Korespondenční Seminář z Programování

35. ročník **KSP** **Září 2022**

Milí řešitelé, řešitelky a řešitelčata!

Dostává se k vám první číslo hlavní kategorie 35. ročníku KSP.

Letos se můžete těšit v každé z pěti sérií hlavní kategorie na **4 normální úlohy**, z toho alespoň jednu praktickou opendatovou. Také 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: **30. října 2022 ve 32:00 (tedy další ráno v 8:00)**

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

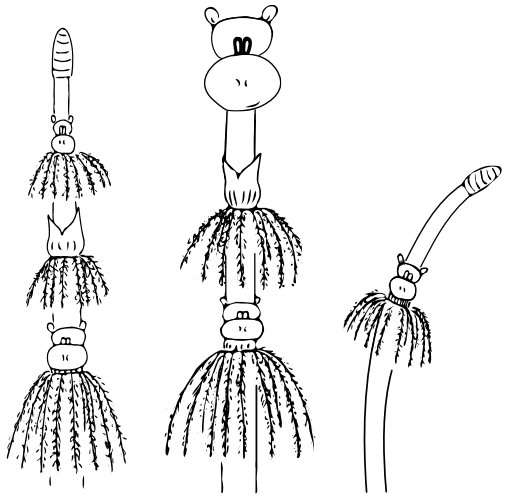
Značky úloh: Lehčí úloha (či její část) vhodná pro začátečníky; Praktická open-data úloha; Úloha, u které doporučujeme začíst se do kuchařky; Seriálová úloha

Odměna série: **Sladkou odměnu** si vysloužíš, kdo **z každé úlohy série získá alespoň 2 body**.

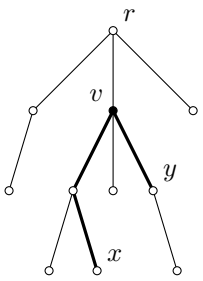
První série třicátého pátého ročníku KSP

35-1-1 Hroší přeslička — 9 bodů

Petr se vypravil do hor pátrat po stopách bájné hroší přesličky (Hippochaete montana). Říká se, že odvar z ní rozjasní mysl a pomůže nejenom při řešení úložek v KSP. Najít ji ale vůbec není jednoduché.



Horu si lze představit jako množinu N křižovatek a $N-1$ cestiček, přičemž každá cestička spojuje dvě křižovatky. Na horách se dá bezpečně pohybovat jedině po cestičkách a křižovatkách. Navíc platí, že mezi libovolnými dvěma křižovatkami lze přecházet pouze jedním způsobem (formálně řečeno se jedná o *strom*). Vypadat může třeba takto:



Křižovatka r odpovídá vrcholu hory (je nejvýš), křižovatka v je níž, y ještě níž a tak dále. Obecně jdeme-li z nějaké křižovatky směrem k r, pohybujeme se nahoru, jinak dolů.

Trasa budeme říkat procházce z jedné křižovatky na jinou, která nenavštíví žádnou křižovatku vícekrát. Každá trasa má jediný *nejvyšší bod* – křižovatku nejbližší vrcholu hory. Na obrázku jsme tučně nakreslili trasu z x do y, jejím nejvyšším bodem je v. Trasa z x do v má nejvyšší bod v.

Teď uvážíme všechny možné trasy. Jelikož každá trasa je jednoznačně určena svou první a poslední křižovatkou, existuje právě $N \cdot (N-1)$ různých tras.

Legenda praví, že hroší přeslička roste pouze na takovém místě, které se *nejčastěji* vyskytuje jako nejvyšší bod trasy.

Na našem obrázku je například křižovatka v nejvyšším bodem 34 tras. (6 tras vede zespodu do v, dalších 6 z v dolů,

22 tras má začátek i konec někde pod v). Ale ještě častěji se vyskytuje křižovatka r – je nejvyšším bodem 66 tras.

Petr se rozhodl, že přesličku najde. V atlase si proto našel mapu hory s vyznačenými cestičkami. Pomůžete mu určit, kam by se měl vypravit, aby hroší přesličku zaručeně získal?

Vaším úkolem je vymyslet a sepsat algoritmus, který na vstupu dostane popis terénu hory a na výstupu vrátí identifikátor křižovatky, na které podle legendy roste hroší přeslička. Pokud takových míst existuje více, nechť vypíše libovolné z nich.

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

Můžete předpokládat, že každá křižovatka má unikátní celočíselný identifikátor z rozmezí od 1 do N. Vrchol hory se nachází na křižovatce číslo 1. Samotný terén je následně popsaný pomocí $N-1$ cestiček ve tvaru `U V`, kde `U` a `V` jsou křižovatky na okrajích cestičky a zároveň `U` se nachází nad `V`.

35-1-2 Vaření lektvarů — 10 bodů

Aspirující alchymista Demivirtos dostal zakázku na dodání sady lektvarů. Jelikož se snaží vybudovat si dobrou pověst, chce ji splnit co nejrychleji. Ve svojí nově rozšířené laboratoři dokáže vařit prakticky neomezené množství lektvarů zároveň, problém však je, že složitější lektvary používají jako ingredience ty jednodušší. Může je tedy začít připravovat, až když jsou dovařené všechny jejich prerekvizity. Jakmile však už jeden lektvar dodělá, má ho neomezené množství, takže ho může použít na výrobu dalších lektvarů, a dokonce mu zbude.

Spočítejte, kdy může Demivirtos začít připravovat jednotlivé lektvary, aby měl zakázku připravenou co nejdříve. Zakázka je připravená, když jsou hotové všechny lektvary.

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 čísla N – počet různých lektvarů a M – počet závislostí.

Na druhém řádku dostanete N čísel S_i říkající, jak dlouho trvá připravit i-tý lektvar.

Na každém ze zbylých M řádků je pak dvojice čísel P_j a Q_j. Každá říká, že Q_j-tý lektvar se může začít připravovat až po dokončení P_j-tého. Lektvary jsou číslovány od 1 do N.

Máte zaručeno, že postup, jak lektvary uvařit, skutečně existuje – nestane se tedy, že dva nebo více lektvarů bude čekat vzájemně na sebe.

Formát výstupu: Vypište jeden řádek s N čísly S_i říkající, kdy se má začít připravovat i-tý lektvar.

Lehčí varianta (za 4 body): Slibujeme, že v prvních dvou vstupech bude každý lektvar vyžadovat maximálně jeden jiný.

Ukázkový vstup:

```
5 3
2 2 3 4 1
3 4
1 2
2 4
```

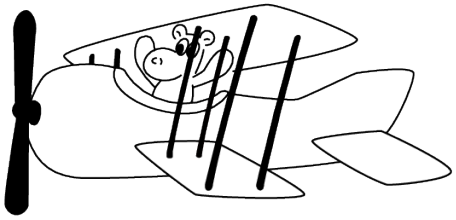
Ukázkový výstup:

```
0 2 1 4 6
```

Zakázka bude připravená v čase 8, kdy se dokončí čtvrtý lektvar, který potřebuje první tři. Pátý lektvar, který nic nepotřebuje, se může vařit kdykoliv mezitím.

35-1-3 Letadélko — 12 bodů

Kevin se Sárou byli o prázdninách v Řecku. A protože Kevin už měl koupání u moře dost, rozhodl se, že vyzkouší svůj model letadélka.



Kevin svoje letadélko pustil z pláže na úrovni mořské hladiny a nasměroval ho k Sáře, která čekala na druhém ostrově. Letadélko se začalo pohybovat vpřed, kromě toho dělalo i pohyby nahoru a dolů. Na konci slétlo zpět na úroveň mořské hladiny, aby ho Sára mohla chytit. Letadélko ale není voděodolné, takže nikdy nemohlo klesnout pod úroveň mořské hladiny.

Kevin se chtěl následně podívat na záznam letové výšky, bohužel ale zjistil, že slaný mořský vzduch letadélku nesvědčí a některé části záznamu jsou vymazané. Vzhledem k tomu, že poškozených částí je velmi málo, Kevin si myslí, že by mohl projít všechny možnosti, jak mohlo letadlo letět, a z toho, co si pamatuje, najít tu správnou. Řekněte mu, kolik možností je, aby tušil, jak dlouho mu to bude trvat.

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 dostatne číslo N – délku záznamu.

Na druhém řádku dostanete záznam složený ze tří symbolů: `^` znamená, že letadlo letělo o 1 metr nahoru, `v` znamená, že letadlo letělo o 1 metr dolů. `?` znamená, že záznam je zde poškozený a letadlo mohlo letět o 1 metr dolů nebo nahoru.

Slibujeme, že počet poškozených částí je velmi malý.

Formát výstupu: Vypište jedno číslo – počet možností, jak lze otazníky nahradit `^` nebo `v`, aby let letadélka splňoval dvě podmínky:

- Letadélko startovalo ve výšce 0 a skončilo ve výšce 0.
- Letadélko nikdy nekleslo pod mořskou hladinu.

Počet možností může být opravdu velký, a proto bude v některých jazycích zapotřebí použít 64-bitového datového typu (v C++ je to `long long int`). Pokud programujete v Pythonu, pak toto řešit nemusíte.

Ukázkový vstup:	*Ukázkový výstup:*
`4` `^??v`	`2`

Existují dvě různé možnosti letu: `^^vv` a `^v^v`.

35-1-4 Atlas zlomků — 14 bodů

Velký detektiv K. S. Pinkerton luští tajnou zprávu, ve které jsou slova zašifrovaná do zlomků. Všiml si, že všechny zlomky ve zprávě jsou v základním tvaru (nedají se zkrátit), leží mezi 0 a 1 (včetně) a jejich čitatelé i jmenovatelé jsou v rozsahu 1 až N. Hodil by se mu seznam všech takových zlomků.

Navrhněte algoritmus, který pro zadané N vypíše všechny zlomky uvedených vlastností. Pokuste se o co nejlepší časovou složitost vzhledem k počtu vypsaných zlomků. Zlomky můžete vypsat v libovolném pořadí, ale každý by se měl objevit právě jednou.

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

Příklad: Pro $N = 5$ to jsou zlomky 1/1, 1/2, 1/3, 1/4, 1/5, 2/3, 2/5, 3/4, 3/5, 4/5.

35-1-X1 Rejstřík zlomků — 10 bodů

Podobně jako v úloze 35-1-4, i zde budeme hledat zlomky v základním tvaru mezi 0 a 1, jejichž čitatel i jmenovatel jsou v rozsahu 1 až N. Vypište nejmenších K takových zlomků uspořádaných vzestupně. Slibujeme, že $K \geq N$.

Příklad: Pro $N = 5$ a $K = 6$ máte vypsat seznam 1/5, 1/4, 1/3, 2/5, 1/2, 3/5.

35-1-S Lineární rovnice — 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. 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 ponoříme do světa lineární algebry, do světa vektorů a matic, které elegantně popisují geometrické objekty a operace s nimi. Lineární algebra vládne vědeckým výpočtům, a to i v oblastech, které s geometrií nemají téměř nic společného. Nabídne nám algoritmy na problémy, které se jinak řeší těžko, a mnoho dalších algoritmů pomocí jejích triků dokážeme zefektivnit. Vstupenku do tohoto světa naleznete v následujících odstavcích.

Vektory

Aritmetický vektor není nic jiného než seznam čísel. Těmto číslům říkáme *složky* vektoru. Může jich být libovolný počet d, pak vektoru říkáme *d-rozměrný* neboli *d-dimenzionální.* Trojrozměrný vektor může vypadat třeba takhle:

$$\begin{pmatrix} 3 \\ 5 \\ 1 \end{pmatrix}$$

Složky tedy zapisujeme do sloupce pod sebe a celý vektor je obalený závorkami. Když budeme potřebovat zapsat vektory kompaktněji, použijeme řádkový zápis. Symbol $^\top$ značí *transpozici*, se kterou se více seznámíme u matic:

$$\begin{pmatrix} 1 \\ 3 \end{pmatrix} = (1, 3)^\top$$

Stejně jako známe obor reálných čísel $\mathbb{R}$, obor všech d-rozměrných vektorů složených z reálných čísel značíme $\mathbb{R}^d$.

Vektory si lze představit jako body v d-dimenzionálním prostoru. Nejčastěji se budeme setkávat se dvoudimenzionálními vektory, které si představíme jako body v rovině: jejich složky udávají souřadnice bodu. Běžné jsou také třídimenzionální vektory reprezentující body v prostoru. Vektor si také můžeme vyložit jako posunutí z počátku souřadnic do zmíněného bodu. Vektor, který leží v počátku a nikam neveden (takže všechny jeho složky jsou nulové) se nazývá *nulový vektor* a budeme jej značit $\mathbf{0}$.

Prostým reálným číslům se (pro lepší odlišení od vektorů) říká *skaláry*. Abychom odlišení dosáhli i v matematice, budeme vektory psát tučně: $\mathbf{x}$. Jednodimenzionální vektor má jedinou složku a lze jej tedy považovat jak za vektor, tak za skalár.

Vektory stejných rozměrů můžeme sčítat jednoduše sečtením po složkách. Stejně tak vynásobení vektoru skalárem bude znamenat prosté vynásobení každé ze složek. Této operaci budeme také říkat *škálování*. To zní podobně jako skalár, náhoda?

$$\begin{pmatrix} 1 \\ 3 \end{pmatrix} + \begin{pmatrix} 5 \\ -1 \end{pmatrix} = \begin{pmatrix} 6 \\ 2 \end{pmatrix}$$

$$2 \begin{pmatrix} 1 \\ 3 \end{pmatrix} = \begin{pmatrix} 2 \\ 6 \end{pmatrix}$$

Lineární kombinace

Množina všech vektorů $\mathbb{R}^d$ má hezkou vlastnost: Součet dvou vektorů z této množiny opět leží v této množině. Totéž platí pro vynásobení vektoru reálným skalárem. Pokuste se odůvodnit, proč tomu tak je. Jelikož tato vlastnost vyvstává v matematice poměrně často, má své vlastní jméno: Říkáme, že množina $\mathbb{R}^d$ je *uzavřená* na operace sčítání a násobení skalárem. Množině, která je uzavřená na tyto dvě operace (a kde je sčítání a násobení definované „rozumně“), se říká *vektorový prostor*. S vektorovými prostory se blíže seznámíme ve třetím dílu.

Vektorový prostor můžeme popsat pomocí souřadnic. Nejprve zavedeme *jednotkové vektory,* které obsahují samé nuly a jednu jedničku. Například ve trojrozměrném prostoru to jsou $(1, 0, 0)^\top$, $(0, 1, 0)^\top$ a $(0, 0, 1)^\top$. Pomocí jednotkových vektorů pak můžeme zapsat libovolný vektor:

$$\begin{pmatrix} a \\ b \\ c \end{pmatrix} = a \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} + b \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} + c \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

Takovému zápisu říkáme *lineární kombinace* jednotkových vektorů. Něco takového si můžeme dovolit jedině díky uzavřenosti, která nám zaručuje, že všechny mezivýsledky budou ležet v $\mathbb{R}^3$.

Možná jste na střední škole potkali analytickou geometrii, která popisuje rovinu pomocí kartézské soustavy souřadnic. Totéž jde říci pomocí lineární algebry: rovina odpovídá vektorovému prostoru $\mathbb{R}^2$, souřadné osy mají stejný směr jako

jednotkové vektory a konkrétní souřadnice bodu odpovídají složkám vektoru.

Lineární kombinace umí pracovat i s jinými vektory než jednotkovými. Obecně nám umožňuje vzít několik libovolných vektorů, ze kterých poskládáme nový vektor tím, že je přeškálujeme a poté sečteme. Zapsáno matematicky, lineární kombinace $\mathbf{v}$ vektorů $\mathbf{x}_1, \ldots, \mathbf{x}_n$ je následující výraz pro libovolné hodnoty koeficientů $a_1, \ldots, a_n$:

$$\mathbf{v} = \sum_{i=1}^{n} a_i \mathbf{x}_i$$

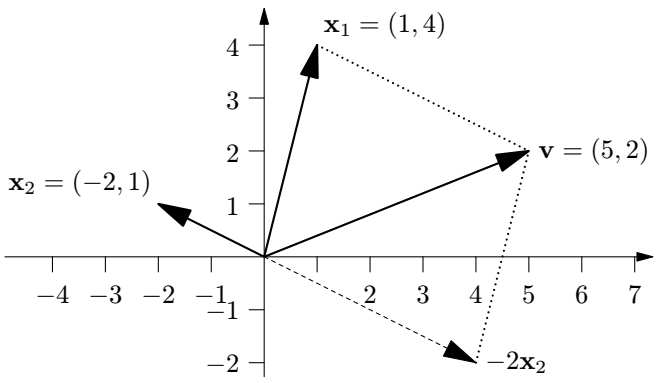
Pokud jste se ještě nesetkali se sumou ($\sum$), výraz výše znamená: Pro všechny hodnoty i od 1 do n, vezmi výraz $a_i \mathbf{x}_i$ a všechny tyto výrazy sečti. Po rozepsání bychom tedy dostali:

$$\mathbf{v} = a_1 \mathbf{x}_1 + a_2 \mathbf{x}_2 + \ldots + a_n \mathbf{x}_n$$

Co to celé vlastně znamená? Dejme tomu, že se v prostoru umíme pohybovat pouze některými směry, $\mathbf{x}_1, \ldots, \mathbf{x}_n$, a chceme se dostat do bodu určeného vektorem $\mathbf{v}$. Pokud najdeme vhodné koeficienty, pro které je $\mathbf{v}$ lineární kombinací směrů pohybu, zjistíme, jak daleko musíme v každém z těchto směrů ujít, abychom se do daného bodu dostali.

Následující obrázek ilustruje lineární kombinaci

$$\mathbf{v} = \mathbf{x}_1 - 2\mathbf{x}_2.$$



Úkol 1 – Robot na Marsu [5b]:

Vesmírná sonda Hipporover přistává na Marsu. Po přistání se má vydat směrem k nejbližšímu kráteru. Sonda má dvě sady koleček, jedny jí umožní jet dopředu, jedny do strany. Pro účely navigace nahráli výzkumníci do sondy mapu povrchu Marsu v kartézském souřadnicovém systému a naprogramovali přesnou posloupnost pohybů, jaké sonda musí vykonat, aby se ke kráteru dostala.

Každý pohyb má udaný směr (dopředu, doleva) a vzdálenost, jakou má sonda v tomto směru ujet. Tato vzdálenost může být záporná, poté jede sonda opačným směrem. Pohyb dopředu o jeden metr odpovídá vektoru $(1, 0)^\top$, pohyb doleva vektoru $(0, 1)^\top$.

Ale ouha, sonda měla potíže při přistání a některé součástky se zkratovaly. Teď jde do nějakých koleček více napětí, než by mělo, a sonda nejezdí rovně, dokonce ani směry jejich pohybů již na sebe nejsou kolmé.

Naštěstí má sonda kalibrační systém a dokáže zjistit vektory svých pohybů vůči kartézské soustavě souřadnic. Skvělé, ale jak nyní opravit plán a dostat vozítko do cílového místa? Výzkumníci si s tím nevědí rady. Pomozte jim!

Dostanete dva dvourozměrné vektory. První popisuje, jak se sonda pohne, pokud dostane povel popojet o jeden metr dopředu. Druhý popisuje totéž pro povel doleva. Můžete předpokládat, že tyto vektory jsou nenulové a mají různý směr.

Popište, jak simulovat původní povely pomocí nových pohybů. Tedy jaké pohyby má sonda provést pro povel „popojeď dopředu o m metrů“ a jaké pro povel „doleva o m metrů“.

Lineární obal

Mohlo by nás také zajímat, do jakých všech bodů se lze pomocí pohybů $\mathbf{x}_1, \ldots, \mathbf{x}_n$ dostat. To tedy bude množina všech lineárních kombinací těchto vektorů pro všechny možné hodnoty koeficientů. Tato množina se nazývá *lineární obal* a vektory $\mathbf{x}_1, \ldots, \mathbf{x}_n$ se v tomto kontextu nazývají *generátory*.

Funkce, která počítá lineární obal, se značí span. Tato funkce má jeden argument, konkrétně množinu generátorů. Lineární obal proto budeme značit $\text{span}\{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$.

Jak lineární obal vypadá? Pokud se nemůžeme pohybovat žádným směrem, zůstaneme v počátku, lineární obal je tedy jediný bod. Můžeme-li se pohybovat jedním směrem, množina všech dosažitelných bodů má podobu přímky procházející počátkem. Přidáním druhého generátoru lze vyrobit z přímky rovinu, přidáním třetího trojrozměrný prostor.

Lineární obal je uzavřený na lineární kombinace stejně jako celý vektorový prostor $\mathbb{R}^d$. (Nevěříte? Zkuste to dokázat, není to těžké.) Je tedy taktéž vektorovým prostorem. Protože je zároveň podmnožinou prostoru $\mathbb{R}^d$, říká se mu *podprostor*.

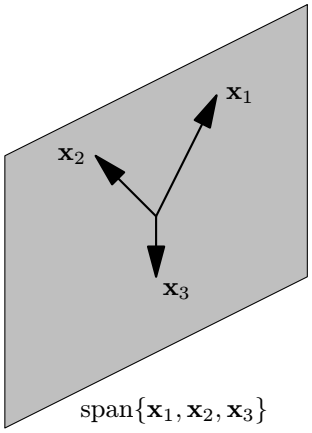
Lineární závislost a nezávislost

Před chvílí jsme nahlédli, že lineární obal dvou vektorů tvoří rovinu. Je tomu tak ale vždy? Zkuste najít vektory $\mathbf{x}_1, \mathbf{x}_2$, které dávají protipříklad.

Může se stát, že lineární obal bude pouze přímka, a to v případě, že je $\mathbf{x}_1$ násobkem $\mathbf{x}_2$. Poté už $\mathbf{x}_2$ leží ve $\text{span}\{\mathbf{x}_1\}$, takže „nic nepřidá“. Říkáme, že vektory $\mathbf{x}_1, \mathbf{x}_2$ jsou *lineárně závislé*.

Tento pojem můžeme zobecnit pro více vektorů. Vektory $\mathbf{x}_1, \ldots, \mathbf{x}_n$ jsou lineárně závislé právě tehdy, když lze jeden z nich vyjádřit jako lineární kombinaci zbytku. Ekvivalentně platí, že některý z těchto vektorů je nadbytečný a jeho odebráním z generátorů se lineární obal nezmění.

Na následujícím obrázku je znázorněn lineární obal tří vektorů. Libovolné dva z nich by vygenerovaly stejnou množinu.



Pakliže mezi vektory není žádný závislý na ostatních, řekneme, že jsou vektory *lineárně nezávislé*. Podíváme-li se do učené knihy, pravděpodobně najdeme pro nezávislost jinou definici: Vektory $\mathbf{x}_1, \ldots, \mathbf{x}_n$ jsou nezávislé právě tehdy, když rovnice

$$\sum_{i=1}^{n} a_i \mathbf{x}_i = \mathbf{0}$$

platí pouze, jsou-li všechny koeficienty $a_i = 0$.

Uvědomme si, že je tato definice ekvivalentní. Kdyby existovala netriviální lineární kombinace (s některými koeficienty nenulovými), poté můžeme tento nenulový člen převést na druhou stranu a příslušným koeficientem rovnici podělit, čímž získáme vyjádření tohoto vektoru jako lineární kombinaci zbytku.

Přidáme jednu vlastnost: Jsou-li $\mathbf{x}_1, \ldots, \mathbf{x}_n$ lineárně nezávislé a

$$\mathbf{v} = \sum_{i=1}^{n} a_i \mathbf{x}_i,$$

poté je toto vyjádření jednoznačné, tedy existuje pouze jedna sada koeficientů a_i, která tuto rovnici splní.

Zde již důkaz nemusí být na první pohled vidět. Jak vůbec ukázat, že je vyjádření jednoznačné? Využijeme důkaz sporem: Budeme předpokládat, že existují dvě vyjádření. Odtud se pokusíme dojít ke sporu s některým z našich předpokladů, v tomto případě s předpokladem lineární nezávislosti. Připomeňte si definici nezávislosti a doplňte detaily. Pokud si nebudete vědět rady, řešení najdete na konci textu.

Afinní množiny

Již umíme reprezentovat přímky, roviny atd. jako lineární obaly jejich generátorů, ovšem pouze, pokud procházejí počátkem. S tím se jistě nespokojíme. Jak popsat tyto množiny, které leží v obecné poloze, takzvané *afinní množiny*?

Nejprve si představíme, že afinní množinu posuneme, a to tak, aby některý její vektor $\mathbf{v}$ skončil v počátku. Posunutou množinu již umíme reprezentovat jako podprostor U. Posunutím zpět o vektor $\mathbf{v}$ získáme hledanou afinní množinu. Protože posunutí znamená ke každému vektoru z U přičíst $\mathbf{v}$, značíme výsledek $U + \mathbf{v}$.

Dokonce existuje *afinní kombinace* jako analogie dříve zavedené lineární kombinace. Podobně jako dříve chceme, aby všechny afinní kombinace vektorů $\mathbf{x}_1, \ldots, \mathbf{x}_n$ daly dohromady tu nejmenší afinní množinu, jež tyto vektory obsahuje.

Nejprve se zkusme zamyslet nad afinní kombinací dvou bodů $\mathbf{x}_1, \mathbf{x}_2$. Afinní množina, kterou hledáme, je přímka jimi určená. Jak matematicky tuto přímku popsat? Možná bude jednodušší uvážit úsečku mezi těmito body. Bod $\mathbf{v}$ ležící na úsečce je jakýmsi váženým průměrem krajních bodů:

$$\mathbf{v} = a_1 \mathbf{x}_1 + (1 - a_1)\mathbf{x}_2$$

Pro úsečku bychom uvažovali $0 \leq a_1 \leq 1$. Pokud povolíme libovolné hodnoty koeficientů, získáme celou přímku.

Máme tedy vzorec pro afinní kombinaci dvou vektorů. Zobecnění pro více vektorů pak je:

$$\mathbf{v} = \sum_{i=1}^{n} a_i \mathbf{x}_i, \text{ přičemž } \sum_{i=1}^{n} a_i = 1$$

Čili vzorec zůstává stejný jako u lineárních kombinací, jen navíc požadujeme, aby součet koeficientů byl 1.

Afinní závislost můžeme zavést analogicky tak, že je některý z vektorů při generování afinní množiny nadbytečný.

Soustavy rovnic

Probravše mnoho teorie se konečně dostáváme k praktickému využití: Naučíme se řešit soustavy lineárních rovnic, jako třeba tuto:

$$\begin{aligned} x_1 + \ x_2 - 6x_3 &= -9 \\ x_1 + 4x_2 - \ x_3 &= 16 \\ 3x_1 - \ x_2 - \ x_3 &= -13 \end{aligned}$$

Obecně budeme mít d neznámých $x_1, \ldots, x_d$, které sdružíme do vektoru $\mathbf{x}$. Množina všech potenciálních řešení je tedy náš známý prostor $\mathbb{R}^d$. Dále dostaneme n rovnic, které množinu řešení omezují.

V příkladu výše popisuje každá rovnice jednu rovinu ve třídimenzionálním prostoru. Průnik prvních dvou rovin je přímka, přidáním poslední roviny se z průniku stane jediný bod.

Obecně vymezují rovnice *nadroviny*, tedy afinní množiny dimenze $d-1$ v prostoru $\mathbb{R}^d$. Průnik k nadrovin v d-dimenzionálním prostoru bude $(d-k)$-dimenzionální afinní množina. Pokud tedy budeme mít d nadrovin, jejich průnik bude mít dimenzi 0 a hledané řešení $\mathbf{x}$ bude jediný bod.

Tedy až na případy, kdy tomu tak není. Může se stát, že průnik prvních několika nadrovin je rovnoběžný s další přidávanou nadrovinou. Pak řešení existovat nebude. Případně se může stát, že průnik celý leží v nadrovině, poté se přidáním této nadroviny dimenze nesníží. V jistém smyslu je tento případ podobný lineární závislosti, také je jedna z rovnic nadbytečná. Tato spojitost není náhodná.

Úkol 2 – Nezávislost a soustavy [3b]:

Když jsme zaváděli lineární nezávislost, neřekli jsme si, jak ji ověřit. K tomu nám poslouží právě soustavy rovnic. Popište, jak s jejich pomocí určit, že jsou vektory nezávislé, respektive jak najít koeficienty, které vyjádří jeden z nich jako lineární kombinaci zbytku.

Gaussova eliminace

Při řešení se budeme snažit z rovnic eliminovat neznámé tak, aby šlo jednoduše dopočítat hodnoty těch ostatních. K tomu nám budou sloužit *elementární úpravy:*

1) *Vynásobení rovnice nenulovou konstantou.*
2) *Přičtení jedné rovnice k druhé.* Případně můžeme spojit tuto úpravu s minulou a přičíst násobek jedné rovnice k druhé. Pochopitelně můžeme také odečítat.
3) *Výměna dvou rovnic.*

Na těchto úpravách je založený algoritmus známý pod jménem *Gaussova eliminace.* Pojďme si ho ukázat na soustavě uvedené výše. Nejprve eliminujeme x_1 ze všech rovnic kromě té první. Dosáhneme toho pomocí 2. elementární úpravy, budeme přičítat násobky první rovnice k ostatním. K druhé rovnici přičteme (-1)-násobek, ke třetí (-3)-násobek.

$$\begin{aligned} x_1 + \ x_2 - \ 6x_3 &= -9 \\ 0 + 3x_2 + \ 5x_3 &= 25 \\ 0 - 4x_2 + 17x_3 &= 14 \end{aligned}$$

Pokračujeme eliminací x_2. Při tom nesmíme znovu zanést x_1 tam, kde jsme jej eliminovali, proto od teď budeme pracovat pouze s druhou a třetí rovnicí.

Chceme tedy eliminovat x_2 ze třetí rovnice. Mohli bychom k ní přičíst 4/3-násobek té druhé, tím by nám však vznikly zlomky, takže raději nejprve vynásobíme třetí rovnici 3 a poté přičteme 4-násobek druhé rovnice.

$$\begin{aligned} x_1 + x_2 - \ 6x_3 &= -9 \\ 3x_2 + \ 5x_3 &= 25 \\ -12x_2 + 51x_3 &= 42 \end{aligned}$$

$$\begin{aligned} x_1 + x_2 - \ 6x_3 &= -9 \\ 3x_2 + \ 5x_3 &= 25 \\ 71x_3 &= 142 \end{aligned}$$

Z poslední rovnice již zvládneme dopočítat $x_3 = 2$. Dosazením do druhé rovnice dostaneme

$$3x_2 + 5 \cdot 2 = 25,$$

tedy $x_2 = 5$. Po dosazení obou hodnot do první rovnice získáme $x_1 = -2$.

Podívejme se ještě na jeden případ, ve kterém nevyjde řešení jednoznačně.

$$\begin{aligned} 2x_2 + 6x_3 &= 14 \\ x_1 - 3x_2 + 2x_3 &= 5 \\ -x_1 + 4x_2 + \ x_3 &= 2 \end{aligned}$$

V první rovnici chybí x_1. Je zvykem eliminovat tak, aby nuly vznikaly v levém spodním trojúhelníku, proto rovnice prohodíme.

$$\begin{aligned} x_1 - 3x_2 + 2x_3 &= 5 \\ 2x_2 + 6x_3 &= 14 \\ -x_1 + 4x_2 + \ x_3 &= 2 \end{aligned}$$

Eliminujeme x_1 ze třetí rovnice přičtením první.

$$\begin{aligned} x_1 - 3x_2 + 2x_3 &= 5 \\ 2x_2 + 6x_3 &= 14 \\ x_2 + 3x_3 &= 7 \end{aligned}$$

Eliminujeme x_2 ze třetí rovnice odečtením poloviny druhé rovnice.

$$\begin{aligned} x_1 - 3x_2 + 2x_3 &= 5 \\ 2x_2 + 6x_3 &= 14 \\ 0 &= 0 \end{aligned}$$

Poslední rovnice nám již nedává žádnou informaci. Alespoň platí, jinak by řešení neexistovalo. Takto je řešení více, dokonce nekonečně mnoho. Pro každou myslitelnou hodnotu x_3 umíme dopočítat zbylé neznámé. Prohlásíme ji tedy za parametr a řešení vyjádříme v závislosti na něm.

$$\begin{aligned} x_2 &= -3x_3 + 7 \\ x_1 &= -11x_3 + 26 \end{aligned}$$

Alternativně lze řešení zapsat vektorově:

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 26 \\ 7 \\ 0 \end{pmatrix} + x_3 \begin{pmatrix} -11 \\ -3 \\ 1 \end{pmatrix}$$

Úkol 3 – Algoritmus eliminace [7b]:

Naprogramujte algoritmus Gaussovy eliminace. Na vstupu dostanete číslo d a poté d rovnic. Každá rovnice sestává z $d+1$ celých čísel, prvních d určuje koeficienty u neznámých $x_1, \ldots, x_d$, poslední číslo je pravá strana rovnice. Můžete předpokládat, že každá neznámá má v alespoň jedné rovnici nenulový koeficient.

Na první řádek výstupu vypište `N`, pokud řešení neexistuje, `J` je-li jednoznačné, `P` je-li parametrizované. V jednoznačném případě na další řádek vypište d hodnot neznámých. V parametrizovaném případě použijte vektorový zápis (viz výše), nejprve tedy na jeden řádek vypište hodnoty neznámých nezávislé na parametrech, poté na dalších d řádků vypište vždy po d hodnotách. Je-li i-tá neznámá parametrem, poté na i-tém řádku budou neznámé vyjádřené vzhledem k tomuto parametru. Pro neparametrizované neznámé vypište d nul.

Lehčí varianta (za 4 body): Nemusíte řešit parametrizované případy.

K této úloze si nelze generovat vstupy v odevzdávátku, odevzdejte přímo zdrojový kód. Vaše řešení by mělo zejména mít rozumnou časovou složitost. Kromě toho se můžete pokusit naprogramovat algoritmus tak, aby bylo spočítané řešení přesné. Počítání s desetinnými čísly totiž vede k odchylkám.

Řešená soustava s jednoznačným řešením:

Ukázkový vstup:

```
3
1 1 -6 -9
1 4 -1 16
3 -1 -1 -13
```

Ukázkový výstup:

```
J
-2 5 2
```

Řešená soustava s parametrizovaným řešením:

Ukázkový vstup:

```
3
0 2 6 14
1 -3 2 5
-1 4 1 2
```

Ukázkový výstup:

```
P
26 7 0
0 0 0
0 0 0
-11 -3 1
```

Všechny úlohy z tohoto seriálu odevzdávejte dohromady v jednom zazipovaném archivu. Termín odevzdání je 13. listopadu 2022 ve 32:00 (tedy další ráno v 8:00). Poté lze odevzdávat za snížený počet bodů až do konce ročníku.

Řešení důkazů

Jednoznačnost lineární kombinace. Pro spor předpokládejme, že máme dvě různá vyjádření

$$\mathbf{v} = \sum_{i=1}^{n} a_i \mathbf{x}_i = \sum_{i=1}^{n} b_i \mathbf{x}_i.$$

Tyto rovnice od sebe můžeme odečíst.

$$\mathbf{0} = \sum_{i=1}^{n} (a_i - b_i) \mathbf{x}_i$$

Dle předpokladu byla vyjádření různá, tudíž pro alespoň jedno i platí $a_i - b_i \neq 0$. Tedy jsme našli netriviální lineární kombinaci, což je spor s předpokladem lineární nezávislosti vektorů $\mathbf{x}_1, \ldots, \mathbf{x}_n$.

David Klement

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.[1]

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“.[2]

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], ...`).[3]

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

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

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

[3] 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,[4] nejdříve však doporučujeme dočíst tuto kuchařku.

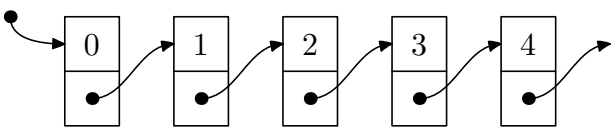
Přidání nového prvku na konec pole také zvládneme v konstantním čase. Problém je přidání nového prvku někam doprostřed (což se nám typicky stane, pokud budeme chtít udržovat hodnoty v poli seřazené a zároveň do něj vkládat nové). V takovém případě se totiž všechny prvky za vkládaným musí posunout o jednu 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).

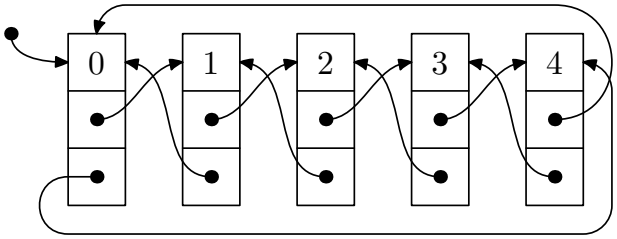


K lepšímu pochopení tohoto principu je důležité si vysvětlit, co to je *ukazatel* (nebo také *odkaz* či anglicky *pointer*). Každé místo v paměti počítače má své číselné označení, kterému říkáme *adresa*. Když si vytváříme nějakou pojmenovanou proměnnou, ta se vlastně odkazuje na určité místo v paměti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

[4] `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.

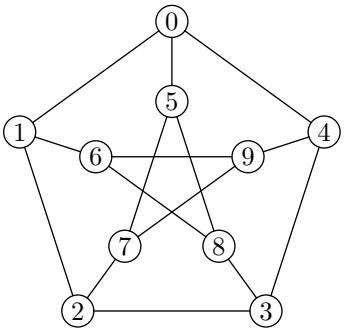
Stromy a grafy v informatice

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.

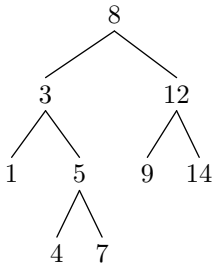
Grafy jsou velmi široké téma. Můžeme hledat jejich minimální kostry, můžeme v nich hledat nejkratší cesty či skrze ně pouštět pod tlakem vodu. Více o nich si tedy můžete přečíst v některé z našich specializovaných grafových kuchařek, které odkazujeme z našeho kuchařkového rozcestníku.[5]

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

[5] `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 $\mathcal{O}(\log n)$.[6]

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;
```

[6] 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.

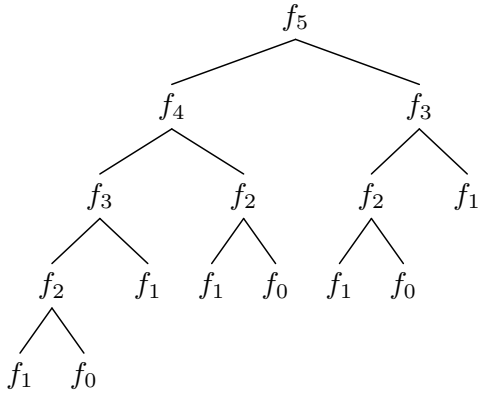
Více se o metodě Rozděl a panuj můžete dozvědět ve stejnojmenné kuchařce.[7]

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.[8]

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-

[7] `http://ksp.mff.cuni.cz/viz/kucharky/rozdel-a-panuj`
[8] `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];
```

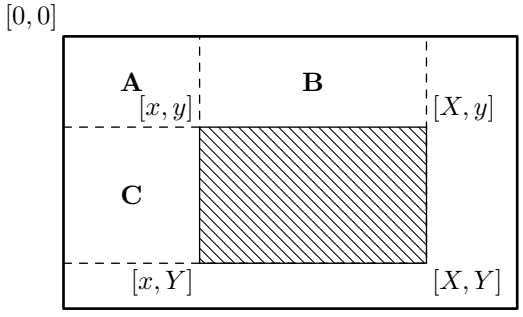
To nám umožňuje snížit čas potřebný na řešení této úlohy na $\mathcal{O}(n^2)$. To je už lepší čas; prozradíme však, že tuto úlohu lze řešit dokonce v lineárním čase, ale to je již nad rámec této kuchařky.

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

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/*