discordu ani jinde** (ať nezkazíte ostatním vymýšlení originálních instrukcí).

36-1-X1 Mezigalaktická kandidátka — 10 bodů

Toto je bonusová úloha pro zkušenější řešitele, těžší než ostatní úlohy v této sérii. Nezískáte za ni klasické body, nýbrž dobrý pocit, že jste zdolali něco výjimečného. Kromě toho za správné řešení dostanete speciální odměnu a body se vám započítají do samostatných výsledků KSP-X.

Galaktickou říši čekají další volby! Jednotlivé členské rasy mezigalaktické unie už předložily císařskému úřadu své seznamy kandidátů, ale císař se teď rozhodl, že v nadcházejících volbách chce, aby na kandidátních listinách jednotlivých ras byla zastoupena všechna pohlaví dané rasy rovnoměrně (což je triviální u ras s jedním pohlavím, ale některé rasy s desítkami různých pohlaví způsobily úředníkům císařského úřadu pořádné vrásky).

Aby to nebylo tak jednoduché, tak úředníci nesmějí nijak přehazovat pořadí navržených kandidátů, jen mohou ze začátku a z konce kandidátních listin nějaký počet kandidátů

[1] James Webb Space Telescope

[2] ČAPEK, Josef. Povídání o pejskovi a kočičce. 16. vyd., Praha: Albatros, 2003. 118 s. Strana 79.

[3] `http://ksp.mff.cuni.cz/viz/35-4-1`

vyškrtnout tak, aby zůstal co nejdelší úsek kandidátů, ve kterém jsou všechna pohlaví dané rasy zastoupena stejným počtem.

Vaším úkolem je tedy vymyslet algoritmus, který pro zadaný počet pohlaví K a pro posloupnost délky N sestávající se z pohlaví kandidátů 1 až K odstraní co nejméně kandidátů ze začátku a konce tak, aby úsek, který zbude, byl vyrovnaný.

Nápověda

Protože se nikomu nepodařilo tuto úlohu vyřešit, rozhodli jsme se, že prodloužíme její deadline do konce 2. série. Navíc vydáváme následující nápovědu, která vám snad pomůže k řešení.

Očekávaná časová složitost je

$$\mathcal{O}(\text{počet kandidátů} \cdot \text{něco logaritmického}).$$

Toho lze dosáhnout třeba metodou Rozděl a panuj s podproblémy o menším počtu pohlaví.

36-1-S Seznámení se s lineární regresí — 15 bodů

Letošním ročníkem vás bude provázet seriál. V každé sérii se objeví jeden díl, který bude obsahovat nějaké povídání a navíc úkoly. Za úkoly budete moci získávat body podobně jako za klasické úlohy série. Některé úkoly budou programovací, nicméně pokud nebude řečeno jinak, tak se nebudou vyhodnocovat přes odevzdávátko. Všechny soubory nahrávejte v jednom zazipovaném souboru. Všechny programovací úlohy prosíme řešte v programovacím jazyce Python 3. Abyste se mohli zapojit i během ročníku, seriál bude možné odevzdávat i po termínu za snížený počet bodů. Vzorové řešení seriálu proto před koncem celého ročníku KSP uvidí jen ti, kdo už seriál odevzdali.

V letošním seriálu se zaměříme na praktické téma, které využívá znalosti z pravděpodobnosti, matematické analýzy a lineární algebry. Tento rok se budeme věnovat strojovému učení (anglicky machine learningu). Buzzwordy jako machine learning, neural networks, ChatGPT či jen AI se v průběhu posledních let dostaly do popředí a spoustu lidí o tomto tématu mluví, i když většina o něm neví téměř nic.

V tomto seriálu se nestihneme dostat tak daleko, abychom si pověděli, jak funguje ChatGPT či jiní chatovací boti, ale ukážeme základní stavební bloky, od kterých se současné nejpokročilejší techniky v umělé inteligenci odvíjejí.

Než se dostaneme k samotnému strojovému učení, tak si nejdříve povíme, co si pod tímto pojmem vůbec představujeme. O nějakém programu (ve strojovém učení programům říkáme modely) můžeme říct, že se učí, pokud se s přibývajícím množstvím dat, kterými jej krmíme, zlepšuje v nějaké metrice. Je důležité poznamenat, že úspěšnost strojového učení měříme nikoliv na těch datech, kterými model krmíme, ale na dosud neviděných datech. Tím se strojové učení zásadně liší od databází či optimalizačních úloh jako třeba problém obchodního cestujícího.

Typy úloh

Úloh, které bychom strojovým učením chtěli řešit, je spoustu. Například odšumění obrázku, generování obrázku dle zadání či odpověď na položenou otázku. To jsou už celkem náročné úlohy, takže začneme s něčím jednodušším – regresí a klasifikací. Slovo *regrese* může znít učeně, ale prakticky to znamená, že budeme odhadovat nějaké číslo. Můžeme si představit, že jsme majitel obchodu s koloběžkami a chceme odhadnout kolik koloběžek půjčíme daný den, když víme, že to bude sobota v lednu a bude 10 °C přes den. Nebo manažer callcentra a chceme vypredikovat kolik lidí zavolá na naši linku.

V úlohách na *klasifikaci* předpovídáme skupinu. Například před sebou máme obrázek zvířete a chceme určit, o které zvíře z daného seznamu se jedná.

Výkonnost

Pro měření výkonnosti (či chybovosti) potřebujeme nějakou metriku (chybovou funkci), která bude říkat, jak moc dobré či špatné řešení jsme vyprodukovali. Jedna z takových metrik může být *průměr čtverců (anglicky mean squared error – MSE).* Tato metrika je jednoduchá: pro sadu vstupů (dat) jsme vyprodukovali predikce. Pro každý vstup pak vypočítáme chybovost pomocí formule $(p-t)^2$, kde p je naše predikovaná hodnota a t je ta správná hodnota, kterou chceme odpovědět. Tyto chyby poté zprůměrujeme a to je naše výsledná metrika MSE.

Když však chceme klasifikovat do tříd, typicky budeme chtít používat jinou metriku na měření výkonosti než průměr čtverců. Standardním způsobem, jak měřit správnost modelu při klasifikaci do dvou či více tříd, je *přesnost (accuracy).* Ta nám říká, jaký podíl dat jsme zařadili do té správné kategorie.

Dále existuje mnoho dalších metrik. Při trénování klasifikačních modelů se ještě potkáme s NLL *(negative log likelihood),* dále se čast používá *f1-skóre,* nebo *preciznost (precision)* či *senzitivita (recall),* které se mimo strojové učení používají například u lékařských testů, a mnoho dalších.

V dnešním dílu seriálu budeme ale používat jen metriku MSE.

Zkušenosti

V reálném životě se můžeme učit (dostávat zkušenosti) různými cestami. Když jsme byli malí, tak nás rodiče učili jména různých předmětů a zároveň nás opravovali, když jsme řekli nějakou blbost. Tomuto typu učení se říká *učení s učitelem (supervised learning).* Neboli máme sadu dat a ke každému *datu* známe správnou odpověď, kde jedno *dato je jeden příklad (vzorek)* ze vstupních dat.

Druhý typ učení je *zpětnovazební (reinforcement learning).* To známe také z reálného života – naučíme se na test z nějakého předmětu, napíšeme ho a to, zda jsme se naučili dostatečně, zjistíme ze známky. Neboli máme nějakého agenta v nějakém prostředí a ten může dělat nějaké akce a po jedné či více akcí dostaneme *zpětnou vazbu (feedback),* jak si agent vede.

Poslední typ učení je nejvíce zvláštní, říká se mu *učení bez učitele (unsupervised learning).* Tento typ učení funguje tak, že programu dáme data a ten má vydat nějaký výsledek. V praxi se tento typ učení používá, když máme data a chceme z nich něco „vykoukat“ (neboli analyzovat data). Tento přístup se používá např. na *clusterizaci* – máme data a program má vydat nějaké shluky dat, které jsou si podobné. Je očividné, že v závislosti na tom, jakou máme metriku na porovnávání dat, se výstupy můžou razantně lišit.

Další možné využití je např. *detekce anomálií.* Uvažte, že máte data síťové komunikace ze své sítě a chcete v reálném čase zjišťovat, jestli se ve vaší síti neděje něco nekalého.

Pro začátek začneme s učením s učitelem, ale později v seriálu si ukážeme i nějaký algoritmus, který používá učení

bez učitele.

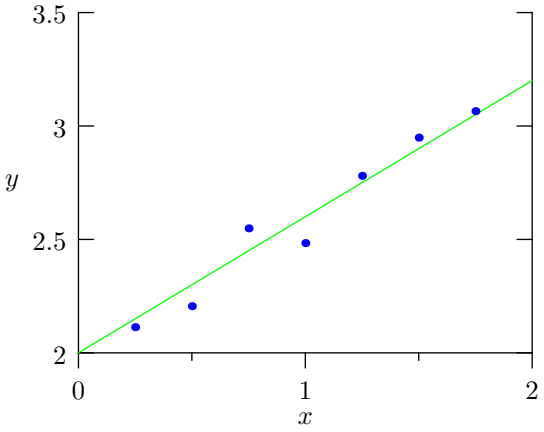
Lineární regrese

Řekli jsme si trochu obecných informací o strojovém učení a nyní si zkusme vyřešit první úlohu. Máme data, kde každé dato má jednu známou *hodnotu (featuru)*. Uvažte, že jste realitní makléř a jediné, co víte o domu, je celková obyvatelná plocha. Chcete znát orientační cenu domu, kterou vám nikdo neřekne. Každý přece chce nakoupit za levno a prodat za draho.

Máte pár dat, kde znáte celkovou obyvatelnou plochu a cenu nemovitosti v inzerátu. Tato data jsou zakreslena jako modré body. Na x-ové ose je celková obyvatelná plocha a na y-ové ose je cena nemovitosti. Cíl je vygenerovat pro každou x-ovou hodnotu predikci – y-ovou hodnotu.

Očekáváme, že body jsou lehce zašuměné, protože všechna měření v reálném světě jsou nějak zašuměná. Zde si to můžete představit tak, že cena v inzerátu není fixní a pokud je dlouhodobě jen jeden zájemce, může kupující snížit cenu. Naopak pokud je hodně zájemců, mohou se předhánět, kdo nabídne více.

Naším cílem je vyprodukovat zelenou čáru, která předpovídá y-ové hodnoty (cenu nemovistosti) pro libovolnou x-ovou hodnotu (celkovou obyvatelnou plochu).



Tento graf je specifický pro jednu hodnotu. Kdybychom měli dvě featury, pak výsledná predikce odpovídá nějaké rovině v trojrozměrném grafu. Tedy by nám někdo o domu řekl další informaci, např. kolik záchodů se v něm nachází. Obecně, predikce je nějaká nadrovina v n-rozměrném prostoru.

Náš první model, který se bude jmenovat *lineární regrese,* funguje následovně. Mějme vstupní data a každé dato bude mít f featur $x_1, x_2, \ldots, x_f$.

Když si vezmeme předchozí příklad s realitním makléřem, první featura bude celková obyvatelná plocha, další featury můžou být den v týdnu, kdy se nemovitost prodávala a poslední počet místností v domě. Dále každé dato bude mít i další hodnotu s odpovědí, kterou chceme vypredikovat. Asi nejjednodušší způsob predikování výsledných dat, který by nás mohl napadnout, je vynásobit vstupní hodnoty vhodnými konstantami a sečíst je. A přesně tohle dělá i lineární regrese.

$$\text{predikce} = x_1 \cdot w_1 + x_2 \cdot w_2 + \ldots + x_f \cdot w_f$$

kde $w_1, w_2, \ldots, w_f$ jsou ony konstanty (váhy), které se budeme snažit naučit, aby predikce byla co nejlepší. Tedy chceme váhy minimalizující naší chybovou funkci. Takto uvedená predikce má však jeden problém. Na dato, které má všechny hodnoty $x_1, x_2, \ldots, x_f$ nulové, vypredikujeme vždycky také nulu. Abychom vyřešili tento problém, budeme přičítat ještě jednu hodnotu b, kterému říkáme *bias.* S biasem vypadá predikce následovně:

$$\text{predikce} = x_1 \cdot w_1 + x_2 \cdot w_2 + \ldots + x_f \cdot w_f + b$$

Většinou se dělá malý trik: aby se bias nemusel explicitně zapisovat, tak se do vstupních dat přidá ještě jedna featura, která bude mít vždy hodnotu 1. Tím pádem se bias může zahrnout do vah a predikce vypadá skoro stejně jako výše (jen přibyla jedna váha):

$$\text{predikce} = x_1 \cdot w_1 + x_2 \cdot w_2 + \ldots + x_{f+1} \cdot w_{f+1},$$

kde $x_{f+1} = 1$ a $w_{f+1} = b$.

Když budeme znát hodnoty $w_1, w_2, \ldots, w_{f+1}$, bude již možné predikovat vysněné hodnoty, jak si můžete sami vyzkoušet v následující úloze.

Úkol 1 – Predikce lineární regrese [3b]: Naprogramujte v Pythonu predikci lineární regrese.

Na prvním řádku vstupu dostanete dvě čísla N a F, kde N je počet dat a F je počet featur. Na následujícím řádku naleznete $F+1$ desetinných čísel reprezentujících váhy modelu, kde poslední z nich je bias. Poté na dalších N řádcích dostanete na každém řádku F desetinných čísel, kde každý řádek reprezentuje jedno vstupní dato.

Vypište N řádků, kde na každém řádku bude jedno číslo (predikce pro dané vstupní dato).

Ukázkový vstup:	*Ukázkový výstup:*

`2 3`
`0.1 1.2 -0.5 3.0`
`17.0 1.0 0.4`
`27.0 2.0 0.8`
	`5.7`
	`7.7`

Nyní máme vypredikovaná data a nabízí se otázka – jak dobrá jsou?

Jak už víme z předchozí části, k tomu máme chybovou funkci – metriku, která nám říká, jak moc špatná data jsme vypredikovali. Tedy čím větší hodnota chybové funkce, tím hůře. Pro lineární regresi tou chybovou funkcí, kterou budeme používat, bude MSE.

Úkol 2 – Měření chyby [3b]: Napište Pythoní program, který pro zadané váhy a vstupní i výstupní data zjistí, jaká je hodnota chybové funkce MSE pro data vypredikovaná s danými váhami.

Formát vstupu je velmi podobný tomu předchozímu. Na prvním řádku vstupu dostanete dvě čísla N a F, kde N je počet dat a F je počet featur. Na následujícím řádku naleznete $F+1$ desetinných čísel reprezentující váhy modelu, kde poslední z nich je bias. Poté na dalších N řádcích dostanete na každém řádku $F+1$ desetinných čísel. Každý z řádku odpovídá jednomu datu – prvních F hodnot na řádku jsou hodnoty jednotlivých vstupních featur a $(F+1)$-ní hodnota odpovídá správné výstupní hodnotě pro dané data. Tedy oproti předchozímu případu dostáváme na každém řádku ještě správnou vstupní hodnotu.

Na výstup vypište jediné číslo – hodnotu MSE.

Doporučujeme při implementaci využívat kusů kódu z předchozího úkolu.

Ukázkový vstup:

```
2 3
0.1 1.2 -0.5 3.0
17.0 1.0 0.4 4.7
27.0 2.0 0.8 8.2
```

Ukázkový výstup:

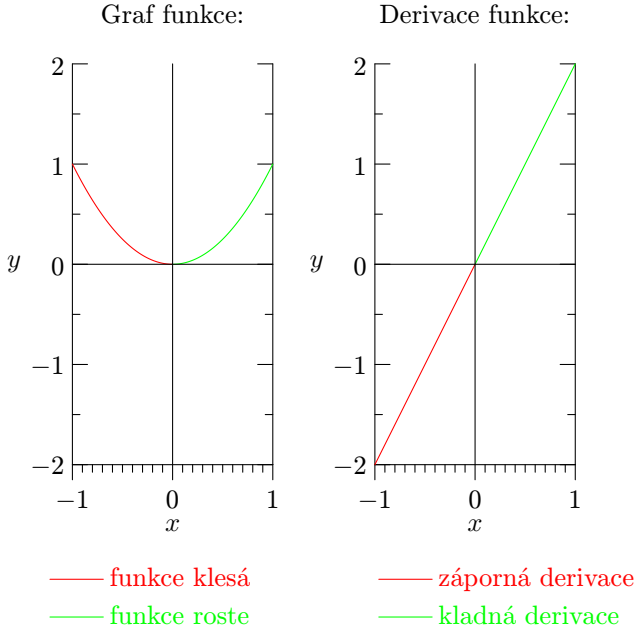
```
0.625
```

Derivace jako blackbox

V běžném světě ale typicky váhy lineární regrese neznáme a potřebujeme je nejprve spočítat. K tomu ale nejdříve potřebujeme znát jeden pojem z matematické analýzy, totiž derivaci.

Derivace je operace na funkcích a říká nám, jak se změní hodnota funkce, když velmi málo změníme jednu vstupní hodnotu (parametr). Je-li hodnota derivace v nějakém bodě kladná, pak původní funkce v daném bodě roste a roste tím strměji, čím větší je hodnota derivace. Je-li naopak záporná, pak v daném bodě klesá. A pokud je derivace nulová, pak funkce na nějakém velmi malém okolí bodu téměř nemění svoji hodnotu.

Vezměme kupříkladu funkci $y = x^2$ a zadívejme se na její derivaci podle x. Její derivace je rovna $2x$ (což zatím nevíme, kde se vzalo). Pro kladná x je hodnota derivace kladná, což odpovídá tomu, že kvadratická funkce na kladných hodnotách roste. Dokonce čím větší vezmeme x, tím větší je hodnota derivace, což zase odpovídá tomu, že kvadratická funkce je s většími x čím dál strmější. Obdobně pro záporná čísla je derivace záporná, jelikož zde hodnota funkce klesá. A pro $x = 0$ vychází derivace nulová, což zase odpovídá tomu, že kolem nuly kvadratická funkce svoji hodnotu nemění. Toto všechno ilustruje následující obrázek.



Derivujeme-li podle nějaké proměnné, ostatní parametry se vůči derivaci tváří jako konstanty (správně se těmto derivacím říká *parciální derivace* a vždy, když budeme zmiňovat derivace, budeme mít na mysli právě ty parciální).

Teď si můžete říct, k čemu nám to je? Když budeme mít nějakou funkci, kterou chceme minimalizovat, prostě vypočítáme derivaci a budeme hledat body, kde je derivace nulová. Proč nulová? Protože když je derivace kladná, tak zvětšení parametru zvětší i funkční hodnotu – tudíž zde určitě není *lokální extrém* (minimum ani maximum). To, že se jedná o minimum (nebo maximum) lokální, znamená, že se jedná o bod, který má nejmenší (či největší) hodnotu ze všech bodů v nějakém svém velmi blízkém okolí. Podobně to bude pro zápornou hodnotu derivace.

Když je hodnota derivace nulová, pak v tomto bodě může být lokální extrém, ale nemusí. Např. můžeme mít funkci, která je klesající až na nějaký úsek, v němž je konstantní. V konstantním úseku je derivace nulová, ale není zde nutně lokální extrém.

K lokálnímu extrému se váže ještě termín – *globální minimum či maximum.* Globální minimum znamená, že funkce z celého definičního oboru nabývá v daném bodě nejmenší hodnoty.

Globálního minima nemusí funkce nabývat jen v jednom bodě, např. funkce sinus má globální minimum hodnotu -1 a nabývá této hodnoty v nekonečně mnoha bodech. Nebo konstantní funkce $y = 2$: ta nabývá globálního minima na celém definičním oboru. Obdobně definujeme i globální maximum. Každé globální minimum (či maximum) je i minimem lokálním, ale naopak to neplatí – můžeme mít lokální extrém, který ale vůbec není celkově nejmenší hodnotou.

Máme-li nějaký bod s nulovou derivací, existují analytické způsoby, jak ověřit, zda se jedná o lokální minimum či nikoliv. Obzvláště ve více rozměrech však tyto metody začínají být poměrně komplikované. Co hůře, analyticky už nedokážeme poznat, zda se jedná o minimum globální. To může znít jako zásadní problém, ale prozatím se spokojíme s tím, že budeme používat samé pěkně vychované chybové funkce, o kterých lze ukázat, že mají právě jeden bod s nulovou derivací, a tím je globální minimum.

Ve skutečnosti je situace ještě o maličko zamotanější. Existují funkce, které nemají derivaci v některých bodech definovanou. A právě v těchto bodech se mohou také nacházet extrémy. S takovými funkcemi se nicméně v seriálu nesetkáme, takže pro nás bude platit, že kde je lokální či globální extrém, tam je derivace nulová.

Teď nám zbývá jen vysvětlit, jak se derivace počítá. Derivace funkce $f(x)$ se zapisuje jako $f'(x)$. U funkcí s více parametry se zapisuje podle jakého parametru derivujeme: $\frac{\partial}{\partial x_1} f(x_1, x_2, \ldots, x_n)$. My budeme používat tuto notaci, protože budeme explicitně psát, podle čeho derivujeme. Derivace samotná se dá počítat pomocí několika jednoduchých pravidel. Většina těchto pravidel má nějaké pěkné odvození, ale to je již mimo rozsah tohoto textu. Pro nás budou důležitá následující:

- derivace konstanty: $\frac{\partial}{\partial x} c = 0$, kde c je konstanta (nebo výraz ve kterém se nevyskytuje proměnná x)
- derivace mocniny: $\frac{\partial}{\partial x} x^n = n \cdot x^{n-1}$
- derivace součtu je součet derivací: $\frac{\partial}{\partial x}(f(x) + g(x)) = \frac{\partial f}{\partial x} + \frac{\partial g}{\partial x}$
- derivace složené funkce: $\frac{\partial}{\partial x} f(g(x)) = \frac{\partial f}{\partial g} \cdot \frac{\partial g}{\partial x}$. Když budeme derivovat funkci jedné proměnné, tak se zápis zjednoduší na $\frac{\partial}{\partial x} f(g(x)) = f'(g(x)) \cdot g'(x)$, ale význam je stejný. Zderivujeme vnější funkci a vynásobíme ji derivací vnitřní funkce. Například pro funkci $f(x) = (x - t)^2$ je derivace vnější funkce $f'(y) = 2 \cdot y$ a derivace vnitřní funkce $y = x - t$ je 1. Tedy $f'(x) = (2 \cdot (x - t)) \cdot 1$.

Pravidel je více, ale s těmito si vystačíme.

Optimální váhy v lineární regresi

Vraťme se k lineární regresi. Už umíme se zadanými váhami pomocí lineární regrese vytvářet predikce a už i víme, jakým způsobem můžeme pomocí MSE zjišťovat, zda váhy jsou pro nějaká data lepší nebo horší než jiné.

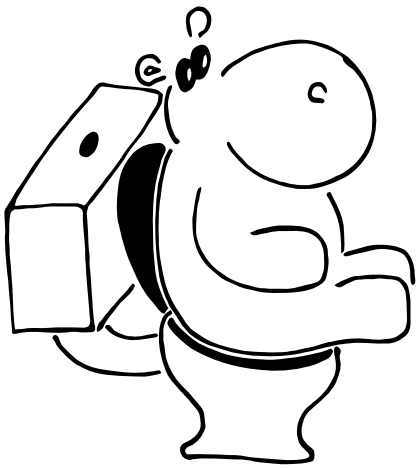
Co ale zatím vůbec neumíme, je pořídit si nějaké dobré váhy nebo ještě lépe ty optimální. Ba co hůř, my ani nedokážeme u zadaných vah poznat, zda jsou optimální, či nikoliv. Na to, jak hledat optimální váhy, si sice ještě budeme muset počkat do příštího dílu, ale nyní se alespoň naučíme, jak vlastně zjistit, jestli váhy optimální jsou.

Co pro takové optimální váhy musí platit? Musí to být váhy takové, že hodnota chybové funkce je nejmenší ze všech možných vah. Jinými slovy se jedná o váhy, které se nacházejí v globálním minimu. Aby se jednalo o globální (a tedy i lokální) minimum, musí být derivace podle všech proměnných rovna nule (tedy se nám nesmí stát, že by hodnota ještě někam klesala).

Sice obecně neplatí, že by nulová derivace znamenala globální minimum, ale zrovna v případě lineární regrese a MSE lze ukázat, že jediný bod s nulovou derivací vůči všem vahám je právě globální minimum.

Zbývá nám tak jediné. Zderivovat chybovou funkci podle vah a ověřit, že pro zadané hodnoty vah je vždy nulová.

Na rozdíl od predikce již nemůžeme postupovat dato po datu, protože optimalita závisí na všech datech, která máme. Speciálně již nebudeme mít jen jednu predikci p, ale několik různých predikcí $p_1, p_2, \ldots, p_n$ pro jednotlivá data, stejně tak budeme mít n správných odpovědí $t_1, t_2, \ldots t_n$. Pro každé dato budeme mít opět f vstupních featur a značení $x_{i,j}$ nám bude říkat hodnotu j-té featury pro i-té testovací dato.



Nyní se již můžeme vrhnout na derivaci chybové funkce. Tato část je o něco techničtější, proto se nebojte přeskočit odvozování, pokud by vám nepřišlo srozumitelné.

Jako chybovou funkci máme MSE a chceme najít derivace podle vah $w_1, w_2, \ldots, w_{f+1}$:

$$\begin{aligned}\frac{\partial}{\partial w_1}\text{MSE} &= \frac{\partial}{\partial w_1}\frac{1}{N}\left((p_1 - t_1)^2 + \cdots + (p_n - t_n)^2\right)\\ \frac{\partial}{\partial w_2}\text{MSE} &= \frac{\partial}{\partial w_2}\frac{1}{N}\left((p_1 - t_1)^2 + \cdots + (p_n - t_n)^2\right)\\ &\vdots\end{aligned}$$

Jelikož víme, že derivace součtu je součet derivací, tak si zkusíme zkusíme zderivovat chybovou funkci pro první predikci p_1. Predikce se počítá pomocí formule $p_1 = x_{1,1} \cdot w_1 + x_{1,2} \cdot w_2 + \cdots + x_{1,f+1} \cdot w_{f+1}$. Derivace chyby predikce p_1 podle jednotlivých vah vyjde následovně:

$$\begin{aligned}\frac{\partial}{\partial w_1}(p_1 - t_1)^2 &= 2 \cdot (p_1 - t_1) \cdot x_{1,1}\\ \frac{\partial}{\partial w_2}(p_1 - t_1)^2 &= 2 \cdot (p_1 - t_1) \cdot x_{1,2}\\ &\vdots\end{aligned}$$

Všimněte si, že derivace podle vah $w_1, w_2, \ldots, w_{f+1}$ jsou dost podobné, prakticky se mění jen to, jakou hodnotou $x_{1,i}$ vynásobíme, a tyto derivace vyjdou stejně pro všechny predikce. Tedy můžeme napsat derivaci chyby pro všechny predikce jako:

$$\begin{aligned}\frac{\partial}{\partial w_1}\text{MSE} &= \frac{1}{N}\left(2(p_1 - t_1) \cdot x_{1,1} + \cdots + 2(p_n - t_n) \cdot x_{n,1}\right)\\ \frac{\partial}{\partial w_2}\text{MSE} &= \frac{1}{N}\left(2(p_1 - t_1) \cdot x_{1,2} + \cdots + 2(p_n - t_n) \cdot x_{n,2}\right)\\ &\vdots\end{aligned}$$

Zpět k původní úvaze...

Chceme, aby se výsledné derivace rovnaly nule. Po pár dalších úpravách se dostaneme k následujícím rovnicím:

$$\begin{aligned}p_1 \cdot x_{1,1} + \ldots + p_n \cdot x_{n,1} &= t_1 \cdot x_{1,1} + \ldots + t_n \cdot x_{n,1}\\ p_1 \cdot x_{1,2} + \ldots + p_n \cdot x_{n,2} &= t_1 \cdot x_{1,2} + \ldots + t_n \cdot x_{n,2}\\ &\vdots\end{aligned}$$

Úkol 3 – Optimalita vah lineární regrese [4b]: Napište v Pythonu program, který na vstupu dostane data a váhy lineární regrese a rozhodne, jestli dané váhy jsou optimální vůči MSE metrice.

Formát vstupu je stejný jako v druhém úkolu. Na prvním řádku vstupu dostanete dvě čísla N a F, kde N je počet dat a F je počet featur. Na následujícím řádku naleznete $F+1$ desetinných čísel reprezentující váhy modelu a poslední váha reprezentuje bias. Poté na dalších N řádcích dostanete na každém řádku $F+1$ desetinných čísel. Každý z řádku odpovídá jednomu datu – prvních F hodnot na řádku jsou hodnoty jednotlivých vstupních featur a $(F+1)$-ní hodnota odpovídá správné výstupní hodnotě pro dané dato. Rozhodněte, jestli váhy jsou na daných datech optimální vůči MSE metrice. Váhy považujeme za optimální, pokud hodnota derivace vůči žádné z vah není v absolutní hodnotě větší než 10^{-6}. Pokud jsou váhy optimální, tak vypište `ANO`, jinak vypište `NE`.

Ukázkový vstup:	*Ukázkový výstup:*

`3 1`
`1.0 4.33333333`
`1.0 5.0`
`5.0 9.0`
`3.0 8.0`
	`ANO`

V typické implementaci derivací chybové funkce a obecně všeho, co jsme doposud dělali, se derivace a predikce nepočítají pro jednotlivá data a jednotlivé váhy odděleně, ale všechny najednou pomocí vektorů, matic a maticových násobení. To proto, že tyto operace běží velmi rychle na grafických kartách.

V seriálu matice potřebovat nebudeme, aby nebyla potřeba žádná předchozí znalost lineární algebry, ale je dobré vědět, že typicky by všechny předchozí operace byly implementovány pomocí matic.

Matice mají ještě jednu výhodu, a to tu, že se pomocí nich dají některé operace snadno zapisovat. Např. se jimi dá elegantně vyjádřit explicitní vzorec pro nalezení optimálních vah. Pokud máte odvahu, můžete si tento vzorec zkusit v následující bonusové úloze najít; pokud ne, tak si v příštím dílu ukážeme mnohem používanější a zároveň jednodušší způsob, jak optimální váhy hledat.

Pokud jste se s lineární algebrou ještě nesetkali, ale i tak by vás to zajímalo, můžete se podívat na náš seriál z minulého roku, který se lineární algebře věnuje.

Úkol 4 – Těžký bonusový úkol: Explicitní výpočet vah [0b]: Najděte maticový zápis toho, jak vypočítat všechny optimální váhy najednou. Tento úkol je zcela dobrovolný a za jeho vyřešení nedostanete žádné body.

Scikit-learn

Naprogramovat si model není nikdy na škodu, protože si většinou ujasníte, jak model funguje, a občas zjistíte, že jste si něco neuvědomili během pouhého čtení. Když už ale chcete trénovat opravdový model, tak je lepší sáhnout po nějaké knihovně, kde je daný model již implementovaný, protože knihovny jsou optimalizované na rychlost, řeší problémy s desetinnými čísly, zaokrouhlováním a spoustu dalších věcí.

V tomto seriále budeme používat jazyk Python a knihovnu scikit-learn. Knihovny na machine learning existují i pro jiné jazyky, ale v machine learningu je Python nejrozšířenější. Není se čemu divit, protože Python se používá hlavně jako rozhraní, kde zapíšete, jak má model vypadat, ale už samotný tréning modelu již neběží v Pythonu, jinak by to bylo pomalé.

Pro instalaci knihovny stačí napsat do příkazové řádky `pip install scikit-learn`. Pokud nevíte jak na to, tak se podívejte na tento návod.[4] Pokud budete mít problém s instalací, tak se nebojte zeptat na Discordu nebo přes email.

Nyní pojďme prozkoumávat knihovnu scikit-learn! Jedna z prvních věcí, které se hodí, je načíst si nějaká data. Scikit-learn má již nějaké *datasety (kolekce/sady dat)* předpřipravené, takže si je můžeme jednoduše načíst. Například dataset diabetes, který obsahuje data o pacientech s cukrovkou. V tomto datasetu máme 442 dat pacientů a každé dato má 10 hodnot (feature). Pojďme si vypsat prvních 5 dat:

```
import sklearn.datasets

# načtení datasetu
diabetes = sklearn.datasets.load_diabetes()

print(diabetes.data[:5])
print(diabetes.target[:5])
```

Funkce `load_diabetes` vrací objekt, který se chová jako slovník a obsahuje data pacientů, cílové hodnoty pro predikci a pár dalších atributů. Data o jednotlivých pacientech dostaneme z atributu `data` a cílové hodnoty z atributu `target`. Samotná data nejsou uložená v pythoním poli, ale v numpy poli. Numpy je matematická knihovna, která se používá pro vědecké výpočty a najdeme tam implementované spoustu funkcí z lineární algebry (práce s maticemi, inverze matic, maticové rozklady, ...), ze statistiky, atd. Hlavní, co je důležité vědět, že u normálního pole v Pythonu, kde když uděláme `[:5]` (slicing), tak se vytvoří nové pole. Numpy je optimalizované na rychlost, takže se nové pole nevytvoří, ale vytvoří se jen pohled na původní pole. Když změníme hodnotu v původním poli, změní se i v novém poli a naopak.

Tuto vlastnost si můžeme vyzkoušet na načteném datasetu:

```
dato = diabetes.data[0]
dato[0] = 1
print(diabetes.data[0])
```

Použití numpy se budeme kvůli jednoduchosti co nejvíce vyhýbat, ale scikit-learn nám přímo vrací numpy pole, případně z numpy něco příležitostně využijeme.

Nyní máme načtená data, tak pojďme vytvořit model lineární regrese. Pro vytvoření modelu potřebujeme naimportovat `sklearn.linear_model` a vytvořit model pomocí `sklearn.linear_model.LinearRegression()`. Konstruktor má pár parametrů, ale všechny vychozí hodnoty jsou pro nás vyhovující. Například parametr `fit_intercept` říká, jestli se bude trénovat bias. Když jsme psali výše o skrývání biasu do vstupních dat, tak toto je implementační detail daného modelu, tedy když budete předávat vstupní data do modelu z knihovny scikit-learn, tak není potřeba do dat přidávat jedničkovou featuru. Ale nic tím nezkazíte, když ji tam přidáte. V daném případě by bylo dobré vypnout `fit_intercept`, protože poté by se v modelu nacházely 2 biasy, což je zbytečné.

Poté model natrénujeme pomocí funkce `fit(data, target)`, kde `data` jsou vstupní data a `target` jsou cílové hod-

[4] `https://ksp.mff.cuni.cz/encyklopedie/python-pip.html`

noty. Nakonec máme natrénovaný model a můžeme predikovat pomocí funkce `predict(data)`, kde `data` jsou data, které chceme predikovat.

```
import sklearn.linear_model as lm
estimator = lm.LinearRegression()
# učíme se na všech datech kromě posledního
estimator.fit(diabetes.data[:-1],
              diabetes.target[:-1])
# predikujeme poslední dato
prediction = estimator.predict(diabetes.data[-1:])
print(f"Predikce: {prediction}")
print(f"Reálná hodnota: {diabetes.target[-1:]}")
print(f"MSE: {(prediction-diabetes.target[-1:])**2}")
```

Úkol 5 – **Prozkoumáváme scikit-learn** [5b]: Scikit-learn v sobě obsahuje spoustu implementovaných modelů, užitečných funkcí a různých transformací na datech. My vás necháme prozkoumat dokumentaci scikit-learn[5] a najít si pár užitečných funkcí.

Rozdělte diabetes dataset na trénovací a testovací množinu. Tento krok se dělá z toho důvodu, protože když trénujeme model a trénujeme ho na všech datech, tak nedokážeme otestovat, jak model funguje. Proto si část dat z trénovací množiny odebereme, aby jsme mohli na těchto datech otestovat, jak model funguje.

Následně natrénujte model lineární regrese na trénovací množině. Poté si nechte vypredikovat hodnoty pro testovací množinu a spočítejte chybovou funkci MSE. Nakonec prozkoumejte další metriky implementované v knihovně scikit-learn na měření výkonnosti modelu a zvolte si jednu vhodnou metriku pro daný typ úlohy, která vám přijde užitečná. Svůj výběr metriky krátce zdůvodněte (stačí jedna věta).

Nic nedělejte ručně, na všechno použijte knihovnu scikit-learn.

Všechny úlohy z tohoto seriálu odevzdávejte dohromady v jednom zazipovaném archivu. Všechny úlohy můžete odevzdat i v jednom souboru či Jupyter notebooku, ale tento soubor pořád musí být zazipovaný. Pokud úlohy odevzdáváte v jednom souboru, tak jednotlivé úlohy oddělte do samostatných funkcí. Termín odevzdání je 26. listopadu ve 32:00 (tedy další ráno v 8:00). Poté lze odevzdávat za snížený počet bodů až do konce ročníku.

Když budete mít jakýkoliv problém, tak se nebojte zeptat na našem Discordu či pomocí emailu. Mějte se famfárově a v příštím dílu si natrénujeme ručně lineární regresi!

Michal Kodad & Ondra Sladký

[5] `https://scikit-learn.org/stable/modules/classes.html`

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.[6]

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“.[7]

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 nějaké konkrétní 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 setkáme 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.

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], ...`).[8]

Ve většině základních jazyků je pole jen *statické*, tedy v okamžiku jeho vytváření musíme počítači říct, jak ho chceme

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

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

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

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, je snadné spočítat umístění konkrétní přihrádky. Proto když se počítače zeptáme na obsah přihrády `pole[42]`, vrátí nám hodnotu 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,[9] nejdříve však doporučujeme dočíst tuto kuchařku.

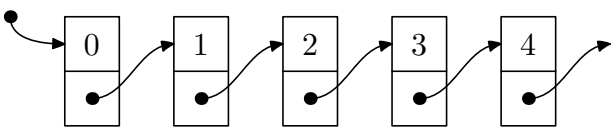
Přidání nového prvku na konec pole také zvládneme v konstantním čase. Problém je přidání nového prvku někam doprostřed (což se nám typicky stane, pokud budeme chtít udržovat hodnoty v poli seřazené a zároveň do něj vkládat nové). V takovém případě se totiž všechny prvky za vkládaným musí posunout o jednu pozici dál, aby se vkládaný prvek vešel na své místo. Taková operace tedy může pro pole délky N (čili pole obsahující 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 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).

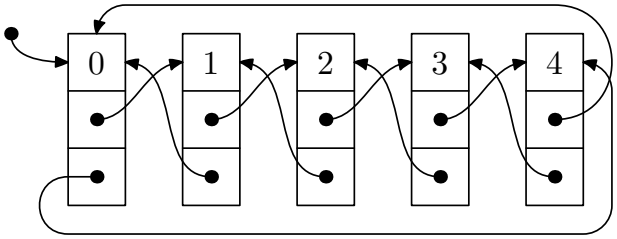


K lepšímu pochopení tohoto principu je důležité si vysvětlit, co to je *ukazatel* (nebo také *odkaz* či anglicky *pointer*). Každé místo v paměti počítače má své číselné označení, kterému říkáme *adresa*. Když si vytváříme nějakou pojmenovanou proměnnou, ta se vlastně odkazuje na určité místo v paměti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

```
#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 =
```

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

```
        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`):

```
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 vloz_po(self, prvek, za_prvek = None):
        if za_prvek is not None:
            prvek.dalsi = za_prvek.dalsi
            prvek.predchozi = za_prvek
            za_prvek.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.vloz_po(prvekB)
seznam.vloz_po(prvekD, prvekB)
seznam.vloz_po(prvekC, prvekD)
seznam.vloz_po(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, také první přijde na řadu. Trochu jinak si ji můžeme 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 do 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, konkrétně na poslední prvek.

Knihovny

Tyto základní struktury už jsou často předpřipravené jako součást určitých *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.

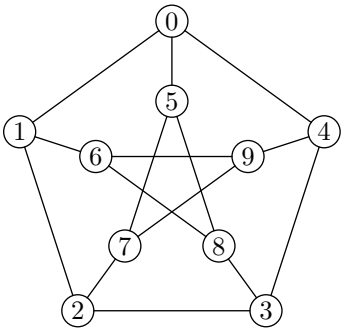
Stromy a grafy v informatice

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 (v popisech bude n 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 (nebo 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 ale dobré o ní vědět.

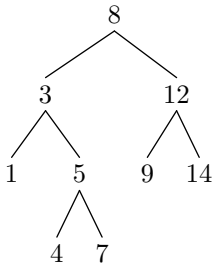
Grafy jsou velmi široké téma. Můžeme hledat jejich minimální kostry, můžeme v nich hledat nejkratší cesty či skrze ně pouštět pod tlakem vodu. Více o nich si tedy můžete přečíst v některé z našich specializovaných grafových kuchařek, které odkazujeme z našeho kuchařkového rozcestníku.[10]

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*.



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

[10] `http://ksp.mff.cuni.cz/kucharky/`

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, 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 podstromu menší než hodnota tohoto vrcholu a hodnoty v jeho pravém podstromu naopak větší.

V takovém stromu 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 sekci *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. Třeba 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.

Kromě 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 – každé 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;

        int a = 0; int b = 1;
        while (n > 1) {
                int pomocna = a + b;
                a = b;
                b = pomocna;
                n--;
        }
```

```
        return b;
}
```

V Pythonu:

```
def fib2(n):
    if n == 0:
        return 0
    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 podkapitolky Předpočítané mezivýsledky).

Rekurzivní varianta v tomto případě může běžet až v čase $\mathcal{O}(2^n)$, což je pro velká n mnohem pomaleji než $\mathcal{O}(n)$. Dá se ale celkem lehce rozmyslet úprava rekurzivní varianty, aby běžele 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

Představme 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, maximálně po $\log n$ krocích se dostaneme na pole velikosti jedna. Říkáme, že algoritmus má *logaritmickou časovou složitost*, píšeme $\mathcal{O}(\log n)$.[11]

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;
```

[11] 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 $\log 2 = 1, \log 8 = 3, \log 1024 = 10$.

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

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

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.

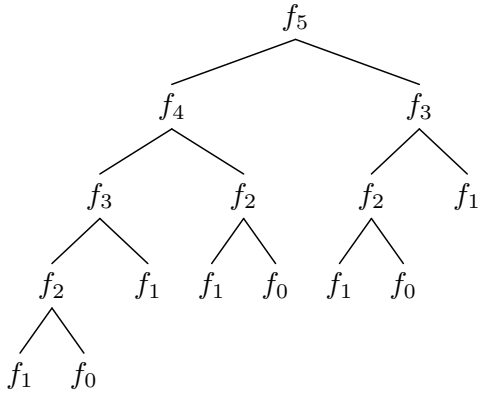
Více se o metodě Rozděl a panuj můžete dozvědět ve stejnojmenné kuchařce.[12]

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 připomenout naši rekurzivní implementaci počítání Fibonacciho čísel zmíněnou výše.

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.

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.[13]

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 sou-

[12] `http://ksp.mff.cuni.cz/viz/kucharky/rozdel-a-panuj`
[13] `http://ksp.mff.cuni.cz/viz/kucharky/dynamicke-programovani`

čet 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];
```

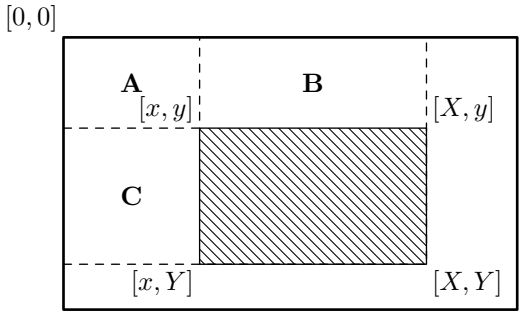
To nám umožňuje snížit čas potřebný na řešení této úlohy na $\mathcal{O}(n^2)$. To je už lepší čas; prozradíme však, že tuto úlohu lze řešit dokonce v lineárním čase, ale to je již nad rámec této kuchařky.

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 $\mathcal{O}_p$, předvýpočet má smysl.

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, nejedná se již 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

matfyz

KSP pro vás připravují studenti Matematicko-fyzikální fakulty Univerzity Karlovy.
Realizace projektu byla podpořena Ministerstvem školství, mládeže a tělovýchovy.

Webové stránky: *https://ksp.mff.cuni.cz/*
E-mail: *ksp@mff.cuni.cz*
Organizátoři a kontakty: *https://ksp.mff.cuni.cz/kontakty/*