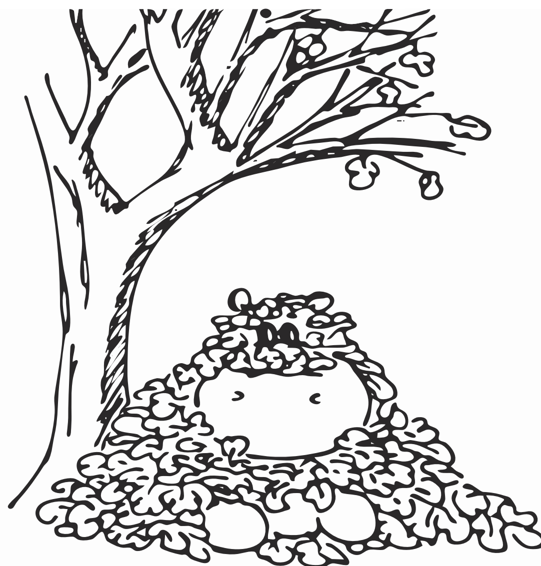
Milí řešitelé!



Pomalu přichází zima, a abyste měli o dlouhých večerech co dělat, máme pro vási druhou sérii našeho semináře. Neobsahuje ještě vaše opravená řešení první série, abyste na nové příklady nemuseli čekat moc dlouho. Nicméně opravená řešení vám (spolu se zadáním třetí série) přijdou dříve, než budete muset tuto druhou sérii odeslat.

Pokud chcete posílat svá řešení elektronickou cestou, prosíme držte se instrukcí, které můžete najít na *http://ksp.mff.cuni.cz/submit/*. Řešení, která přišla mailem, jsme přijali pouze výjimečně.

Termín odeslání Vašich řešení druhé série jest stanoven na 11. prosince 2006 a naše adresa je stále stejná:

Korespondenční seminář z programování
KSVI MFF UK
Malostranské náměstí 25
118 00 Praha 1

Aktuální informace o KSP můžete nalézt na stránkách *http://ksp.mff.cuni.cz/*, diskutovat můžete na fóru *http://ksp.mff.cuni.cz/forum/* a záludné dotazy organizátorům lze zasílat e-mailem na adresu *ksp@mff.cuni.cz.*

Druhá série devatenáctého ročníku KSP

Kapitola 2 *Ve dveřích stála má nová klientka a dvě nadřený gorily. Něžně ji hodily na podlahu vedle mě a zazubily se. Teda, ony se moc nezubily, protože neměly čím. A hned jsem pochopil důvod – obě v ruce třímaly tabulku čokolády. V tom mě osvítil nápad hodný mého génia.*

„Čo ťak blbě čumíš?“ zeptal se mě ten kolozubější.
„Koukám se, co držíš v ruce, a tak mi napadá, to jíte tu čokoládu jen tak? Vy nevíte, co legrace se s ní dá užít, než jí sníte?“
Tázavé přiblblé kukuče mi prozradily, že netuší, a tak jsem začal s výkladem pravidel:

19-2-1 Čokoláda podruhé 6 bodů

Typická gangsterská čokoláda vypadá jako obdélník o rozměrech $M \times N$ dílků, kde alespoň jeden rozměr je sudý. Dva hráči se střídají v lámání tak, že si ten, kdo je na tahu, vybere nějakou celistvou část čokolády a rozlomí jí na dvě části. Gangster, který odlomí dílek 1×1, prohrál. Vaším úkolem je najít takovou strategii, aby začínající mafián vždy vyhrál.

Příklad: Pro čokoládu o rozměrech 2×3 musí první zloduch lámat na dvě části 1×3, načež druhý zloduch prohrává, protože ať láme, jak láme, vždy získá dílky 1×1, 1×2 a 1×3.

Pozn.: Protože čokolády o rozměrech 1×1 a 1×2 neposkytovaly gangsterům dostatečné mlsavé uspokojení, upustily nelegální čokoládovny od jejich výroby, takže je zanedbejte.

Dveře se zabouchly a my slyšeli jen křupání tabulek čokolád, skřípání zubů a po chvíli … „Ty podvodníku!“
„Čože? Já hlaju naplošto a školo češtně.“
A pak třeskly dva výstřely, ozvaly se dvě tupé rány a já jsem velmi odvážně vykoukl dírkou ve dveřích.
„Je po nich, jsme volní!!!“ zajásal jsem.
„A jak se dostaneme ven, Einsteine?“ zpražila mě pohledem.
Faktem je, že na tento způsob jednání jsem byl od žen, zvláště tak krásných jako ona, zvyklý. Nechtěl jsem se ale nechat zahanbit, a tak jsem začal usilovně přemýšlet. Po patnácti minutách mého přemýšlení konečně na něco přišla, zvedla se, chvíli něco šťourala v takové krabičce s dráty na dveřích a najednou se dveře otevřely. Když jsem po letech zjistil, jak je to snadné, tak jsem se divil, že jsem na to nepřišel sám.

19-2-2 Kvalitní hesla 6 bodů

Původní obsah následující pasáže jsme museli vystřihnout na nátlak rodiny Panoraiků, která popisovaný systém stále používá. A tak vás tato rodina požádala alespoň o pomoc při rozpoznávání hesel, na která je policejní program krátký. Heslo je, jak známo, posloupnost alfanumerických znaků délky N. Díky znalosti algoritmu, který policejní počítač používá, se nám povedlo zjistit, jak „kvalitu“ hesla spočítat – heslo je tím lepší, čím větší jeho část se v něm opakuje.

Vaším úkolem je po zadání hesla najít maximální d a různé indexy i a j takové, aby se *souvislé* podúseky délky d začínající na těchto indexech shodovaly. Podúseky se mohou překrývat. Pokud těchto dvojic s maximálním d existuje několik, tak stačí najít jednu z nich.

Příklad: Pro $N = 13$ a heslo `abrakadabraka` jsou hledané indexy 1 a 8 (`abraka`) a délka je 6. Pro $N = 7$ a heslo `aaaaaaa` jsou hledané indexy 1 a 2 a délka je opět 6.

Bonus: Pokud bude váš program pracovat opravdu rychle, dostanete až 5 bonusových bodů.

Dveře se otevřely a my začali prchat. Teda, já sem začal prchat. Ona stála ve dveřích a rozhlížela se. „Na co čekáš? Mizíme!“
„Ne, nejdřív musíme do jeho kanceláře, ukradneme vše, co se týká jeho práce!“
„Já si nemysl… ženská pitomá, kam zase běžíš?“
Za chvilku jsme dorazili ke dveřím. Sice byly zamčené, ale má nová průvodkyně se ukázala být nejen šarmantní, ale i velmi zručná. Práce se sponkou jí trvala jen několik vteřin. Když jsme vešli, okamžitě se vrhla k šuplatům a začala se přehrabovat ve stole. Poté stejně sebevědomě zplundrovala trezor (kde jen vzala heslo?). Jen jsem tupě zíral a jediné, na co jsem se zmohl, byla otázka.
„Tak málo papírů? Vždyť je skoro prázdnej. To mu určitě všechno dělaj jeho účetní.“
„Blázníš? K těmhle věcem nikoho nepouští, všechno si dělá sám.“
„Tomu nevěřím, já dřu od rána do večera, popíšu stohy papírů a div nebydlím pod mostem.“
„No jo, ale ty nemáš program, který ti naplánuje činnosti tak, že málo děláš a hodně vyděláš.“

Vám je už určitě jasné, že takový program máte napsat.

19-2-3 Moneymaker 10 bodů

Na vstupu dostane váš program číslo N, což je počet úkolů ke zpracování. Zpracování každé úlohy zabere jednotkový čas. Dále pak N řádků, každý se dvěma čísly. První číslo znamená, dokdy je třeba úkol vykonat, a druhé číslo je odměna, kterou za splněný úkol dostaneme. V jednom čase mohu pracovat právě na jednom úkolu. Výstupem programu by pak mělo být takové pořadí úkolů, aby zisk byl maximální. Pokud je takových pořadí více, staší libovolné z nich.

Příklad: Pro $N = 4$ a záznamy

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

je optimální pořadí 3, 4 a 1.

Pobrali jsme rychle vše, co se dalo, a vyběhli ven z budovy. Po cestě jsme zneškodnili několikery spící stráže, až na jednoho. Ten bohužel zburcoval všechny ostatní a ti zaujali obrannou formaci. To nám útěk značně zkomplikovalo a jedinou naší nadějí bylo to, že jsme věděli, jak taková formace vzniká. Ale jak taková obranná formace vypadá? To už jsme nevěděli. S tím nám budete muset poradit vy.

19-2-4 Optimální formace 11 bodů

Formace je tvořena N střelci. Střelec číslo i má dostřel a_i. Střelci stojí ve vrcholech konvexního N-úhelníku (konvexní N-úhelník je takový, že všechny jeho vnitřní úhly jsou v intervalu $(0°, 180°)$) v takovém pořadí, v jakém jsou zapsáni na vstupu. Cílem je vytvořit takovou formaci, aby každý střelec dostřelil k následujícímu a aby obvod N-úhelníku byl maximální. Program by měl libovolnou jednu takovou formaci nalézt a vypsat souřadnice jednotlivých střelců.

Příklad: Pro $N = 5$ a dostřely střelců $(2.5, 2.5, 3, 5, 2)$ leží jeden z možných N-úhelníků na souřadnicích $[0, 0]$, $[0, 2.5]$, $[-2, 4]$, $[-5, 4]$, $[-2, 0]$. Pro $N = 4$ a dostřely $(10, 2, 3)$ nelze N-úhelník sestrojit.

Naštěstí se nám díky znalosti přesného tvaru formace podařilo najít slabinu v jejich obraně, a tak se nám povedlo uprchnout.

Pozn. KSP: Přesprstovi zjevně nedošlo, že popis formace neurčuje jednoznačně její tvar. Záhadou zůstává to, že právě vaše odpověď byla ta správná.

Celí zadýchaní jsme doběhli do hlubokého lesa.
„Co si teď počneme? Kam půjdeme? On si nás najde všude a příště už takové štěstí mít nebudeme!“ ptala se zděšeným hlasem moje společnice.
„Znám křišťálovou studánku, kde nejhlubší je les . . . “
„Co to meleš za nesmysly?“
„Já jsem to řekl nahlas? Já jako myslel, že poblíž tý studánky je opuštěná chalupa, kde bychom se mohli aspoň na noc schovat.“
„A jak jí asi najdeme?“
„No jednoduše, prostě najdeme dva stromy v lese, které jsou u sebe nejblíže ze všech, a tam je les zákonitě nejhlubší.“
„No jo, to mě vlastně nenapadlo . . . “

19-2-5 Hluboký les 13 bodů

A zatímco si Přesprst vychutnavá svůj malý triumf, napište program, který takové stromy najde. Váš program dostane na vstupu číslo N a dále N řádků s reálnými souřadnicemi jednotlivých stromů v lese. V případě, že je takových dvojic stromů víc, stačí vypsat libovolnou z nich.

Příklad: Pro $N = 4$ a stromy

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

by měl program vypsat: Stromy 2 a 3 jsou si k sobě nejblíže.

Po hodinách prodírání se lesem nás sama Prozřetelnost přivedla před práh chaty. Unaveně jsme padli do postele a tvrdě usnuli. Spal jsem snad dva dny, než jsem konečně dokázal otevřít víčka a rozhlédnout se kolem sebe. A v tom mi došlo, že to, co nás do té chaty přivedlo, rozhodně nebyla Prozřetelnost.

To be continued. . .

19-2-6 Prolog 12 bodů

Milí programátoři v Prologu,

jsme rádi, že se vám první díl seriálu o programovacím jazyku Prolog líbil, a přinášíme vám další zajímavosti ze světa logického programování :o)

Termy

Základní jednotkou programovacího jazyka Prolog je *term*. Termy se v Prologu dělí na jednoduché (atomy, čísla, proměnné) a na struktury. Atomy, čísla a proměnné už znáte. *Struktura* je rekurzivní, složený term, tedy součástí struktury mohou být další termy. Příklady struktur jsou:

```
datum(den(3),mesic(10),rok(2006)).
osoba(jmeno(bretislav),prijmeni(rozsejpal)).
```

Ve skutečnosti je strukturou dokonce i klauzule

```
matka(X) :- rodic(X,_), zena(X).
```

`:-` je totiž binární predikát, který má dva argumenty: hlavu a tělo klauzule, a píše se doprostřed, tedy infixově. Klidně byste mohli psát

```
:-(matka(X), (rodic(X,_), zena(X))).
```

Unifikace pořádně a naposled

Po zadání dotazu, například

```
?-je_matka(X).
```

začne Prolog procházet program shora dolů a snaží se najít, „přiřadit“, neboli *unifikovat* zadaný dotaz s hlavou nějaké klauzule v programu. Jinými slovy prostě najít jaksi odpovídající řádek programu. Jak ale přesně funguje unifikace, když jsme teď zjistili, že úplně všechno v Prologu je term a ty můžou být pěkně složité? K naší radosti to funguje přesně tak, jak byste čekali a jak byste to dělali intuitivně:

Jestliže jeden z termů je (nezunifikovaná, volná) proměnná a druhý libovolný term (různý od proměnné, například nějaká struktura, atom nebo číslo), okamžitě se do proměnné dosadí daný term. To jsme viděli v minulém díle.

Jestliže jsou oba termy proměnné, pak je výsledkem jejich ztotožnění.

Příklad:

```
?-A = B.
```

Jsou-li `A` i `B` volné, unifikace uspěje a proměnné se od tohoto okamžiku budou chovat jako jedna. Pokud `A` bude v budoucnu unifikována například s atomem `kleofac`, pak samozřejmě bude i `B = kleofac`.

Jestliže jsou oba termy atomy nebo čísla, pak unifikace uspěje pouze tehdy, pokud jsou oba stejné.

Jestliže jsou oba termy nějaké struktury, pak provedeme unifikaci rekurzivně. Podíváme se na jejich argumenty a pokud jich je stejný počet, zkusíme každý odpovídající si pár argumentů unifikovat. Pokud se to podaří pro každý argument (a samozřejmě pokud se struktury jmenují stejně), jsou struktury shodné.

V žádném jiném případě nejsou termy shodné a nelze je ani unifikovat.

Tím jsme si podrobně vysvětlili mechanismus unifikace. Teď už také chápeme, co přesně dělá binární predikát `=`, totiž že vyvolává unifikaci na své argumenty.

Co myslíte, že by se stalo, pokud byste napsali `A = f(A)`?

Seznamy

V Prologu máme k dispozici datovou strukturu *seznam*. Seznam je posloupnost termů, například čísel, struktur, nebo dalších seznamů.

Prázdný seznam se značí atomem `[]`. Neprázdný seznam se zapíše například takto: `[a,b,c]` nebo `[1,2,3,4]`.

Prolog chápe seznam jako dvě části: *hlavu* a *tělo*. Hlava je první prvek seznamu a tělo celý zbytek seznamu. Tento zbytek chápe Prolog opět jako seznam, tedy v případě seznamu `[a,b,c]` je hlavou prvek `a` a tělem opět seznam `[b,c]`.

K tomu, abychom ze seznamu snadno oddělili hlavu, slouží ještě jiný způsob zápisu seznamu: seznam `[a,b,c]` můžeme napsat jako `[a|[b,c]]`.

Už nás také napadá, jak budeme s prologovskými seznamy pracovat. Vždy si oddělíme hlavu, něco s ní uděláme a potom se pustíme rekurzivně na zbytek seznamu. Hned si to ukážeme na příkladu hledání prvku v seznamu:

```
% prvek(X,Seznam) je X prvkem seznamu Seznam
prvek(X,[X|_]).
prvek(X,[_|Telo]) :- prvek(X,Telo).
```

V prvním řádku se díváme, jestli hledaný prvek není náhodou přímo v hlavě seznamu. Pokud hledaný prvek není v hlavě seznamu, musíme vzít zbytek (tělo) seznamu a pátrat v něm.

Jiný příklad, tentokrát hledáme poslední prvek seznamu:

```
% posl(Seznam,X) vraci posledni prvek seznamu
posl([X],X).
posl([_|Telo],Posl) :- posl(Telo,Posl).
```

První řádek je jasný, poslední prvek jednoprvkového seznamu je onen jediný prvek. Druhý řádek je zajímavější, pokud máme seznam s jedním prvkem, za kterým je ještě nějaký další seznam, zavoláme si rekurzivně predikát `posl` na tělo seznamu s utrženou hlavou. Takto se postupně volají predikáty `posl` na čím dál kratších seznamech. Proměnná `Posl` je přitom stále volná. Jakmile dojedeme na konec seznamu a máme už jen jednoprvkový seznam, dostaneme se na dno rekurze, uplatní se první řádek programu a v tom okamžiku se nám úspěšně unifikuje proměnná `Posl`.

Nakonec příklad bez komentáře:

```
% vypust(X,Sezn,NovySezn)
% vypusti jeden vyskyt X ze seznamu
% Sezn a vrati novy seznam NovySezn
vypust(X,[],[]).
vypust(X,[X|T],T).
vypust(X,[Y|T],[Y|L]) :- vypust(X,T,L).
```

Kvíz

⋆ Který zápis seznamu *není* správný?

1. `[ [a,b], c]`
2. `[a, b, [c] ]`
3. `[a, [b,c] ]`
4. `[ [a,b] | c]`
5. `[a,b,c,[]]`

⋆ Kolik prvků má seznam `[a,b,c,[]]`?

1. 2
2. 3
3. 4

⋆ Jaký výsledek dostaneme při `?-prvek(a,Seznam).`?

1. `No.`
2. `Yes.`
3. `[a,a,a,a,a,...]`
4. `[a,a,a,a,a,...] ; No.`
5. `[a|_] ; [_,a|_] ; [_,_,a|_] ; ...`

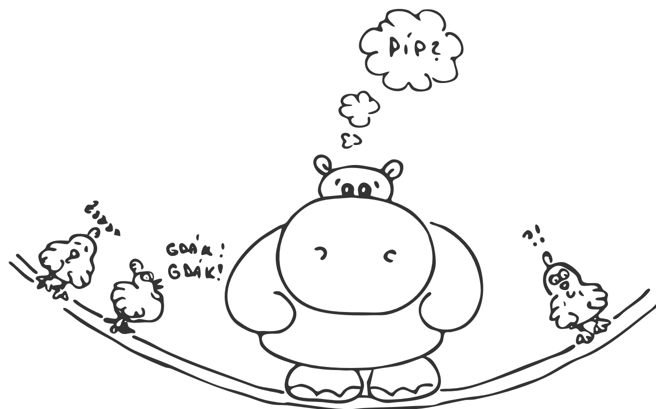
⋆ Jaký bude výsledek dotazu
`p(A,B,q(c,A,B))=p(a,b,q(c,a,d))`?

1. Yes.
2. No.

Soutěžní úložky

1. Příliš těžké slepice (4 body) Slepice sedí na hřadě. Bidýlko se pod nimi prohýbá a každou chvíli hrozí, že praskne. Nejprohnutější a nejzatíženější je bidýlko pod prostřední slepicí. Slepice ale neví, která z nich je uprostřed. Napište slepicím program v Prologu, který dostane seznam slepic tak, jak sedí na bidýlku zleva doprava, a vybere prostřední z nich. Pokud je slepic sudý počet, vybere tu, která sedí více vpravo. Nezapomeňte, že slepice neumí počítat, takže nesmíte používat žádnou aritmetiku. Program by měl být co nejrychlejší. Slepice také neoplývají přílišnou chytrostí, takže byste svůj program měli náležitě okomentovat a popsat :o))

Příklad: Pro vstup `[a,b,c]` je prostřední slepice `b`, pro vstup `[1,2,3,4]` je prostřední slepice `[3]`.



2. Permutující slepice (4 body) Poté, co jste vyřešili problém bidla praskajícího pod slepičími špeky, začalo slepice pálit dobré bydlo a obrátily se na vás s dalším problémem. Slepice se mezi sebou neustále hašteří, ve které části bidla bude která sedět. Už to tak dál nejde, a proto se budou každou noc střídat v pořadí. Napište program, který vypíše všechny možné rozmístění slepic na bidýlku, každé právě jednou. Vstupem programu je seznam slepic a výstupem seznam seznamů obsahující všechny permutace slepic, tj. všechny možnosti, jak mohou slepice sedět, každou právě jednou.

Příklad: Pro vstup `[a,b]` je výstupem tento seznam seznamů: `[ [a,b], [b,a] ]`, pro vstup `[1,2,3]` je výstupem následující seznam permutací: `[ [1,2,3],[1,3,2], [2,1,3], [2,3,1], [3,2,1], [3,1,2] ]`.

3. Palindromické slepice (4 body) Jelikož jste dokázali usmířit rozhádané opeřence, byli jste požádáni o vyřešení posledního problému. Slepice zjistily, že nejbezpečnější rozmístění na bidýlku je takové, že proti sobě symetricky sedí vždy slepice stejné váhy. Například rozmístění vah `[1,2,3,3,2,1]`, `[4]` či `[4,3,5,6,5,3,4]` jsou bezpečná umístění, zatímco `[4,3]`, `[4,3,1]`, `[5,6,7,7,6,6]` nejsou bezpečná umístění. Napište program, který zjistí, zda slepice sedí na bidýlku bezpečně. Jinými slovy, zjistěte, zda číslo zapsané jako seznam číslic je palindrom, čili slovo, které se čte stejně zepředu i zezadu. Pro sladkou bodovou odměnu se snažte program co *nejvíce* urychlit (nejlépe na lineární :–).

Tímto se s vámi loučíme a těšíme se na setkání v příštím díle. Kokokodák!

Recepty z programátorské kuchařky

Rozděl a panuj

Dnešní díl programátorské kuchařky se bude zabývat algoritmy založenými na metodě *Rozděl a panuj.* A tak by se slušelo začít tím, jaká je myšlenka této metody: Často se setkáme s úlohami, které lze snadno rozdělit na nějaké menší úlohy a z jejich výsledků zase snadno složit výsledek původní velké úlohy. Přitom menší úlohy můžeme počítat opět týmž algoritmem (zavoláme si ho rekurzivně), leda by již byly tak maličké, že dokážeme odpovědět triviálně bez jakéhokoliv počítání. Zkrátka jak říkali staří římští císařové: Divide et impera. Uveďme si pro začátek jeden staronový příklad:

Quicksort

QuickSort (alias QS) je algoritmus pro třídění posloupnosti prvků. Už o něm byla jednou řeč v „třídící kuchařce" v první sérii 18. ročníku KSP. Tentokrát se na něj podíváme trochu podrobněji a navíc nám poslouží jako ingredience pro další algoritmy.

QS v každém svém kroku zvolí nějaký prvek (budeme mu říkat *pivot*) a přerovná prvky v posloupnosti tak, aby napravo od pivota byly pouze prvky větší než pivot a nalevo pouze menší. Pokud se vyskytnou prvky rovné pivotu, můžeme si dle libosti vybrat jak levou, tak pravou stranu posloupnosti, funkčnost algoritmu to nijak neovlivní. Tento postup pak rekurzivně zopakujeme zvlášť pro prvky nalevo a zvlášť pro prvky napravo od pivota, a tak získáme setříděnou posloupnost.

Implementací QS je mnoho a mimo jiné se liší způsobem volby pivota. My si předvedeme jinou, než jsme ukazovali v třídící kuchařce (hlavně proto, že se nám od ní pak snadno budou odvozovat další algoritmy) a pro jednoduchost budeme jako pivota volit poslední prvek zkoumaného úseku:

```
{budeme třídit takováto pole}
type Pole=array[1..MaxN] of Integer;

{přerovnávací procedura pro úsek a[l..r]}
function prer(a:Pole; l,r:Integer):Integer;
var i,j,x,q:Integer;
begin
 {pivotem se stane poslední prvek úseku}
 x:=a[r];                        {hodnota pivota}
 i:=l-1;  {a[i] bude vždy poslední <= pivotovi}

 {samotné přerovnávání}
 for j:=l to r-1 do
  if a[j]<=x then  {právě probíraný prvek     }
  begin            {menší/rovný hodnotě pivota}
   Inc(i);         {pak zvyš ukazatel         }
   q:=a[j];        {a proveď přerovnání prvku }
   a[j]:=a[i];
   a[i]:=q;
  end;

 {nakonec přesuneme pivota za poslední <=}
 q:=a[r];
 a[r]:=a[i+1];
 a[i+1]:=q;
 prer:=i+1;      {vrátíme novou pozici pivota}
end;

{hlavní třídící procedura, třídí a[l..r]}
procedure QuickSort(a:Pole; l,r:Integer);
var m:Integer;
begin
 if l<r then              {máme ještě co dělat?}
 begin
  m:=prer(l,r);   {přerovnej, m pozice pivota}
  QuickSort(l,m-1);     {setřiď prvky napravo}
  QuickSort(m+1,r);      {setřiď prvky nalevo}
 end;
end;
```

Bohužel volit pivota právě takto je docela nešikovné, protože se nám snadno může stát, že si vybereme nejmenší nebo největší prvek v úseku (rozmyslete si, jak by vypadala posloupnost, ve které se to bude dít pokaždé), takže dostaneme-li posloupnost délky N, rozdělíme ji na úseky délek $N-1$ a 1, načež pokračujeme s úsekem délky $N-1$, ten rozdělíme na $N-2$ a 1 atd. Přitom pokaždé na přerovnání spotřebujeme čas lineární s velikostí úseku, celkem tedy $\mathcal{O}(N+(N-1)+(N-2)+\ldots+1)=\mathcal{O}(N^2)$.

Na druhou stranu pokud bychom si za pivota vybrali vždy *medián* z právě probíraných prvků (tj. prvek, který by se v setříděné posloupnosti nacházel uprostřed; pro sudý počet prvků zvolíme libovolný z obou prostředních prvků), dosáhneme daleko lepší složitosti $\mathcal{O}(N\log N)$. To dokážeme snadno:

Přerovnávací část algoritmu běží v čase lineárním vůči počtu prvků, které máme přerovnat. V prvním kroku QS pracujeme s celou posloupností, čili přerovnáme celkem N prvků. Následuje rekurzivní volání pro levou a pravou část posloupnosti (obě dlouhé $(N-1)/2\pm1$); přerovnávání v obou

částech dohromady trvá opět $\mathcal{O}(N)$ a vzniknou tím části dlouhé nejvýše $N/4$. Zanoříme-li se v rekurzi do hloubky k, pracujeme s částmi dlouhými nejvýše $N/2^k$, které dohromady dají nejvýše N (všechny části dohromady dají prvky vstupní posloupnosti bez těch, které jsme si už zvolili jako pivoty). V hloubce $\log_2 N$ už jsou všechny části nejvýše jednoprvkové, takže se rekurze zastaví. Celkem tedy máme $\log_2 N$ hladin (hloubek) a na každé z nich trávíme lineární čas, dohromady $\mathcal{O}(N \log N)$.

V tomto důkazu jsme se ale dopustili jednoho podvodu: zapomněli jsme na to, že také musíme medián umět najít. Jak z této nepříjemné situace ven?

- *Naučit se počítat medián.* Ale jak?
- *Spokojit se se „lžimediánem“:* kdybychom si místo mediánu vybrali libovolný prvek, který bude v setříděné posloupnosti v prostřední polovině (čili alespoň čtvrtina prvků bude větší a alespoň čtvrtina menší než on), získáme také složitost $\mathcal{O}(N \log N)$, neboť úsek délky N rozložíme na úseky, které budou mít délky nejvýše $(1-1/4)\cdot N$, takže na k-té hladině budou úseky délek $\leq (1-1/4)^k \cdot N$, čili hladin bude maximálně $\log_{1-1/4} N = \mathcal{O}(\log N)$. Místo $1/4$ by dokonce fungovala libovolná jiná konstanta, ale ani to nám nepomůže k tomu, abychom uměli lžimedián najít.
- *Recyklovat pravidlo* typu „vezmi poslední prvek“ a jen ho trochu vylepšit. To bohužel nebude fungovat, protože pokud budeme při výběru pivota hledět jenom na konstantní počet prvků, bude poměrně snadné přijít na vstup, pro který toto pravidlo bude dávat kvadratickou složitost, i když obvykle půjde dokázat, že takových vstupů je „málo“. [Také se to tak často dělá.]
- *Volit pivota náhodně* ze všech prvků zkoumaného úseku. K náhodné volbě samozřejmě potřebujeme náhodný generátor a s těmi je to svízelné, ale zkusme na chvíli věřit, že jeden takový máme nebo alespoň že máme něco s podobnými vlastnostmi. Jak nám to pomůže? Náhodně zvolený pivot nebude sice přesně uprostřed, ale s pravděpodobností $1/2$ to bude lžimedián, takže po průměrně dvou hladinách se ke lžimediánu dopracujeme (rozmyslete si, proč, nebo nahlédněte do seriálu o pravděpodobnostních algoritmech v 16. ročníku). Proto časová složitost takovéhoto randomizovaného QS bude v průměru 2-krát větší, než lžimediánového QS, čili v průměru také $\mathcal{O}(N \log N)$. Jednoduše řečeno, zatímco fixní pravidlo nám dalo dobrý čas pro průměrný vstup, ale existovaly vstupy, na kterých bylo pomalé, randomizování nám dává dobrý průměrný čas pro všechny možné vstupy.

Hledání k-tého nejmenšího prvku

Nad QuickSortem jsme zvítězili, ale současně jsme při tom zjistili, že neumíme rychle najít medián. To tak nemůžeme nechat, a proto rovnou zkusíme vyřešit obecnější problém: najít k-tý nejmenší prvek (medián je to pro $k = \lfloor N/2 \rfloor$).

První řešení této úlohy se nabízí samo. Načteme posloupnost do pole, prvky pole setřídíme nějakým rychlým algoritmem a kýžený k-tý nejmenší prvek nalezneme na k-té pozici v nyní již setříděném poli. Má to však jeden háček. Pokud prvky, které máme na vstupu, umíme pouze porovnat, pak nedosáhneme lepší časové složitosti (a to ani v průměrném případě) než $\mathcal{O}(N \log N)$ – rychleji prostě třídit nelze, důkaz můžete najít například v třídící kuchařce.

O něco rychlejší řešení je založeno na výše zmíněném algoritmu QuickSort (často se mu proto říká QuickSelect). Opět si vybereme pivota a posloupnost rozdělíme na prvky menší než pivot, pivota a prvky větší než pivot (pro jednoduchost budeme předpokládat, že žádné dva prvky posloupnosti nejsou stejné). Pokud se pivot nalézá na k-té pozici, je to hledaný k-tý nejmenší prvek posloupnosti, protože právě $k-1$ prvků je menších. Zbývají dva případy, kdy tomu tak není. Pakliže je pozice pivota v posloupnosti větší než k, pak se hledaný prvek nalézá nalevo od pivota a postačí rekurzivně najít k-tý nejmenší prvek mezi prvky nalevo. V opačném případě, kdy je pozice pivota menší než k, je hledaný prvek v posloupnosti napravo od pivota. Mezi těmito prvky však nebudeme hledat k-tý nejmenší prvek, ale $(k-p)$-tý nejmenší prvek, kde p je pozice pivota v posloupnosti.

Časovou složitost rozebereme podobně jako u QuickSortu. Nešikovná volba pivota dává opět v nejhorším případě kvadratickou složitost. Pokud bychom naopak volili za pivota medián, budeme nejprve přerovnávat N prvků, pak jich zbude nejvýše $N/2$, pak nejvýše $N/4$ atd., což dohromady dává složitost $\mathcal{O}(N + N/2 + N/4 + \ldots + 1) = \mathcal{O}(N)$. Pro lžimedián dostaneme rovněž lineární složitost a opět stejně jako u QS můžeme nahlédnout, že náhodnou volbou pivota dostaneme v průměru stejný čas jako se lžimediánem.

Program bude velmi jednoduchý, využijeme-li přerovnávací proceduru od QS:

```
function kty(var a:Pole; l,r,k:Integer):Integer;
var x,z:Integer;
begin
 x:=prer(a,l,r); {přerovnej, x je pozice pivota}
 z:=x-l+1;     {pozice pivota vzhledem k [l..r]}
 if k=z then
   kty:=a[x]            {k-tý nejmenší je pivot}
 else if k<z then
   kty:=kty(a,l,x-1,k) {k-tý nejmenší je nalevo}
 else
   kty:=kty(a,x+1,r,k-z);               {napravo}
end;
```

k-tý nejmenší podruhé, tentokrát lineárně a bez náhody

Existuje však algoritmus, který řeší naši úlohu lineárně, a to i v nejhorším případě. Je založený na ďábelském triku: zvolit vhodného pivota (jak ukážeme, bude to jeden ze lžimediánů) rekurzivním voláním téhož algoritmu. Zařídíme to takto:

- Pokud jsme dostali méně než 6 prvků, použijeme nějaký triviální algoritmus, například si posloupnost setřídíme InsertSortem (opět viz třídící kuchařka) a vrátíme k-tý prvek setříděné posloupnosti.
- Rozdělíme prvky posloupnosti na pětice; pokud není počet prvků dělitelný pěti, poslední pětici necháme nekompletní.
- Spočítáme medián každé pětice. To můžeme provést například rekurzivním zavoláním celého našeho algoritmu, čili v důsledku InsertSortem. (Také bychom si mohli pro 5 prvků zkonstruovat rozhodovací strom s nejmenším možným počtem porovnání, což je rychlejší, ale jednak pouze konstanta-krát, jednak je to daleko pracnější.)
- Máme tedy $N/5$ mediánů. V nich rekurzivně najdeme medián m (označíme mediány pětic za novou posloupnost a na ní začneme opět od prvního bodu).
- Přerovnáme vstupní posloupnost po quicksortovsku a jako pivota použijeme prvek m. Po přerovnání je pivot,

podobně jako v předchozím algoritmu, na $(z+1)$-ní pozici v posloupnosti, kde z je počet prvků s menší hodnotou, než má pivot.

- Opět, podobně jako u předchozího algoritmu, pokud je $k = z+1$, pak je právě pivot m k-tým nejmenším prvkem posloupnosti. V případě, že tomu tak není a $k < z+1$, budeme hledat k-tý nejmenší prvek mezi prvními z členy posloupnosti, v opačném případě, kdy $k > z+1$, budeme hledat $(k-z+1)$-ní nejmenší prvek mezi posledními $n-z-1$ prvky.

Řečeno s panem Pascalem:

```
{potřebujeme přerovnávací funkci, která
 dostane pozici pivota jako parametr}
function prerp(var a:Pole;
               l,r,m:Integer):Integer;
var q:Integer;
begin
 {pivota prohodíme s posledním prvkem}
 q:=a[m]; a[m]:=a[r]; a[r]:=q;
 {a zavoláme původní přerovnávací fci}
 prerp := prer(a,l,r);
end;

{hledání k-tého nejmenšího prvku z a[l..r],}
{vracíme pozici prvku, nikoliv jeho hodnotu}
function kth(var a:Pole; l,r,k:Integer):Integer;
var medp:Pole;          {pole pro mediány pětic}
    i,j,q,x,pocet,m,z:Integer;
begin
 pocet:=r-l+1;        {s kolika prvky pracujeme}

 if pocet<=1 then            {pouze jeden prvek?}
  kth:=l            {výsledek ani nemůže být jiný}
 else if pocet<6 then begin  {méně než 6 prvků}
  for j:=l+1 to r do begin      {=> InsertSort}
   q:=a[j];
   i:=j-1;
   while (i>=l) and (a[i]>q) do begin
    a[i+1]:=a[i];
    Dec(i);
    end;
   a[i+1]:=q;
   end;
  kth:=l+k;
  end
 else begin         {mnoho prvků, jde to tuhého}
  {rozdělíme prvky do pětic}
  q:=1;                {zatím máme jednu pětici}
  i:=l;                {levý okraj první pětice}
  j:=i+4;             {pravý okraj první pětice}
  while j<=r do begin   {procházíme celé pětice}
   medp[q]:=kth(a,i,j,2);         {medián pětice}
   Inc(q);                     {zvyš počet pětic}
   Inc(i,5);            {nastav levý okraj pětice}
   Inc(j,5);           {nastav pravý okraj pětice}
   end;
  if i<=r then begin       {zbyla neúplná pětice}
   medp[q]:=kth(a,i,r,(r-i+2) div 2);
   Inc(q);
   end;

  {najdeme medián mediánů pětic, je na pozici m}
  m:=kth(medp,1,q-1,q div 2);

  {přerovnej a zjisti, kde skončil pivot}
  x:=prer(a,l,r,m);
  z:=x-l+1;           {pozice vzhledem k [l..r]}
  if k=z then
   kth:=m              {k-tý nejmenší je pivot}
  else if k<z then
   kth:=kth(a,l,x-1,k)    {k-tý nejmenší nalevo}
  else
   kth:=kth(a,x+1,r,k-z);               {napravo}
 end;
end;
```

Zbývá dokázat, že tato dvojitá rekurze opravdu má lineární složitost. Zkusme se proto podívat, kolik prvků posloupnosti po přerovnání je větších než prvek m. Všech pětic je $N/5$ a alespoň polovina z nich (tedy $N/10$) má medián menší než m. V každé takové pětici pak navíc najdeme dva prvky menší než medián pětice, takže celkem existuje alespoň $3/10 \cdot N$ prvků menších než m. Větších tedy může být maximálně $7/10 \cdot N$. Symetricky ukážeme, že i menších prvků může být nejvýše $7/10 \cdot N$.

Rozdělení na pětice, hledání mediánů pětic a přerovnávání trvá lineárně, tedy nejvýše cN kroků pro nějakou konstantu $c > 0$. Pak už algoritmus pouze dvakrát rekurzivně volá sám sebe: nejprve pro $N/5$ mediánů pětic, pak pro $\leq 7/10 \cdot N$ prvků před/za mediánem. Pro celkovou časovou složitost $t(N)$ našeho algoritmu tedy platí:

$$t(N) \leq cN + t(N/5) + t(7/10 \cdot N).$$

Nyní zbývá tuto rekurzivní nerovnici vyřešit, což provedeme drobným úskokem: uhodneme, že výsledkem bude lineární funkce, tedy že $t(N) = dN$ pro nějaké $d > 0$. Dostaneme:

$$dN \leq (c + 1/5 \cdot d + 7/10 \cdot d) \cdot N.$$

To platí například pro $d = 10c$, takže opravdu $t(N) = \mathcal{O}(N)$.

Násobení dlouhých čísel

Dalším pěkným příkladem na rozdělování a panování je násobení dlouhých čísel – tak dlouhých, že se už nevejdou do integeru, takže s nimi musíme počítat po číslicích (ať už v jakékoliv soustavě – my volíme desítkovou, často se hodí třeba 256-ková). Klasickým „školním“ algoritmem pro násobení na papíře to zvládneme na kvadratický počet operací, zde si předvedeme efektivnější způsob.

Libovolné $2N$-ciferné číslo mužeme zapsat jako $10^N A + B$, kde A a B jsou N-ciferná. Součin dvou takových čísel pak bude $(10^N A + B) \cdot (10^N C + D) = (10^{2N} AC + 10^N (AD + BC) + BD)$. Sčítat dokážeme v lineárním čase, násobit mocninou deseti také (dopíšeme příslušný počet nul na konec čísla), N-ciferná čísla budeme násobit rekurzivním zavoláním téhož algoritmu. Pro časovou složitost tedy bude platit $t(N) = cN + 4t(N/2)$. Nyní tuto rovnici můžeme snadno vyřešit, ale ani to dělat nebudeme, neboť nám vyjde, že $t(N) \approx N^2$, čili jsme si oproti původnímu algoritmu vůbec nepomohli.

Přijde trik. Místo čtyř násobení čísel poloviční délky nám budou stačit jen tři: spočteme AC, BD a $(A+B) \cdot (C+D) = AC + AD + BC + BD$, přičemž pokud od posledního součinu odečteme AC a BD, dostaneme přesně $AD + BC$, které jsme předtím počítali dvěma násobeními. Časová složitost nyní bude $t(N) = c'N + 3t(N/2)$. (Konstanta c' je o něco větší než c, protože přibylo sčítání a odčítání, ale stále je to konstanta. My si ovšem zvolíme jednotku času tak, aby bylo $c' = 1$, a ušetříme si tak spoustu psaní.)

Jak naši rovnici vyřešíme? Zkusíme ji dosadit do sebe samé a pozorovat, co se bude dít:

$$\begin{aligned} t(N) &= N + 3(N/2 + 3t(N/4)) = \\ &= N + 3/2 \cdot N + 9t(N/4) = \\ &= N + 3/2 \cdot N + 9/4 \cdot N + 27t(N/8) = \ldots = \\ &= N + 3/2 \cdot N + \ldots + 3^{k-1}/2^{k-1} \cdot N + 3^k t(N/2^k). \end{aligned}$$

Pokud zvolíme $k = \log_2 N$, vyjde $N/2^k = 1$, čili $t(N/2^k) = t(1) =$ nějaká konstanta d. To znamená, že:

$$t(N) = N \cdot (1 + 3/2 + 9/4 + \ldots + (3/2)^{k-1}) + 3^k d.$$

Výraz v závorce je součet prvních k členů geometrické řady s kvocientem 3/2, čili $((3/2)^k - 1)/(3/2 - 1) = \mathcal{O}((3/2)^k)$. Tato funkce však roste pomaleji než zbylý člen $3^k d$, takže ji klidně můžeme zanedbat a zabývat se pouze oním posledním členem: $3^k = 2^{k \log_2 3} = 2^{\log_2 n \cdot \log_2 3} = (2^{\log_2 n})^{\log_2 3} = n^{\log_2 3} \approx n^{1.58}$. Konstanta d se nám „schová do $\mathcal{O}$-čka", takže algoritmus má časovou složitost přibližně $\mathcal{O}(n^{1.58})$. Umí se to i lépe – $\mathcal{O}(n \log n)$, ale to je mnohem ďábelštější a pro malá n se to sotva vyplatí.

Program si pro dnešek odpustíme, šetříme ť naše lesy.

Poznámky na ubrousku aneb Rozmyslete si

- Při hledání k-tého nejmenšího prvku jsme předpokládali, že všechny prvky jsou různé. Prohlédněte si algoritmy pozorně a rozmyslete si, že budou fungovat i bez toho. Opravdu?
- Proč jsme zvolili zrovna pětice? Jak by to dopadlo pro trojice? A jak pro sedmice? Fungoval by takový algoritmus? Byl by také lineární?
- Ve výpočtu $t(N)$ jsme si nedali pozor na neúplné pětice a také jsme předpokládali, že pětic je sudý počet. Ono se totiž nic zlého nemůže stát. Jak se to snadno nahlédne? Proč nestačí na začátku doplnit vstup „nekonečny" na délku, která je mocninou deseti?
- Kdybychom neuhodli, že $t(N)$ je lineární, jak by se na to dalo přijít?
- Ještě jednou QS: Představte si, že budujete binární vyhledávací strom vkládáním prvků v náhodném pořadí. Obecně nemusí být vyvážený, ale v průměru v něm půjde vyhledávat v čase $\mathcal{O}(\log N)$. Žádný div: stromy, které nám vzniknou, odpovídají přesně možným průběhům QuickSortu.

Dnešní menu Vám servírovali
David Matoušek & Martin Mareš