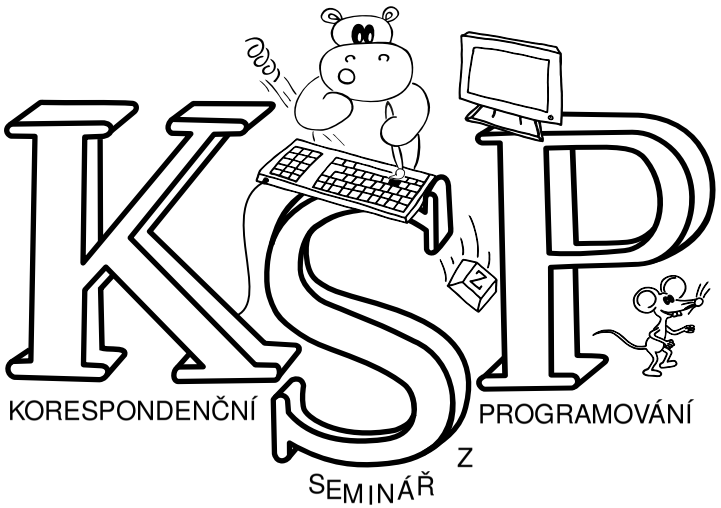
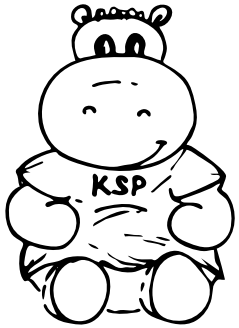
Dokud existují počítače, bude existovat i KSP!





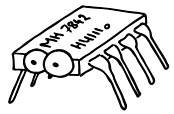
Toužíš po nových vědomostech?

Chceš poznávat nové lidi?

Zajímáš se o počítače?

Láká Tě trocha soutěžení?

Hledáš výzvu pro svou hlavu?



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

Co je KSP?

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

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

Kdo seminář organizuje?

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

Co najdu v zadání?

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

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

Jak úlohy vypadají?

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

V praktických úlohách nejde o popis, ale o výsledek. U úloh (jsou open-data) si stáhneš vstupní data, která zpracuješ Tebou zvoleným způsobem, nejlépe programem v libovolném programovacím jazyce. Výstup odevzdáš a ihned vidíš, zda je výsledek správný.

Vymýšlení mi nejde, co s tím?

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

Proč mám KSP řešit?

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

Díky KSP můžeš poznat informatiku v celé její kráse – mocné programy, magické datové struktury... prostě to, co se ve škole nedozvíš.

To může být užitečné nejen při řešení matematické olympiády kategorie P. Navíc nejlepší řešitele zveme na soustředění, kde můžeš poznat nové kamarády.

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

Hmmm... soustředění?

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

Dostanu i něco hmotnějšího?

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

Vůbec nevím, jak začít...

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

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

Napadá mě jen špatné řešení

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

Co když mi něco není jasné?

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

Zadání

KSP-Z: *http://ksp.mff.cuni.cz/z/*

KSP-H: *http://ksp.mff.cuni.cz/*

Studijní texty

http://ksp.mff.cuni.cz/encyklopedie/

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.

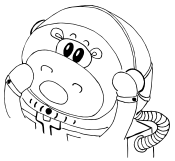
Webové stránky: *https://ksp.mff.cuni.cz/*
E-mail: *ksp@mff.cuni.cz*
Diskusní fórum: *https://ksp.mff.cuni.cz/forum/*

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

Korespondenční Seminář z Programování

32. ročník **KSP** **Září 2019**

Milí řešitelé, řešitelky a řešitelčata!



Právě se k vám dostává první číslo jubilejního ročníku KSP – ano, KSP letos slaví 2^5 let své existence a toto kulaté číslo si určitě zaslouží pozornost.

Letos se můžete těšit v každé z pěti sérií hlavní kategorie na: **5 normálních úloh**, z toho alespoň jedna praktická opendatová, **seriál o zpracování dat** jako 6. úlohu a kuchařku na nějaké zajímavé informatické téma hodící se k úlohám dané série.

Do celkového bodového hodnocení se z každé série stále **započítává 5 nejlépe vyřešených úloh** (tedy nemusíte vyřešit úplně všechny a i tak můžete dosáhnout na plný počet bodů). Také se vám body za úlohy **přepočítávají podle vašeho služebního stáří** – na přesnou definici se podívejte do pravidel na našem webu.

Také budeme zveřejňovat autorská řešení hned po skončení série – pokud nás pak při opravování napadnou nějaké komentáře k řešením od vás, tak je zveřejníme 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 (této 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, placku a možná i další překvapení.

Termín série: **21. října 2019 v 8:00**

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

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

Odměna série: Každému, kdo vyřeší 4 úlohy alespoň na polovinu bodů, pošleme **sladkou odměnu**.

První série třicátého druhého ročníku KSP

Hlídková loď se po dvou týdnech letu vyloupla z hyperprostoru a její výkonné senzory začaly zkoumat tuhle zapadlou hvězdnou soustavu, mezitím co jí z cívek motorů postupně vyzařovala zbytková energie přechodu do normálního prostoru. Na palubě neměla moc velkou posádku – pouhých deset členů – ale to by mělo na tuto misi bohatě stačit.

Místní kolonie se před šestnácti dny přestala ozývat, poslední zachycená zpráva hovořila o nějakých technických problémech. Proto taky z Antaraxu, nejbližší velké lidské základny, vyslali Hefaista jako rychlou servisní a hlídkovou loď, aby zjistila, co se děje.

Kapitánka Laren se otočila na svého komunikačního specialistu, jediné nepozemšťana v posádce: „Zaxi, vysílá základna něco alespoň na podsvětelné komunikaci?“ „Zkoumám skippere… moment, něco jsem zachytil… automatická zpráva, ale je nějaká zkomolená, zkusím ji vyfiltrovat.“

32-1-1 Zkomolené vysílání — 9 bodů

Hlídková loď Hefaistos zachytila několik zpráv od lidské základny v místní hvězdné soustavě. Vypadá to jako automatické zprávy, kde každá z nich byla vyslána dvakrát po sobě, ale závadou na vysílači obsahuje nějaký signál navíc.

Přesněji řečeno základna chtěla vyslat zprávu X (sestávají se jen z velkých písmen anglické abecedy), a to tak, že ji vyslala dvakrát zopakovanou za sebou, ale do tohoto řetězce se na náhodnou pozici vmísilo jedno písmenko navíc. Příkladem může být původní zpráva `ABC` a vysílání `ABCAWBC`.

Vaším úkolem bude samozřejmě co nejrychleji získat původní zprávu X, případně rozhodnout, že to nelze, či že to není jednoznačné.

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 Z udávající počet zachycených zpráv. Pak bude následovat Z řádků, kde na každém z nich bude vždy uvedené nejprve číslo K udávající délku zachycené zprávy, pak mezera a poté samotná zpráva sestávající se z K velkých písmen anglické abecedy.

Formát výstupu: Na výstup vypište pro každou zachycenou zprávu (tedy celkově na Z řádků) text původní zprávy, případně text `[chyba]`, pokud se zpráva nedá zrekonstruovat, nebo `[neunikatni]`, pokud existuje více možných řešení.

Ukázkový vstup:	*Ukázkový výstup:*
`3`	`ABC`
`7 ABCAWBC`	`[chyba]`
`6 UVWXYZ`	`[neunikatni]`
`9 ABABABABA`	

„Mám to!“ zvolal náhle Zax a celá čtyřčlenná posádka můstku se mu nahrnula za konzoli, na které svítil nápis:

`SELHANI STITU BEHEM IONTOVE BOURE, DUVOD NEZNAMY. VETSINA KOLONISTU UKRYTA VE SKLADU 3. ZADAME O POMOC.`

Hefaistos byl na misi vybrán, protože jako jediná z lodí u Antaraxu byl uzpůsoben pro přistání na planetách bez jakékoliv pozemní podpory a navigace. Nebylo tedy o čem přemýšlet a kapitánka vydala rozkaz k navedení lodě na nejkratší přibližovací kurz k planetě a k přistání bez předchozího přechodu na orbitu.

Hefaistos se řítil k planetě akcelerovaný svým iontovým motorem, něco za polovinou své cesty se obrátil motory vzad a začal stejnou silou decelerovat tak, aby do atmosféry planety vstoupil přesně vypočtenou vstupní rychlostí.

Ve správný moment se zasunula většina vnějších aparatur na trupu, okna a trysky iontových motorů zakryly rozměrné tepelné štíty a manévrovací trysky otočily loď břichem dolů, aby začala brzdit třením o atmosféru. Byla to drsná jízda, ale na takovéto zacházení byla devadesátimetrová hlídková loď konstruovaná.

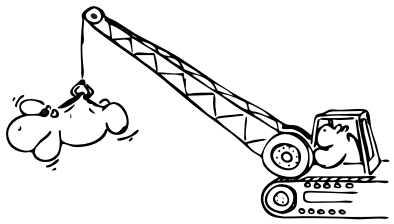
Během několika minut klesla rychlost natolik, aby mohly být tepelné štíty opět zataženy a pomocí atmosférických motorů zbrzděn zbylý pohyb. Pilot zručně zakroužil s lodí okolo základny a skrz velká panoramatická okna můstku se naskytl pohled na několik budov – byly vidět větší škody od iontové bouře, ale většina budov stále stála.

Hefaistos se pomalu snesl na plochu vedle jedné z větších budov. Přistávací vzpěry se vysunuly ze stále rozpáleného trupu a jejich masivní tlumiče zbrzdily dosednutí devítiset tun váhy lodě.

Vzduch na planetě byl podobný pozemskému, jen s velkou koncentrací pro člověka jedovatých sloučenin a s roční průměrnou teplotou okolo nuly. Výsadek s kapitánkou v čele se tak mohl vydat prozkoumat základnu jen v teplém oblečení a s dýchacími maskami – těžké nepohodlné skafandry tentokrát nebyly potřeba.

U vstupu do hangáru skladu číslo tři zjistili, že základna je bez hlavního zdroje energie. Bez něj se jim nepovede odemknout vnitřní dveře hangáru, aby se dostali dál dovnitř. Sklad tři měl sice nahoře u stropu záložní zdroj energie, ale očividně již vyčerpal své palivové články a nikdo asi nepřemýšlel, jak do něj dopravit nové palivové články, když byl hangárový jeřáb bez energie.

Kompatibilních palivových článků měl naštěstí Hefaistos jako servisní loď dost a dokonce vezl na palubě i malý pásový manipulátor. Výsadek se tedy rozhodl postavit si z různých krabic, palet a kontejnerů ve skladu jakousi rampu, po které by zvládli dopravit palivové články až nahoru.



32-1-2 Stavba rampy — 12 bodů

Technici by potřebovali postavit rampu z několika kusů krabic, které se nalézají vyskládané podél jedné ze stěn hangáru. Mají k tomu k dispozici pásový manipulátor, kterým vždycky mohou nějaké dvě krabice vysunout z řady a prohodit je. Ale čím vyšší a těžší krabice (váha krabice je přímo úměrná její výšce), tím náročnější je s ní manipulace a tím více energie na to manipulátor spotřebuje.

A jak má vypadat taková rampa, kterou chtějí postavit? Podobně jako schody – musí to být řada krabic, která na jedné straně bude začínat tou nejnižší z nich a na té druhé bude končit tou nejvyšší z nich.

Formálněji řečeno dostanete na vstupu zadané krabice tak, jak na počátku stojí v řadě, každá krabice bude zadaná svou výškou. Vaším úkolem bude krabice seřadit od nejmenší po největší tak, že jedinou povolenou operací bude prohození dvou libovolných krabic. Toto prohození bude stát tolik energie, jaký je součet výšek obou přesouvaných krabic. Vymyslete algoritmus, který najde postup, jak krabice prohazovat, aby na konci byly správně seřazené a celková potřebná energie byla co nejmenší.

Příklad: Mějme krabice výšek $1, 4, 2, 3, 5$. Ty umíme seřadit se spotřebováním 11 jednotek energie: Nejprve prohodíme krabice 2 a 3 (to nás stojí 5 jednotek energie) a pak prohodíme krabice 4 a 2 (za 6 jednotek energie).

Upozornění: Důležitou součástí tohoto řešení je i důkaz, že váš postup skutečně vrací optimální strategii prohození.

Palivové články zapadly na své místo a hlavní inženýr McCormack, balancuje na vršku improvizované rampy, zatáhl za páku. Záložní zdroj naskočil, ale v hangáru se rozsvítila pouze nouzová světla. Během několika sekund naskočil i řídicí panel vedle přetlakových dveří dál do útrob skladu, avšak jen v nouzovém režimu místního ovládání.

„Zůstaňte zatím tady, já a Drake. . . ,“ kývla kapitánka na jednoho ze dvou mariňáků přidělených k posádce, „. . . se podíváme dovnitř a zkusíme najít kolonisty.“

Za přechodovou komorou následovala dlouhá chodba lemovaná po obou stranách menšími sklady. Na konci chodby se nalézal výtah a schodiště pokračující níž. Podzemní patro vypadalo obdobně, jenom na místě, kde se ve vrchním patře nacházel hangár, byly velké těžké tlakové dveře. Došli až k nim a kapitánka je zkusila otevřít. Ovládání zobrazilo jen sérii chyb a pak se restartovalo, ale vzápětí se z interkomu ozval lidský hlas: „Kdo jste a co tu děláte?“

„Jsem kapitánka Laren La Boy, servisní a hlídková loď Hefaistos, přiletěli jsme z Antaraxu po ztrátě spojení s vámi před šestnácti dny.“ „Tak rychle?“ podivil se hlas. „No, Hefaistos má hyperprostorový pohon čtvrtého stupně, proto taky vyslali nás, byli jsme u Antaraxu nejrychlejší loď.“

To hlasu asi stačilo a dveře se pomalu otevřely. Naší dvojici se naskytl pohled na místnost plnou nepořádku, přikrývek a hromady prázdných kontejnerů nouzových zásob. Mezi tím vším polehávalo odhadem tak třicet lidí.

„Nemáte něco k jídlu?“ přišoural se malý kluk. „Jo tady, berte,“ sáhl Drake do svého batohu a vytáhl několik balení energetických tyčinek s čokoládou, které tam měl v očekávání něčeho podobného. Jenom trochu podcenil počet.

Kapitánka si šla promluvit s ředitelem kolonie stranou a Drake mezitím pozoroval, jak se kolonisté pustili do konzumace tyčinek – vycházela tak jedna tyčinka na dva kolonisty. . .

32-1-3 Čokoládová tyčka — 11 bodů

Dva hladoví kolonisté mají dohromady jednu čokoládovou tyčku a každý by z ní chtěl sníst co možná nejvíce. Tyčka je rozdělená na různě velké dílky a každý dílek lze ohodnotit nějakou jeho energetickou hodnotou. Oba kolonisté se postupně střídají v ulamování dílků, ale vždy mohou ulomit jenom jeden krajní dílek (takže, s výjimkou posledního tahu, mají vždy právě dva dílky na výběr).

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];
```

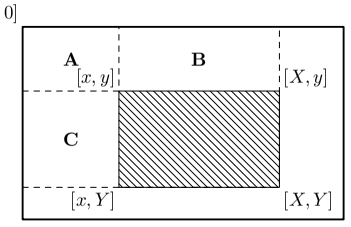
To nám umožňuje snížit čas potřebný na řešení této úlohy na $\mathcal{O}(n^2)$. To je už lepší čas; prozradíme však, že tuto úlohu lze řešit dokonce v lineárním čase, ale to je již nad rámec této kuchařky.

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 končících 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).

```
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.

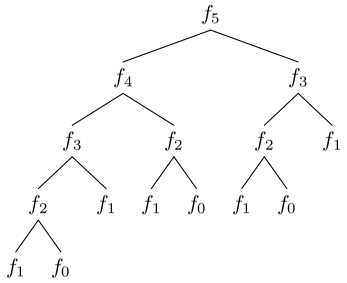
Více se o metodě Rozděl a panuj můžete dozvědět ve stejnojmenné kuchařce.[16]

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.[17]

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 $\mathcal{O}(n^3)$.

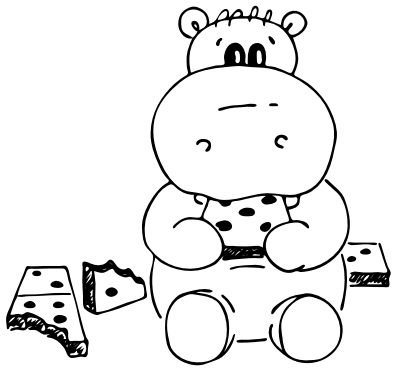
To není pro takhle jednoduchou úlohu zrovna ten nejpěknější čas, zkusme ho zlepšit. Ukážeme si, jak vypočítat sou-

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

Tyčinka je zadaná jako seznam energetických hodnot jednotlivých dílků. Spočítejte pro zadanou tyčinku s D dílky strategii pro prvního kolonistu tak, aby získal dílky dohromady s co největší energetickou hodnotou. Předpokládejte, že druhý kolonista bude vždy hrát optimálně.

Váš algoritmus počítající strategii by zároveň měl doběhnout v rozumně krátkém čase (exponenciální čas vzhledem k D už je určitě příliš pomalý).

Příklad: Pro tyčinku s hodnotami $5, 15, 3, 1$ je pro prvního kolonistu nejlepší vzít v prvním tahu dílek s hodnotou 1, i když by mohl vzít i dílek s hodnotou 5 – pokud vezme při prvním tahu dílek s hodnotou 1, tak získá celkově 16, pokud by ale v prvním tahu vzal dílek s hodnotou 5, tak při optimálních tazích druhého kolonisty by odešel nejvýše s dílky v hodnotě 8.



Během následující hodiny distribuovali mezi kolonisty několik beden nouzových zásob a mezitím se dozvěděli o událostech, které se zde staly – vše začalo nebývale silnou iontovou bouří, během které vysadila komunikace. To by ještě nebylo nic divného, jenže pak začaly jeden po druhém selhávat spolu vůbec nesouvisející systémy základny, a nakonec selhal i několikrát zálohovaný magnetický štít. Kolonisté se uchýlili do bezpečí krytu pod skladem číslo tři, ale ze sedmačtyřiceti kolonistů jich třináct během bouře zmizelo neznámo kam. Po týdnu vysadil i záložní zdroj budovy (i když podle jejich odhadů měly články vydržet přinejmenším tři měsíce) a nakonec je objevil až výsadek z Hefaista.

Kapitánka nařídili oběma mariňákům zahájit průzkum celého areálu základny a spolu s hlavním inženýrem a několika techniky z řad kolonistů se vydala k hlavní budově základny. Po připojení energetického vedení z Hefaista sice budova oživla, ale jen v nouzovém režimu – zjistili, že centrální počítačové jádro je vymazané. Naštěstí na základně byla na holografických médiích uskladněna i kopie operačního systému základny. Jenom její instalace chvíli zabere.

32-1-4 Instalace OS — 9 bodů

Operační systém pro hlavní počítač základny se sestává z velkého počtu instalačních balíčků, které jsou nějakým způsobem rozdělené na dvou holografických médiích.

Cílem je instalovat všechny balíčky, ale některé balíčky se nedají instalovat, dokud nejsou nainstalované nějaké jiné (například program pro ovládání dveří potřebuje knihovnu pro ovládání servomotorů). Obecně každý balíček může mít závislost na libovolném (i nulovém) počtu jiných balíčků. Navíc je slíbeno, že závislosti nikdy netvoří cyklus (tedy vždy existuje způsob, jak se dá operační systém nainstalovat).

Máme zadaná čísla k_1 a k_2, která značí počty instalačních balíčků na jednotlivých médiích. Instalační balíčky jsou rozděleny hezky popořadě, tedy na prvním médiu jsou balíčky s čísly $1, 2, \ldots, k_1$ a na druhém balíčky s čísly $k_1 + 1$, $k_1 + 2, \ldots, k_1 + k_2$. Celkový počet instalačních balíčků je tedy $k_1 + k_2$. Dále známe Z závislostí, tedy např. že balíček 15 závisí na balíčku 38, a ten tedy musí být nainstalován dříve.

V mechanice pro čtení holografických médií může být v jednu chvíli pouze jedno médium. Z něj můžeme nainstalovat jakékoliv balíčky, které na ničem nezávisí nebo závisí pouze na balíčcích, které už máme instalované. Pak musíme médium vyměnit za druhé, abychom mohli pokračovat v instalaci.

Pro zadané závislosti balíčků a jejich rozdělení na instalační média najděte nejmenší nutný počet prohození instalačních médií, aby se nám povedlo instalovat všechny balíčky.

Příklad: Zaveďme si značení $a \to b$ znamenající, že balíček a závisí na balíčku b (tedy balíček b musí být instalovaný první). Pak pro holografická média s počty balíčků 3 a 3 (tedy média $\{1, 2, 3\}$ a $\{4, 5, 6\}$) a pro závislosti $2 \to 6$, $3 \to 5$ a $5 \to 2$ potřebujeme 4 kroky: Nejdříve vložíme druhé médium a nainstalujeme balík 6, poté vložíme první médium a nainstalujeme balíky 1 a 2, poté opět vložíme druhé médium a nainstalujeme balíky 4 a 5 a nakonec vložíme opět první médium a nainstalujeme poslední balík 3.

Po úmorném procesu instalace se povedlo hlavní počítač základny opět nahodit. Postupně se začaly připojovat i jednotlivé budovy a po další hodině snahy se rozběhl i fúzní reaktor a základna tak přestala být závislá na „pupeční šňůře“ od Hefaista. Poškození základny od bouře bylo značné, ale nic neopravitelného. Nic však nevysvětlovalo to množství poruch, které se vyskytly současně s bouří.

Například hlavní komunikační anténa se prostě propadla do země a zbortila se jako věž ze sirek, generátor magnetického štítu vybuchl i s půlkou budovy a třeba záložní generátor v hlavní budově zmizel úplně.

V troskách budov se povedlo nalézt čtyři oběti bouře, ale po zbylých devíti kolonistech nebylo ani vidu ani slechu. Mariňáci se tedy rozhodli vystoupat do blízkých vrcholků táhnoucích se okolo základny, aby získali větší rozhled.

32-1-5 Výhled z vrcholků — 10 bodů

Dvojice mariňáků vystoupala na jeden vrcholek, aby měla co nejlepší rozhled do krajiny. Současný výhled jim ale nestačí a rádi by vystoupali na nějaký vyšší vrcholek – nechtějí však přitom sejít moc nízko.

Můžeme si představit mapu vrcholků jako čtvercovou síť obsahující na každém políčku výšku v metrech. Mezi sousedními políčky se můžeme pohybovat do všech 8 směrů. Máme označené políčko, na kterém jsme, a chceme si naplánovat cestu na nějaké políčko s větší výškou.

Během cesty budeme muset asi projít přes nějaká další políčka, ale chceme, aby minimální výška, ve které se vyskytneme, byla co možná nejvyšší (neboli chceme maximalizovat minimální výšku, ve které se cestou ocitneme). Pokud takových vyšších míst se stejnou minimální výškou během cesty existuje víc, chtěli bychom najít to nejvyšší možné.

Vymyslete algoritmus, který něčeho takového dosáhne.

Příklad: Mějme výšky vrcholků jako v tabulce níže. Nechť stojíme na vrcholku s výškou 5. Na vrcholek 9 bychom se uměli dostat tak, že bychom naklesali do výšky 2, ale na vrcholky 7 a 8 se umíme dostat tak, že naklesáme jenom do výšky 4 a přejdeme po hřebeni. Žádná lepší možnost tu neexistuje, takže naše zadání splňuje nejlépe vrcholek s výškou 8, do kterého cestou projdeme přes nejnižší výšku 4. Jedna z možných cest je vyznačena.

```
2 2 1 2 1
2[5]3 7 2
2 1[4]4 1
9 1 1[8]1
```

„Skippere, tady Drake… našli jsme něco zajímavého. Směrem na jih od základny jsou v písku vidět stopy terénního pásového transportéru. Nejsou vůbec zafoukané bouří, takže budou maximálně čtyři dny staré.“

Kapitánka Laren se podívala tázavě na ředitele základny. „My jsme to určitě nebyli, od výpadku nouzového zdroje jsme byli všichni uvězněni v tom krytu pod skladem číslo tři,“ odpověděl ředitel.

Kapitánka se zamyslela a pak vydala rozkaz…

* * *

Letos experimentujeme s příběhem k úlohám, který můžete sami ovlivnit. Co by podle vás měla kapitánka Laren La Boy udělat? Zahlasujte do ***7. října*** *v anketě.*[1]

První díl příběhu pro vás sepsal

Jirka Setnička

32-1-6 Data na OSMou — 16 bodů

Pokud vám připadá už trochu nuda, že v KSP máme jenom úlohy s umělými daty bez nějakého reálného významu, tak jsme právě pro vás připravili tento seriál. Letošní seriál se totiž bude pokoušet ukázat vám práci s různými zajímavými zdroji velkých dat. Zaměříme se taky na různé nástrahy – co když se vám data nevejdou do paměti, co když se vám odzipovaná data nevejdou ani na disk, jaké je to spustit si program přes noc, abyste ráno zjistili, že spočítal výsledek, ale spadl na posledním řádku, … a podobně.

Opravdové mapy

V prvním dílu se naučíme pracovat s mapami z projektu OpenStreetMap, na což navážeme ještě v dílu druhém. V dalších dílech se pak podíváme na nějaké další zajímavé zdroje dat.

Projekt OpenStreetMap[2] je taková mapová Wikipedie – je to online mapa podobná mapám od Google a nebo Seznamu, akorát jí může editovat každý. Reálně je (narozdíl od Wikipedie) značná část dat převzatá z jiných zdrojů, ale pořád jsou volně dostupné a můžete tam zanést své oblíbené zkratky přes les :)

OSM můžeme využívat jako běžný uživatel a hledat si na `https://osm.org/` třeba kde bude další soustředění. Ale narozdíl od Google nebo Seznam map si můžeme stáhnout i strojově zpracovatelná zdrojová data a něco si nad nimi spočítat.

To je přesně to, co teď uděláme, protože si přeci chceme něco naprogramovat. Budeme používat data ve formátu XML, která se dají z OSM získat. Tady se pokusíme vysvětlit strukturu a smysl dat a ukázat nějaké obecné principy. Detailnější technické informace o tom, kde si data stáhnout a jak je načítat, se pak můžete dočíst na webové stránce seriálu.[3]

Formát dat

Pokud očekáváte, že se nám popis vstupního formátu pro úlohu vejde na jeden odstavec, tak je na čase si odvyknout. V praxi tomu tak moc často nebude a OSM data nejsou výjimkou.

Nicméně nebude to nic hrozně komplikovaného. V principu datový soubor obsahuje jen tři typu objektů: *vrcholy*, *cesty* a *relace*.

1. Vrcholy

Vrcholy (`node`) jsou nejzákladnější objekty v OSM a jenom ony nesou informace o poloze. Vrchol je zkrátka nějaký bod na Zemi zadaný pomocí zeměpisných souřadnic (šířka a délka), ze kterého můžeme konstruovat jiné složitější objekty, například cesty.

Nejdůležitější vlastnosti vrcholu jsou poloha a identifikátor. Polohu najdete jako souřadnice v atributech `lat` a `lon`, identifikátor je číslo v atributu `id`. Upozorňujeme, že vrcholů je na světě přes 5 miliard, takže nestačí 32-bitové celé číslo. Polohu je pro dobrou přesnost také lepší ukládat do 64-bitového desetinného čísla.

Ale vrcholy nemusíme používat jenom pro stavbu složitějších objektů – i on samotný může být něčím významný. Proto můžou mít vrcholy *tagy* – další informace, které se k nim vážou. Tag je dvojice klíče a hodnoty a může v něm být skoro cokoliv.

Tagy tu opravdu nemáme prostor vysvětlit všechny. Proto ty, kterým váš program nebude rozumět, můžete tiše ignorovat. Jejich význam ale bývá poměrně intuitivní, uvedeme několik příkladů:

- Tag `name` je jméno objektu (třeba jméno vesnice nebo jméno restaurace).
- Tag `source` značí, odkud se data vzala.
- Skupina tagů `addr:street`, `addr:postcode`, `addr:city` a `addr:housenumber` nám říká adresu bodu.

Ukázka reálného bodu z OSM:

```
<node id="2578544869" visible="true" version="4"
 changeset="71085651"
 timestamp="2019-06-10T03:14:47Z"
 user="itamas80" uid="1719518"
 lat="50.0435083" lon="14.4031207">
    <tag k="bus" v="yes"/>
    <tag k="highway" v="bus_stop"/>
    <tag k="name" v="Pod Žvahovem"/>
    <tag k="name:de" v="Schwahower Grund"/>
    <tag k="public_transport" v="platform"/>
</node>
```

2. Cesty

Cesty (`way`) jsou skupiny bodů. Dalo by říci, že vlastně tvoří hrany grafu, ale v OSM se cesty používají třeba i na vyznačení ploch, vodních toků nebo hranic obcí (pak často mluvíme o *polygonech* namísto o cestách). Cesta tedy není vždy něco, po čem se dá chodit, ale něco co se skládá

[1] `https://poll.ly/#/PvykX7ka`

[2] `https://www.openstreetmap.org/`

[3] `http://ksp.mff.cuni.cz/viz/serial-osm`


```
        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 $\mathcal{O}(\log n)$.[15]

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;
```

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

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--;
        }
```


z více bodů. Body jsou naše známé vrcholy, cesta se na ně odkazuje pomocí elementu `<nd>`.

Cesty někdy značí plochy a v takovém případě začínají a končí stejným bodem.

Jak ale poznáme reálný význam cesty? Podobně jako vrcholy můžou mít cesty libovolné tagy. Například tag `name` bude obsahovat pravděpodobně jméno ulice a tag `highway` nám již svou přítomností říká, že se jedná o nějakou cestu, a jako svoji hodnotu bude mít typ této cesty (tedy jestli je to pěšina nebo dálnice – třeba `highway=footway` je chodník pro pěší a `highway=residential` je normální ulice vedoucí k domkům).

Ukázka cesty tvořící uzavřený polygon (v tomto případě nějakou garáž):

```
<way id="59584456" visible="true" version="2"
 changeset="38311484"
 timestamp="2016-04-05T08:39:44Z"
 user="JandaM" uid="2169558">
    <nd ref="739135731"/>
    <nd ref="739135841"/>
    <nd ref="739135871"/>
    <nd ref="739135899"/>
    <nd ref="739135731"/>
    <tag k="building" v="garage"/>
    <tag k="building:ruian:type" v="18"/>
    <tag k="ref:ruian:building" v="49847236"/>
    <tag k="source" v="cuzk:ruian"/>
</way>
```

3. Relace

Posledním typem dat v OSM je relace (`relation`). Relace se nevyužívají tak moc jako cesty a vrcholy, ale často také nesou užitečné informace. Relace je vlastně skupina jiných objektů – cest, vrcholů a nebo dalších relací. Například když se podíváme na tramvajovou trať, můžeme si všimnout, že koleje jsou v relacích podle toho, jaké linky na nich pravidelně jezdí. Existuje tedy relace pro pravidelné linky hromadné dopravy. Drobný problém je, že se linky mění docela často a v mapách jsou tak často neaktuální nebo nepřesné informace.

Další použití relací jsou velké oblasti, které jsou složeny z mnoha cest – těm říkáme *multipolygony*. Tímto způsobem najdete například zakódované hranice obcí nebo městských částí, některé lesy a vodní plochy. Hlavní výhodou oproti cestě je, že oblasti můžou být tvořeny několika mnohoúhelníky, případně v sobě můžou mít díry – vodní plochy v sobě můžou mít ostrovy, politické oblasti nejsou vždy souvislé.

Detailněji si multipolygony můžete nastudovat na OSM Wiki.[4] Aby to nebylo tak komplikované, tak se budeme zabývat jen těmi relacemi, které jsou souvislé a nemají díry. Pak stačí z relace vytáhnout všechny cesty, uspořádat je do „kruhu“ a máme mnohoúhelník. Pozor na to, že typicky nebude konvexní a může mít všechny možné divné tvary.

* * *

Význam všech tagů používaných v mapách si můžete najít na OSM Wiki, například si lze najít všechny možné hodnoty pro tag `highway`.[5]

Pokud si chcete prohlédnout kus mapy trochu vizuálněji, než prohlížením XML souboru v textovém editoru, můžete tak učinit na webu `https://osm.org/`: Přibližte si kus mapy, který vás zajímá, a nahoře se přepněte do „Edit → Edit with iD“. Tím se dostanete do editoru mapy, ve kterém se dají rozkliknout všechny objekty. Když kliknete na cestu nebo vrchol, v levém panelu se dole zobrazí seznam všech tagů.

Úlohy obecně

Úlohy budeme počítat nad několik různými výřezy z OSM. Data z OSM se dají volně stáhnout v různých formátech, nejčastěji jako gzipovaný XML soubor s příponou `.osm.gz`, bzipovaný XML soubor s příponou `.osm.bz2` nebo jako PBF binární soubor s příponou `.pbf`. Servery, ze kterých se dají stáhnout, i nějaké detaily všech formátů a jejich parsování jsme shrnuli na webové stránce seriálu.[6]

My v úlohách budeme pracovat s gzipovanými XML soubory, protože práce s XML nám přišla pro účely seriálu výhodnější a protože knihovny pro gzip jsou v programovacích jazycích na lepší úrovni, než knihovny pro práci s bzip2.

Normálně byste si při práci s OSM asi stáhli nějaký poslední týdenní export z nějakého z veřejných serverů, ale abychom si byli jistí, že pracujeme nad stejnými daty (a že zrovna třeba cesta s nějakým ID náhodou nezmizela), tak jsme pro vás vystavili na našem webu konkrétní výřezy tří oblastí:

- Evropa – velká testovací data, asi 43 GB zkomprimovaného XML (392 GB v nekomprimované verzi).
- Brno – malá testovací data, asi 20 MB zkomprimovaného XML (186 MB v nekomprimované verzi).
- Hrochův Týnec – velmi malá data pro ověření vašeho řešení, prozradíme vám u nich výstup našeho řešení.

Doporučujeme si nejdříve řešení otestovat na menším vstupu, protože nad celou Evropu to nedoběhne rychle. Také doporučujeme nenechávat řešení na poslední neděli, protože by některá řešení nemusela doběhnout (naše odladěné řešení třetího úkolu pro celou Evropu běží přes 9 hodin).

Každý úkol od vás budeme chtít vyřešit na malých (Brno) i velkých (Evropa) datech. Za vyřešení **na malých i velkých datech** získáte za úkol **plný počet bodů**, za vyřešení **jenom na malých datech** získáte **polovinu bodů**.

A co nám máte vlastně odevzdat jako řešení? Odevzdávete prosím zip archiv, který bude obsahovat:

- Krátký slovní popis, jakým způsobem jste přistupovali ke které úloze a třeba jak vám to přišlo těžké.
- Výsledek pro každou úlohu pro malá i velká data (pokud je to jedno číslo, tak se dá klidně napsat do slovního popisu; jestli je to seznam IDček nalezených objektů, tak raději do samostatného textového souboru; obrázek jako obrázek).
- Zdrojový kód programu, který jste použili výpočet pro řešení.
- Popis jak dlouho trvalo úlohy spočítat a na jak výkonném hardware (typ, počet jader a frekvence procesoru, množství paměti) – čas se dá změřit třeba vypsáním času na začátku a na konci programu, případně na Linuxu třeba příkazem `time`.

[4] `https://wiki.openstreetmap.org/wiki/Relation:multipolygon`
[5] `https://wiki.openstreetmap.org/wiki/Key:highway`
[6] `http://ksp.mff.cuni.cz/viz/serial-osm`

A nyní už k samotným úkolům a k několika principům velkých dat okolo nich...

Streamové parsování

Občas se nám načtená data nevejdou do paměti (třeba v případě našeho velkého výřezu Evropy – pochybujeme, že někdo z vás má v roce 2019 doma stroj s 390 GB operační paměti na nekomprimovaná data... a ani 45 GB ve zkomprimované variantě asi většina z vás do paměti také nenacpe). To při zpracování velkých dat není vůbec nevšední věc a potkáme se s ní docela často.

Jak si s tím poradit? Nejlepší bude skrz data jenom projít a cestou zpracovat těch několik věcí, které nás zajímají. Takovému způsobu procházení se říká streamové a třeba na spočítání počtu vrcholů v celé mapě nepotřebujeme skoro žádnou paměť – do paměti vždy načteme jenom jeden XML tag, podíváme se, jestli je to `node`, a pokud ano, tak přičteme k nějakému počítadlu plus jedna.

Nejlepší je, že se ve většině programovacích jazyků dá lehce vrstvit několik streamových zpracování za sebou. To využijeme právě u našich dat, kde za sebe budeme potřebovat navrstvit streamový „odzipovávač“ a streamové parsovátko XMLka. Pak můžeme našemu programu předhodit přímo `.osm.gz` soubor a když v programu zavoláme funkci na naparsování dalšího XML tagu, tak to může pod kapotou vypadat třeba takto:

- Zavoláme funkci `getNextTag()`
- Funkce `getNextTag()` má u sebe buffer, do kterého postupně plní byty. Dokud buffer neobsahuje celý validní XML tag, tak v cyklu volá funkci `gzip.getNextByte()` a postupně připisuje do bufferu další a další znaky. Ve chvíli, kdy je tag načtený celý, tak ho vrátí a odmaže si ho z bufferu.
- Funkce `gzip.getNextByte()` má u sebe buffer odzipovaných dat, které vrací po jednotlivých bytech. Ve chvíli, kdy je buffer prázdný, tak načte ze souboru na disku další chunk, který může „odzipovat“ a opět si tím naplnit buffer.

Ukázka tohoto postupu je v ukázkových programech na stránce s technickými detaily na webu.

Úkol 1 [3b]:

Pojďme si nabyté znalosti zkusit na něčem jednoduchém. Zajímalo by nás, kolik je na mapě unikátních názvů ulic.

Přesněji řečeno, najděte si v Brně a nebo v Evropě všechny cesty, které mají (neprázdné) tagy `highway` a `name`. Pak spočítejte, kolik unikátních hodnot `name` se tam vyskytuje (názvy lišící se jenom ve velikosti písmen nechť jsou pro naše účely různé názvy). Toto číslo je řešením úkolu.

Pro Hrochův Týnec naše řešení našlo 16 různých názvů ulic (včetně názvu „lávka (zákaz vstupu)“, což je sice chybný název podle OSM – zákaz vstupu by se měl vyjadřovat jinak – ale v datech je, a tak ho počítáme).

Možná si říkáte, jestli bychom nemohli spočítat, kolik je na mapě ulic (včetně těch duplicitních názvů). Samozřejmě mohli, ale není to tak jednoduché – ulice totiž často nejsou jedna cesta, ale je to několik cest se stejným jménem, které jsou akorát spojené v nějakých vrcholech. Někdy jsou ulice rozdělené na více cest bez závažného důvodu, ale někdy se dokonce větví a cyklí a pak už by jednou cestou reprezentovat ani nešly. Proto je potřeba před počítáním nejdříve seskupit rozštěpené ulice a to není tak jednoduché. Ale rozhodně to není nemožné, tak to klidně zkuste :)

Více průchodů

Občas nám jeden průchod daty nestačí. Typickým příkladem v OSM je, když potřebujete získat vrcholy cest s nějakým tagem. Dokud ale nemáte nalezené ony cesty, ani nevíte, které vrcholy vás budou zajímat. A pamatovat si všechny vrcholy v paměti také nelze. Jak si poradíme s tímto?

Nejjednodušší přístup je asi zdrojový soubor projít vícekrát. Při prvním průchodu si poznamenáme IDčka všech objektů, které by nás zajímaly (mohou to být právě vrcholy nějaké cesty), a pak se vrátíme na začátek souboru. Této operaci se většinou říká *seek*, ale ne všechny vrstvy našeho streamového zpracování musí tuhle operaci podporovat (typicky archivační vrstvy mají se seekem problémy kvůli svému vnitřnímu stavu). Ale postup zavřít původní soubor, otevřít ho a znovu ho obalit gzip streamerem a XML parserem bude fungovat asi ve všech jazycích.

Při druhém průchodu už poté při na načítání objektů můžeme kontrolovat, jestli jsou to pro nás zajímavé objekty a kdyžtak si je uložit do paměti (když těch zajímavých objektů není miliarda) nebo je nějak jinak zpracovat.

Vizualizace

Občas nejlepší způsob, jak se podívat na mapová data, není číst hromadu čísel ze standardního výstupu, ale nakreslit si nějaký obrázek. Ostatně když se díváte třeba na OSM na webu jako uživatel, tak právě vidíte nějaké vyrenderované výsledky. To už je ale docela komplexní úloha a jak vykreslit mapu, aby byla dobře použitelná, je už více otázka kartografická, než programátorská.

Ale i nějaké mnohem jednodušší vizualizace nám mohou hodně pomoci při zkoumání nějakého aspektu – mnohdy jeden obrázek vydá za tisíce slov. Pojďme si zkusit nakreslit třeba nějakou *heatmapu*.

Heatmapa je obecně znázornění velikosti nějakého jevu v nějaké oblasti (například množství tepla, odtud i ten název). Nejčastěji se vyrábí tak, že si určíte oblast (pro nás to bude vždy celý datový soubor), ten si rozsekáte na stejně velké čtverečky a pak procházíte data a poznamenáváte si, kolik je ve kterém čtverečku věcí, které vás zajímají. Nakonec každý čtvereček bude prezentovat jeden pixel na výsledném obrázku a čím více věcí v odpovídajícím čtverečku, tím tmavší pixel bude.

Při skládání bodů do pixelů obrázku se pravděpodobně chytnete do drobé pasti – Země není placka a souřadnice bodů v OSM jsou souřadnice na elipsoidu. Když je přepočítáme na rovinu tím naivním způsobem, že s nimi počítáme jako se souřadnicemi v rovině, tak dostaneme nějakým způsobem nepřesnou mapu. Dokud kreslíme jednoduchý diagram pro Českou Republiku, tak je to jedno, ale v rámci celé Evropy už je to docela rozdíl.

Ale pro naše použití se teď s touto primitivní projekcí spokojíme – náš program nebude první, který nebude řešit nějaké komplikované projekce, a aspoň něco by ve výsledném obrázku poznat být mělo i tak. Ale je dobré počítat s tím, že čím severnější oblasti mapa obsahuje, tím je zkreslení touto projekcí horší.

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.

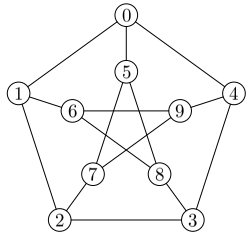
Stromy a grafy v informatice

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.

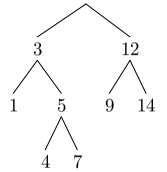
Grafy jsou velmi široké téma. Můžeme hledat jejich minimální kostry, můžeme v nich hledat nejkratší cesty či skrze ně pouštět pod tlakem vodu. Více o nich si tedy můžete přečíst v některé z našich specializovaných grafových kuchařek, které odkazujeme z našeho kuchařkového rozcestníku.[14]

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

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

```
        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

Úkol 2 [5b]:

Co tak si nakreslit hustotu zástavby? Definice „hustoty zástavby“ bude sice trochu sporná, ale pojďme prostě spočítat počet domů na území a vykreslit to jako obrázek.

Jak na to? Dům je typicky cesta s tagem `building`. Občas v mapách sice nějaké domky chybí, nebo je místo nich jenom jeden bod s adresou, ale tak ty prostě nebudeme počítat. Podle toho jak budeme chtít velké rozlišení si mapu rozdělíme na čtverečkovou mřížku a pro každý čtvereček napočítáme počet domů, které jsou uvnitř.

Protože cesta ohraničující dům je tvořena více vrcholy, budeme jako polohu domu pro zjednodušení brát polohu jeho prvního vrcholu. To nám sice může občas dům těsně na hranici jednoho čtverce přesunout do vedlejšího, ale ve větším měřítku se takové chyby vyprůměrují.

Pro Brno vyrobte obrázek široký 1000 pixelů, pro Evropu vyrobte obrázek široký 4000 px, výšku určete proporcionálně. Barevnost si určete sami (škála ani nemusí být lineární, pokud se vám tak heatmapa bude líbit více, jenom nám v případě nějaké netriviální barevnosti přibalte legendu s vysvětlením, co které odstíny znamenají).

Ještě by se vám mohlo hodit vědět, že v Evropě je na naší mapě OSM zhruba 170 milionů domů – takže si spočítejte, kolik informací o každém domě si můžete pamatovat, abyste se ještě vešli do paměti svého počítače.

Heatmapa Hrochova Týnce by nebyla příliš zajímavá a heatmapu Brna máte za úkol spočítat vy, takže na webu se můžete podívat na heatmapu ČR.

Pokud si právě říkáte, že nemáte tušení, jak pomocí svého oblíbeného jazyka vyrobit obrázek, tak nezoufejte, pravděpodobně to bude docela jednoduché. Nejlepší možnost je popsat svůj problém do Googlu a najít na obrázky vhodnou knihovnu (nebo zjistit, že je k dispozici ve standardní knihovně). Pak bude asi jenom potřeba zavolat funkci „vyrob obrázek“ nějaké velikosti a pak mu nastavovat barvy jednotlivých pixelů (a pozor na to, v jakém rohu je počátek souřadnic). Pokud by to nijak rozumně nešlo, tak můžeš zkusil využít textový formát PGM.[7] Sice bude obrázek obludně velký, ale jde otevřít například v GIMPu a pak uložit jako normální PNG.

Trocha geometrie

Občas je dobrý nápad umět vybrat mapy jenom body, které leží v nějakém zajímavém území (v řeči OSM řekněme v *polygonu*), třeba na území nějakého státu (třeba když vyrábíme OSM export tohoto státu).

Jak jsme již napsali výše, dokud jsou oblasti malé, jdou popsat jedinou cestou (kde poslední a první bod jsou stejné). Ale v případě větších oblastí už není praktické mít cestu s milionem bodů, ale spíše chceme říct, že je oblast ohraničená těmito padesáti cestami.

Takovéto větší polygony jsou v OSM popsané relacemi, které právě obsahují ony cesty (a občas i nějaký bod, například administrativní bod té oblasti, když jde třeba o město). Pozor na to, že cesty na sobě po obvodu nemusí navazovat (mohou být náhodně pomíchané), OSM to nijak nezakazuje a je na programech, aby si s tím pak poradily.

V OSM může relace představovat také takzvaný *multipolygon*, který může mít i díry – například rybník s ostrovem. V takovém případě je to relace mnoha cest, kde některé mají roli `outer` (to jsou normální hranice) a `inner` (to jsou „díry“). Ještě se můžete setkat se *superrelacemi*, což jsou relace obsahující relace, které teprve obsahují cesty. Ale oběma těmto složitějším případům se zatím vyhneme.

V momentě, kdy se nám povede načíst polygon (na což může být potřeba více průchodů), už můžeme aplikovat například kuchařku o geometrii,[8] případně článek z Wikipedie[9] a můžeme dělat zajímavé věci.

Úkol 3 [8b]:

Pojďme si zkusit najít pítka v následujících oblastech:

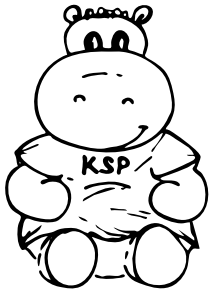
- Centrum Brna: ID 434760
- Alpy: ID 2698607

Obě oblasti jsou „slušné“ polygony určené relací, která obsahuje přímo cesty a neobsahuje žádné díry.

Zdroj pitné vody se pozná tak, že má nastavený tag `amenity=drinking_water` (tedy „toto je pítko“) a nebo má nastaveno `drinking_water=yes` (což může být například i na veřejném WC, kde mají pitnou vodu). Odevzdejte nám seznam pítek v daných oblastech jako seznam identifikátorů vrcholů, kde se můžeme napít, na každém řádku jedno číslo.

A to je prozatím vše. Budeme se těšit na vaše řešení a mezitím si pro vás nachystáme další zajímavé věci o OSM do druhé série :-)

Jirka Setnička & Standa Lukeš



[7] `https://en.wikipedia.org/wiki/Netpbm_format`
[8] `http://ksp.mff.cuni.cz/viz/kucharky/geometrie`
[9] `https://en.wikipedia.org/wiki/Point_in_polygon`

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.[10]

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“.[11]

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], ...`).[12]

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

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

U pole již má smysl přemýšlet, jak dlouho bude která operace trvat. Díky tomu, že jsou jednotlivé prvky v poli naskládané pevně za sebou, 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,[13] nejdříve však doporučujeme dočíst tuto kuchařku.

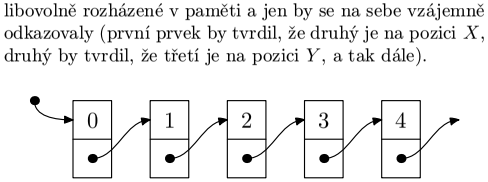
Přidání nového prvku na konec pole také zvládneme v konstantním čase. Problém je přidání nového prvku někam doprostřed (což se nám typicky stane, pokud budeme chtít udržovat hodnoty v poli seřazené a zároveň do něj vkládat nové). V takovém případě se totiž všechny prvky za vkládaným musí posunout o jednu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

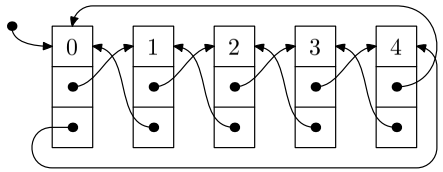


K lepšímu pochopení tohoto principu je důležité si vysvětlit, co to je *ukazatel* (nebo také *odkaz* či anglicky *pointer*). Každé místo v paměti počítače má své číselné označení, kterému říkáme *adresa*. Když si vytváříme nějakou pojmenovanou proměnnou, ta se vlastně odkazuje na určité místo v paměti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

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

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

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

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