- Provedené rozhodnutí již nikdy neodvolává (tedy nebacktrackuje).

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

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

Příklady hladových algoritmů

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

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

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

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

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

Závěr

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

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

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

Úvodním kurzem vaření podle kuchařky vás provedl

Jirka Setnička

matfyz

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

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`.

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/

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 lineárním 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];
```

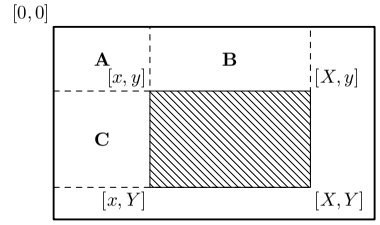
To nám umožňuje snížit čas potřebný na řešení této úlohy na $\mathcal{O}(n^2)$. To je už lepší čas; prozradíme však, že tuto úlohu lze řešit dokonce v lineárním čase, ale to je již nad rámec této kuchařky.

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.

```
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.

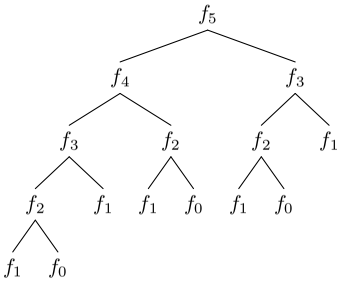
Více se o metodě Rozděl a panuj můžete dozvědět ve stejnojmenné kuchařce.[9]

Předpočítané mezivýsledky

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

Velmi často se totiž setkáváme s tím, že něco počítáme stále dokola. Jako příklad si můžeme 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.[10]

Prefixové součty

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

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

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

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

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

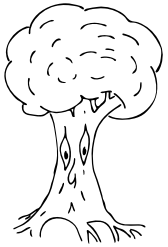
To není pro takhle jednoduchou úlohu zrovna ten nejpěknější čas, zkusme ho zlepšit. Ukážeme si, jak vypočítat sou-

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

Korespondenční Seminář z Programování

31. ročník **KSP** **Září 2018**

Milí řešitelé, řešitelky a řešitelčata!



Nový školní rok právě začal a s ním přinášíme i nový ročník KSP. Leták s úlohami první série právě držíte v ruce.

Pravidelní řešitelé si možná všimnou, že úloh není tolik jako minulé ročníky. Pro tento ročník vám v každé sérii upečeme 6 úloh. Z této šestice bude vždy jedna úloha opendatová a jedna přiřazená k seriálu, jehož téma nás bude provázet celým ročníkem. Další změna se chystá v termínu vydání autorských řešení. Ta budeme letos zveřejňovat současně s termínem série (tedy stejně jako jste zvyklí z kategorie Z).

Do celkového bodového hodnocení se z každé série stále **započítává 5 nejlépe vyřešených úloh.**

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.

Každému řešiteli, který v tomto ročníku z každé série dostane alespoň 5 bodů, darujeme KSP propisku, blok, placku a možná i další překvapení.

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

Termín série: **15. října v 8:00**

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

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

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

První série třicátého prvního ročníku KSP

Žila byla jednou jedna dívenka. Bydlela s maminkou v domečku na kraji vesnice, otec totiž utekl ještě dříve, než se narodila. Její maminka byla švadlena, a protože jednou trochu neodhadla množství látky, spousta červené jí zbyla.

Nevěděla, co si s ní počít. Když přišla móda červených čepečků, mnoho jich vyrobila. Jenomže jak to tak bývá, móda rychle přešla. Lidé je však už měli objednané, tak dceruška musela obcházet sousedství a přesvědčovat je, že si čepečky musí vzít. Od té doby jí nikdo neřekl jinak než Červená karkulka.

Tato práce ji ale strašně nebavila, proto přemýšlela, jak si ji co nejvíce zjednodušit.

31-1-1 Karkulčin byznys — 12 bodů

Karkulka má po vesnici roznést několik červených čepečků. Stojí zde N domů, všechny leží na jedné přímce a jejich pozice známe. Každý dům si objednal několik červených čepečků. Máme zadanou pozici Karkulčina domu, kde Karkulka začíná s přesně tolika čepečky, kolik potřebuje. Může se pohybovat jen doleva a doprava, přičemž každý metr pohybu ji stojí přesně tolik jednotek energie, kolik má ještě nerozdaných čepečků. Když je Karkulka na pozici s domem, může tam vyložit příslušný počet čepečků, jinak čepečky nikam jinam pokládat nemůže. Kudy má jít, aby ji to stálo co nejméně energie a nebolely ji pak nožičky?

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 jsou dvě kladná celá čísla: počet domů N a pozice Karkulčina domu (ten mezi oněch N domů nepočítáme). Následuje N řádků popisujících jednotlivé domy. Na každém z nich jsou opět dvě kladná celá čísla: tentokráte pozice domu a počet objednaných čepečků. Máte zaručeno, že žádné dva domy se nenacházejí na stejné pozici a že domy jsou na vstupu seřazené vzestupně podle pozic.

Formát výstupu: Vypište jediné číslo: minimální možný počet jednotek energie, kterou musí Karkulka na roznesení čepečků vynaložit. Pozor, na reprezentaci výsledku můžete potřebovat 64bitová čísla (`long long` v C, `long` v Javě; v Pythonu to není potřeba řešit).

Ukázkový vstup:	*Ukázkový výstup:*

`3 6`
`1 3`
`7 5`
`20 5`
	`156`

Karkulka začíná na pozici 6 s $3+5+5=13$ čepečky. V optimálním řešení nejprve Karkulka navštíví dům na pozici 7, což ji stojí $|6-7|\cdot 13 = 13$ jednotek energie, poté navštíví dům na pozici 1, což ji stojí $|7-1|\cdot 8 = 48$ jednotek energie, a nakonec donese zbylé čepečky do domu na pozici 20, což ji stojí $|20-1|\cdot 5 = 95$ jednotek energie.

Jednoho dne řekla maminka Karkulce: „Karkulko, babička má dneska svátek. Běž ji navštívit a přines jí bábovku a láhev vína. Ale jdi přímo po cestě a nikde se nezdržuj,

mohlo by se ti něco zlého přihodit.“

Karkulka tedy šla. Cesta vedla přes hluboký, temný les. Vtom kde se vzal, tu se vzal, stojí před ní obrovský šedý vlk. „Kam jdeš, Karkulko?“ zeptal se. „K babičce,“ odpověděla Karkulka, „má dnes svátek a já jí nesu bábovku a láhev vína.“ „A kde babička bydlí?“ vyptával se vlk. „Támhle za lesem,“ odpověděla důvěřivá Karkulka a prozradila vlkovi i adresu.

Vlk měl zrovna obrovský hlad, a tak vymyslel na Karkulku léčku. „Karkulko,“ povídá vlk, „nechceš natrhat babičce květiny? Určitě bude moc ráda!“

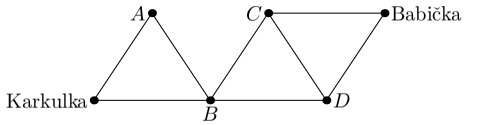
31-1-2 Skoro nejkratší cesta — 10 bodů

Karkulka chce babičce natrhat květiny, ale vzpomněla si, co jí maminka kladla na srdce. Ráda by ji poslechla, ale květiny vedle cesty vypadají tak nádherně! Rozhodla se tedy, že si prodlouží cestu jen o kousek.

Les je propletený pěšinkami a křižovatkami. Protože Karkulce strašně dlouho trvá, než se zorientuje, kam z křižovatky vyrazit, dává si pozor hlavně na to, kolika křižovatkami projde. Zároveň chce ale sbírat květiny. Půjde tedy cestou, která má právě jednu křižovatku navíc oproti nejkratší cestě. Pomůžete Karkulce spočítat, kolika různými cestami se může vydat?

Pokud již znáte grafy (třeba díky tomu, že jste si přečetli přiloženou kuchařku), můžete tento hvozd chápat jako neorientovaný neohodnocený graf. V jednom jeho vrcholu stojí Karkulka a v jiném se nachází babiččin dům.

V příkladě na následujícím obrázku jedna z nejkratších cest vede přes křižovatky B a C. Karkulka může jít cestami ABC, ABD, BCD nebo BDC, celkem má tedy na výběr ze čtyř různých cest.



Jen co se Karkulka vydala trhat květiny, vlk už pádil k babiččině domu. „Kdo tam?“ zeptala se babička, když vlk zaklepal. „To jsem já, Karkulka,“ odpověděl sladkým hláskem vlk. Babička otevřela dveře a chramst, vlk ji hned spolknul.

Karkulce netrvalo trhání květin dlouho a brzy také přišla k babiččině domečku. Vlk nechal otevřené dveře, a tak vešla rovnou do kuchyně. S tím však vlk nepočítal. Ještě si nestačil obléknout babiččinu košili a nasadit babiččiny brýle, a tak zavolal: „Karkulko, to jsem ráda, že jsi tu! Zrovna se mi pomíchalo koření na poličce a potřebuji, abys je seřadila, ještě než půjdeš za mnou do ložnice.“

Vlk uměl dokonale napodobit babiččin hlas, a tak si Karkulka ničeho podezřelého nevšimla. Byla zvyklá babičku poslouchat, a hned se tedy do úkolu pustila.

31-1-3 Řazení kořenek — 10 bodů

Babička má lahvičky s kořením naskládané vedle sebe na jedné dlouhé poličce. Stojí tam bez ladu a skladu, jak to babičce zrovna přišlo pod ruku, a Karkulka je chce seřadit podle abecedy. Může vždy vzít jednu lahvičku a posunout ji o několik míst doleva nebo doprava, a zařadit ji tak na libovolné jiné místo na poličce. Protože se bojí, aby nějakou lahvičku nerozbila, chce je setřídit na co nejméně přesunů. Poradíte jí, jak to má udělat?

Pro zjednodušení se nemusíte zaobírat řazením slov – můžete předpokládat, že každá lahvička má na sobě napsanou pozici, na které se bude vyskytovat po seřazení.

Například se na poličce mohou vyskytovat kořenky v pořadí 2, 3, 5, 4, 1, 6, 7 (rovnou označujeme kořenky jejich pořadím v abecedě). Jedním ze správných řešení je pak vzít lahvičku 1 a dát ji na začátek a poté lahvičku 5 a přesunout ji o jedno místo doprava. Lze si snadno všimnout, že přesunutím jedné kořenky se posloupnost setřídit nedá.

Jakmile Karkulka koření seřadila, vešla k babičce do ložnice. Babička ležela na posteli, čepec na hlavě, brýle na nose. Karkulce se však něco nezdálo. „Babičko,“ povídá Karkulka, „co to tu tak smrdí?“

To vlk nečekal. Myslel, že se Karkulka třeba zeptá, proč má tak velké oči nebo zuby, ale takhle otázka ho hluboce urazila. Přesto se však vzpamatoval, pohotově vyskočil a Karkulku sežral. Svalil se zpátky na postel a hned usnul.

Náhodou šel kolem babiččina domu zrovna jeden ze zdejších myslivců.

31-1-4 Myslivci — 8 bodů

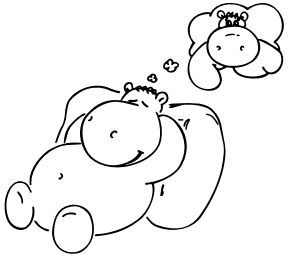
V babiččině vesnici stojí N hájoven a v každé bydlí jeden myslivec. Také se zde nachází M domků, ve kterých bydlí ostatní obyvatelé. Všechny hájovny i domy leží na jedné přímce, jejich polohu máme zadanou. Každý den se každý myslivec vydá na návštěvu do jednoho domu a večer se vrátí zpátky do své hájovny. Myslivci se vzájemně nenavštěvují. Kolik nejméně se každý myslivec nachodí, než navštíví všechny domy? Jinými slovy, pro každou z N hájoven chceme zjistit součet vzdáleností z této hájovny ke všem domkům.

Například se hájovny mohou nacházet na pozicích 3 a 6 a ostatní domky na pozicích 8, 1 a 4. Součet vzdáleností od první hájovny je pak 8 a od druhé 9.

Myslivec vešel do pokoje, uviděl v posteli vlka a hned mu bylo vše jasné. Nebylo to poprvé, co se tohle stalo, zkrátka všechna děvčátka z vedlejší vesnice, která jdou sama hlubokým lesem, se zapovídají s vlkem a takhle to skončí. Myslivec rozpáral vlkovi břicho a nejdřív vyskočila Karkulka a hned za ní i babička, obě dvě šťastné, že zbytek života nemusí strávit ve vlkovi.

Vlkovi do břicha zašili kameny, a když se pak probudil a šel se napít, spadl do studny stejně jako všichni vlci před ním, kteří si chtěli pochutnat na zdejších obyvatelích.

Na oslavu vytáhla Karkulka bábovku, kterou přinesla. Babička si však najednou uvědomila, že během toho zmatku zůstala její zubní protéza nejspíš ve vlkově žaludku, a tak musela bábovku odmítnout. Karkulka si tedy rozdělila bábovku jen s myslivcem – přesně půl na půl.




```
        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 $\mathcal{O}(\log n)$.[8]

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

Ukázka hlavní smyčky v C:

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

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

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.

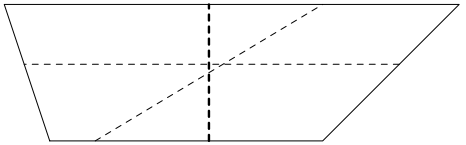
V jazyce C:

```
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--;
        }
```


31-1-5 Krájení bábovky — 13 bodů

Bábovka má tvar kolmého hranolu s konvexní podstavou (tj. takovou, že úsečka nakreslená mezi libovolnými dvěma body podstavy leží celá v podstavě). Karkulka s myslivcem ji chtějí rozpůlit co nejkratším svislým řezem. Kudy má takový řez vést? Pokud existuje více vhodných nejkratších řezů, vyberte si libovolný z nich.

Na následujícím obrázku je uveden příklad bábovky s čtyřúhelníkovou podstavou. Je na něm naznačeno několik řezů (čárkovaně), které rozdělí dort přesně na poloviny. Nejkratší takový možný (tj. ten hledaný) je vyznačen tučně.



Bábovka je snědená, všichni jsou živí a zdraví (až na chudáka vlka), Karkulka se může vrátit domů. Tak tedy... zazvonil zvonec a... však to znáte!

Zuzka Urbanová & Klárka Tauchmanová

31-1-6 Hroznýšovo okno — 15 bodů

Letošní seriál se bude zabývat psaním interaktivních aplikací a obecně programů, které musí reagovat na více vstupů a obsluhovat více výstupů najednou. Naučíme se vyrobit GUI (Graphical User Interface) a obecně psát programy tak, aby na ně uživatel nemusel čekat, když zrovna počítají něco dlouhého.

K takovým účelům samozřejmě existuje spousta různých nástrojů, na GUI například .NET framework od Microsoftu, GTK, Qt nebo WxWidgets, bez GUI třeba asyncio v Pythonu nebo libUCW pro C. My si pro výukové účely vybereme PyQt5, což je binding knihovny Qt5 (jinak určené pro C++) do Pythonu. Předvedené principy však zhusta platí pro ledasjaký jiný nástroj.

Instalace

Pravděpodobně nejvíc standardní metodou, jak si v Pythonu pořídit PyQt5, je Pip. Pokud nevíte, co to je Pip, nebo jej neumíte použít, napsali jsme návod na Pip.[1]

```
$ pip3 install pyqt5
```

Pokud jste linuxáci a preferujete systémový balík, pak například v Debianu hledejte `python3-pyqt5`.

Tím máme vyřešenou instalaci. To bylo snadné.

Hello world

Pojďme si vyzkoušet, jestli to vůbec funguje. Pořídíme si triviální GUI.

```
#!/usr/bin/python3

from PyQt5.QtWidgets import \
    QApplication, QMessageBox
import sys

app = QApplication(sys.argv)
box = QMessageBox(text="ahoj světe")
box.show()
app.exec()
```

[1] `http://ksp.mff.cuni.cz/encyklopedie/python-pip.html`

Program by měl zobrazit okýnko s textem „ahoj světe“ a po jeho odkliknutí skončit. Pokud to udělá, je velmi pravděpodobné, že máte PyQt5 nainstalované a funguje.

A teď co vlastně program dělá:

- `from PyQt5.QtWidgets import ...` – PyQt5 je, stejně jako samotné Qt5, rozděleno na několik částí; v začátku využijeme především QtWidgets, QtCore a QtGui. Balík QtWidgets obsahuje různé grafické prvky, v tomto případě QMessageBox, a také samotný objekt QApplication.
- `import sys` potřebujeme na argumenty příkazové řádky, které ke svému běhu Qt potřebuje znát.
- `QApplication` je základní objekt aplikace. Stará se o samotné prostředí, nastavuje se v něm řada parametrů GUI (jako například interval pro dvojklik nebo styl oken) a nakonec se v něm schovává smyčka událostí (viz dále).
- `QMessageBox` je to okýnko, které jste před chvílí viděli. Metodou `show` je třeba jej zobrazit, jinak ho neuvidíte a nebudete moci odkliknout.
- `app.exec` nakonec celou aplikaci spustí, okýnko nakreslí a vyřídí vaše kliknutí na OK, aby jej mohl zase smazat a skončit. Pokud na tomhle řádku dostanete podivný syntax error, použijte `exec_` (s podtržítkem na konci).

Událostmi řízené programování

Programy, které si představujeme ve vzorových řešeních, jsou dávkové. Takový program přečte vstup, zpracuje jej a vydá výstup. Mezitím na uživatele nereaguje. Není interaktivní. Stejným druhem programů pravděpodobně řešíte například opendatové úlohy.

Abychom ale dokázali na uživatele reagovat (téměř) hned, musíme přistoupit k programu z jiné strany.

Jádrem programu bude *smyčka událostí* (event loop). Můžeme si ji pro začátek představit jako magického hlídacího psa, který si pamatuje všechno, co má hlídat a co v takovou chvíli má dělat. Hlídací pes většinu doby tvrdě spí, jakmile se však objeví nějaká událost, hned vyskočí a zavolá všechno, co se má při události spustit.

Místo postupného vykonávání zadaného programu si tedy zaregistrujeme nějaké *události* a jejich *obsluhu*. To se dělá zavoláním příslušné funkce; ukážeme si to. Když pak událost nastane (například stisk myšítka či klávesy nebo třeba vyprší časový limit), předá se popis této události obslužné funkci (tzv. handler či hook), která událost zpracuje. Jakmile nejsou žádné události k vyřízení, hlídací pes jde zase spát a počká, než se objeví nějaká další.

Takovému přístupu říkáme událostmi řízené programování (event-driven programming). Pořádné vysvětlení si ale necháme do některého z dalších dílů, nyní nám bude stačit takovýto úvod.

Netrpělivý program

Předvedeme si další typ události: časovač. Na tom si také ukážeme, jak se události vyřizují. Použijeme na to objekt typu `QTimer` z balíku `QtCore`, který zařídí, aby se nějaká akce vykonala se zpožděním.

Následující program zobrazí zprávu na omezenou dobu:

```
#!/usr/bin/python3

from PyQt5.QtCore import QTimer
from PyQt5.QtWidgets import \
    QApplication, QMessageBox

import sys

app = QApplication(sys.argv)

box = QMessageBox(text="ahoj světe")
box.show()

timer = QTimer(singleShot=True)
timer.timeout.connect(box.close)
timer.start(2000)

app.exec()
```

Tři řádky, které přibyly, nastavují časovač a jeho akci:

- `QTimer(singleShot=True)` vytvoří samotný objekt časovače a nastaví, že se spustí pouze jednou.
- `timer.timeout.connect(box.close)` připojí na událost `timeout` obslužnou funkci `b.close`. S výhodou využíváme toho, že `QMessageBox` má metodu `close`, která jej zavře, takže ji můžeme přímo zavolat.
- `timer.start(2000)` nastaví budík na 2 sekundy (hodnota je v milisekundách). Až uplynou, vyvolá se událost `timeout` a zavolají se všechny obslužné funkce, které jsme si k ní zaregistrovali.

Pokud bychom chtěli budík típnout, můžeme zavolat `timer.stop()`. Když to stihneme včas (před uplynutím nastavené doby), k události `timeout` nedojde.

Úkol 1 [4b]: Napište program, který po dvou sekundách box se zprávou neuzavře, ale zprávu přepíše například na text „budík už zvoní“. K tomu se hodí metoda `setText` třídy `QMessageBox`.

QWidget

Použít jako základ programu `QMessageBox` je ve skutečnosti dost nešťastné rozhodnutí. Pro počáteční ilustraci, jak fungují události v rámci Qt, to stačí, jinak je ale určený k zobrazování zpráv pro uživatele jako modální dialog; to znamená, že zbytek aplikace zamrzne, dokud uživatel box neodklikne. Později si ukážeme, jak se používá běžně.

GUI napsané v Qt se skládá z `QWidget`ů naskládaných do sebe. Každý prvek (i celé okno) je widget. `QMessageBox` v ukázce je také widget, ve kterém je rozmístěná zpráva, ikonka (pokud ji nastavíme) a tlačítka.

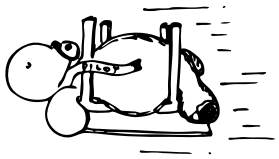
A aby Qt vědělo, jak widgety rozmístit uvnitř jiných widgetů, používá na to `QLayout`. Těch samozřejmě také existují různé typy, například `QVBoxLayout` a `QHBoxLayout` pro umístění prvků nad sebe či vedle sebe, nebo `QGridLayout` pro rozmístění prvků do tabulky.

Stopky

Napíšeme si program, který bude ukazovat čas, který uplynul od stisku tlačítka start.

```
#!/usr/bin/python3

from PyQt5.QtCore import QTimer
from PyQt5.QtWidgets import \
    QApplication, QWidget, QLabel, \
    QPushButton, QVBoxLayout

import sys

class Stopky(QWidget):
    def __init__(self, *args, **kwargs):

        # Inicializace QWidgetu samotného
        super().__init__(*args, **kwargs)

        # Výroba ovládacích prvků
        self.label = QLabel(self)
        self.button = QPushButton(self,
                      text="start")
        self.button.clicked.connect(self.start)

        # Umístění ovládacích prvků
        self.layout = QVBoxLayout(self)
        self.layout.addWidget(self.label)
        self.layout.addWidget(self.button)

        # Zobrazení stopek
        self.setLayout(self.layout)
        self.show()

    def start(self):
        # Vyrob časovač
        self.timer = QTimer()
        self.timer.timeout.connect(self.tick)
        self.timer.start(1000)

        # Počáteční čas je nula
        self.elapsed = 0
        self.updateLabel()

    def tick(self):
        # Uplynula další sekunda
        self.elapsed += 1
        self.updateLabel()

    def updateLabel(self):
        # Zobrazení uplynulého času
        self.label.setText(str(self.elapsed))

# Spuštění celého programu
app = QApplication(sys.argv)
stopky = Stopky()
app.exec()
```

Program je nyní celý schován v objektu `Stopky`, což je také hlavní okno programu. V konstruktoru objektu se okno sestaví a zobrazí a taktéž se připojí událost k tlačítku. Obslužná funkce tlačítka pak spustí časovač a jeho obslužná funkce periodicky zvyšuje čítač a zobrazuje jeho změněnou hodnotu.



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.

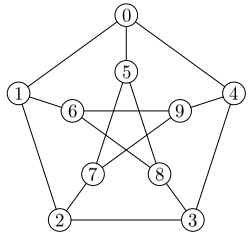
Stromy a grafy v informatice

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.

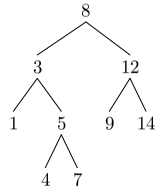
Grafy jsou velmi široké téma. Můžeme hledat jejich minimální kostry, můžeme v nich hledat nejkratší cesty či skrze ně pouštět pod tlakem vodu. Více o nich si tedy můžete přečíst v některé z našich specializovaných grafových kuchařek, které odkazujeme z našeho kuchařkového rozcestníku.[7]

Stromy

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

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

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



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

[7] `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 [4b]: Přidejte do výše uvedeného programu tlačítko „stop“, které stopky zastaví.

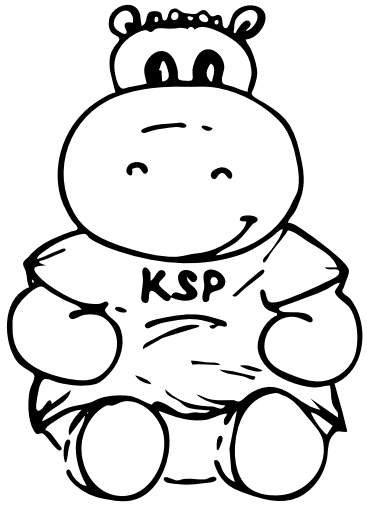
Úkol 3 [4b]: Přidejte do výše uvedeného programu ještě jeden `QLabel`, který bude zobrazovat, v jakém intervalu se stopky aktualizují (nyní je to jedna sekunda). Přidejte také dvě tlačítka, kterými je možno interval zvyšovat a snižovat.

Úkol 4 [3b]: Zařiďte, aby změna intervalu v předchozím úkolu byla okamžitá bez čekání na předchozí tik časovače.

Poznámka: Úkoly 2, 3 a 4 je možno vyřešit jedním programem.

Stopkami dnešní náhled do PyQt5 uzavřeme a příště si povíme trochu víc teorie, která se ke GUI a UŘP[2] váže, a ukážeme další ovládací prvky a události, které se ve Qt dají použít.

Maria Matějka



[2] Událostmi řízené programování

Recepty z programátorské kuchařky: Základní algoritmy

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

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

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

Většinu klíčových částí se pokusíme též ukazovat v podobě zdrojového kódu ve dvou různých jazycích (v nízkoúrovňovém C, kde je zápis blízký tomu, jak počítač doopravdy pracuje, a v Pythonu, ve kterém se píše o něco příjemněji). Nebudeme ale probírat základy syntaxe těchto jazyků, ty si případně můžete nastudovat jinde.[3]

Část první: Základní pojmy

Algoritmus a program

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

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

Mezi základní příkazy patří:

- Manipulace s daty v paměti (uložení či načtení hodnoty, detailněji v další kapitole).
- Provedení nějakého numerického výpočtu ($+, -, *, /$).
- Vyhodnocení 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], ...`).[5]

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

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

U pole již má smysl přemýšlet, jak dlouho bude která operace trvat. Díky tomu, že jsou jednotlivé prvky v poli naskládané pevně za sebou, 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,[6] nejdříve však doporučujeme dočíst tuto kuchařku.

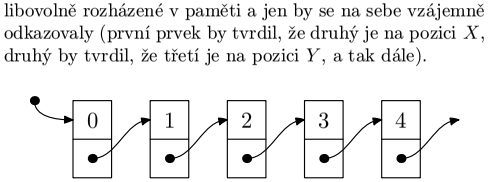
Přidání nového prvku na konec pole také zvládneme v konstantním čase. Problém je přidání nového prvku někam doprostřed (což se nám typicky stane, pokud budeme chtít udržovat hodnoty v poli seřazené a zároveň do něj vkládat nové). V takovém případě se totiž všechny prvky za vkládaným musí posunout o jednu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

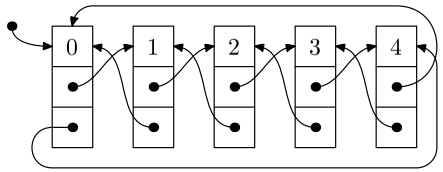


K lepšímu pochopení tohoto principu je důležité si vysvětlit, co to je *ukazatel* (nebo také *odkaz* či anglicky *pointer*). Každé místo v paměti počítače má své číselné označení, kterému říkáme *adresa*. Když si vytváříme nějakou pojmenovanou proměnnou, ta se vlastně odkazuje na určité místo v paměti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

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

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

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

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