do jednoho dlouhatánského čísla a mít vždycky prostorovou složitost konstantní?

T: Máš pravdu, trochu podvod to opravdu je. Když se složitost zavádí pořádně (což už, jak název napovídá, není úplně jednoduché), neměří se prostor v paměťových buňkách, nýbrž přesně v bitech. Velikánská čísla proto zaberou daleko víc místa než malá a to už tak snadno neošidíš. Stejně se pak měří i velikost vstupu (čímž přestanou být problémy s naším příkladem, jelikož ten má pak na vstupu logaritmicky mnoho desítkových číslic) a časová složitost elementárních operací pak není jednotková, nýbrž závisí na počtu bitů čísel, se kterými se právě počítá. Pak do sebe všechno dokonale zapadá a žádné paradoxy nenastávají. Jenže je s tím zase o dost víc práce než předtím. Většinou naštěstí pomůže takový malý úskok: budeme složitosti počítat postaru, jen si zavedeme omezení, že všechna čísla, se kterými pracujeme, musí být polynomy ve velikosti vstupu, čili maximálně n^k pro nějaké k. Pak bude ta „pořádná“ definice složitosti dávat vždy $k \cdot \log n$-krát větší hodnoty než ta „podvodná“, takže vše bude fungovat.

T: (obdivně) Tedy klobouk dolů, Ty toho ale víš... Kde ses to všechno naučil?

T: *(trochu rozpačitě)* Každý začátek je těžký. To základní mně kdysi stejně jako Tobě někdo poradil, zbytek jsem si našel po knížkách (pěkné jsou například Algoritmy a programovací techniky od RNDr. Pavla Töpfera), ve vzorových řešeních KSP a pak na přednáškách na soustředění. Vyplatí se podívat se i do starších ročníků, ty se dají objevit třeba na webu nebo v ročenkách, které KSP každý rok vydává. A když nebudeš vědět, jak dál, klidně se mailem zeptej organizátorů.

T: Tak jo, díky moc, já už musím jít, už se nemůžu dočkat, až napíšu řešení další série.

T: Hodně štěstí, Tomáši. *(pak dodá)* Vím, že to zvládneš.

Opona.

Tips & Tricks: Z letáků KSP si můžete složit knížečku

Korespondenční Seminář z Programování

Ročník osmnáctý, 2005 / 2006

Milí programátoři! Jelikož víme o Vaší touze řešit všelijaké problémy a problémky z oblasti algoritmů a programování, rozhodli jsme se připravit pro Vás další, v pořadí již devatenáctý (první byl totiž nultý) ročník Korespondenčního Semináře z Programování známého spíše pod zkratkou *KSP*, který je určen pro studenty středních i základních škol.

A jakpak takový seminář vlastně probíhá? Jednou za čas od nás poštou dostanete zadání několika (obvykle pěti) úloh, doma je v klidu vyřešíte (samozřejmě nemusíte vyřešit všechny), svá řešení sepíšete (několik rad ohledně sepisování Vašich řešení si můžete přečíst níže) a do určeného termínu zašlete na níže uvedenou adresu. My je poté opravíme a spolu se vzorovými řešeními a výsledkovou listinou pošleme zpět. Tento cyklus se nazývá *série*. Stejně jako v minulém ročníku bude součástí série také tzv. *programátorská kuchařka*, což je kratší povídání o nějakém algoritmu či datové struktuře (nabyté vědomosti můžete často použít při řešení úloh :–)).

Za jeden školní rok obvykle proběhne pět sérií. Poté, co nám pošlete vyřešenou první sérii nebo alespoň přihlášku, obratem Vám odešleme zadání série druhé (čili pokud nedodržíte termín odeslání první série, nepřijde Vám druhá!). Nicméně opravenou první dostanete ještě před tím, než budete muset odeslat druhou, takže se ve svých řešeních druhé série budete moci poučit z výsledků série první.

Závěrečným bonbónkem jednoho ročníku je pak pravidelné *soustředění* nejlepších řešitelů semináře konané obvykle na začátku ročníku dalšího a zahrnující bohatý program čitající jak aktivity ryze odborné (přednášky na různá zajímavá témata apod.), tak aktivity ryze neodborné (kupříkladu hry a soutěže v přírodě). Při vybírání účastníků soustředění ovšem nebude hrát roli jenom dosažený počet bodů, ale také věk a doba, po kterou seminář řešíte.

Úlohy v semináři bývají povětšinou algoritmického rázu – důraz je tedy kladen zejména na nalezení vhodného algoritmu řešícího daný problém a nikoliv na různé triky týkající se obelstění nevýhod toho či onoho počítače. Z toho též plyne, že vyžaduje-li se program, má tento být (pokud ovšem není v zadání uvedeno jinak) v některém z obvyklých vyšších programovacích jazyků (asi nejznámějšími jsou C, C++ a Pascal). Také není potřeba věnovat žádnou zvláštní péči načítání vstupu – můžete předpokládat, že je vždy korektní a že se vždy vejde do paměti, pokud tedy není v zadání řečeno jinak. Neprahneme ani po žádných speciálních efektech a jiných „vylepšeních“. Jde nám hlavně o algoritmy, nikoliv o návrh uživatelských rozhraní a navíc vytváření oken či tlačítek jen snižuje čitelnost zdrojového kódu. Proto okénka v Delphi ani unity `crt` a `graph` v Turbo Pascalu raději nepoužívejte.

Pro každou úlohu je předem stanoven maximální počet bodů, takže například nemáte-li u jedné série čas na vyřešení jedné úlohy, můžete si vybrat tu, za kterou je bodů nejméně. Tu a tam se ale může stát, že někdo dostane více bodů, než je maximum, třeba nalezne-li podstatně lepší řešení, než měl původně na mysli autor úlohy. Hodnotí se zejména:

- *Funkčnost:* tedy zda algoritmus danou úlohu řeší.
- *Efektivita:* jak rychle řešení pracuje a kolik na to spotřebuje paměti; bývá zvykem u každého algoritmu uvádět jeho asymptotickou časovou a paměťovou složitost (viz níže).
- *Přehlednost a kvalita popisu algoritmu:* popis, z něhož není vůbec poznat, jak algoritmus vlastně funguje, je hodnocen stejně jako popis prázdný.
- *Důkaz správnosti:* vedle popisu samého byste též měli více či méně formálně dokázat, že algoritmus se v konečném čase dobere požadovaných výsledků.
- *Popis programu:* není nutno znovu rozebírat, jak program pracuje, neboť to mělo být napsáno již v popisu algoritmu; užíváte-li nějakých méně obvyklých programátorských obratů, je dobré o nich napsat. Rovněž komentáře v textu programu nejsou na škodu.
- *Kvalita programu:* ač nevysokým počtem bodů, přeci jen též bereme ohled na přiměřenost použitých programátorských obratů.

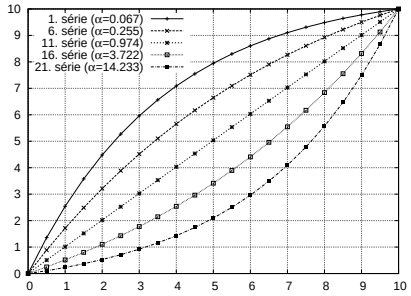
Jelikož každý začátek je těžký, rozhodli jsme se od letošního ročníku mladším řešitelům trochu pomoci. V každé sérii najdete více úloh, od jednoduchých spíše logických hříček až po zapeklité problémy, samozřejmě s patřičně odstupňovaným bodováním. Z úloh, které nám pošlete, si vybereme čtyři nejlépe hodnocené, a než body zapíšeme do výsledkové listiny, přepočítáme je ještě podle následujícího vzorečku:

$$b' = M \cdot \frac{\alpha^{b/M} - 1}{\alpha - 1} \text{ pro } b \le M, \quad b' = b \text{ pro } b > M.$$

Zde je b původní počet bodů, M maximální počet bodů za úlohu, b' nový počet bodů (zaokrouhlíme na jedno desetinné místo) a α konstanta závislá na Vašem služebním stáří, měřeném počtem sérií s, které jste nám odevzdali:

$$\alpha = 25^{\frac{\min(s,23)-11.1}{12}}.$$

Co doopravdy s Vašimi body tato formulka provede, můžete najít v následujícím grafu:



Přihláška do KSP

Jméno a příjmení: .....

Adresa: .....

..... E-mail: .....

Škola: ..... Ročník: .....

Pravidlem číslo jedna nadále zůstává striktní dodržování zásad *fair-play*. Nebudeme se zabývat řešeními, která jsou zjevně opsaná od Vašich spolužáků. Taktéž bychom byli neradi, aby se nám, jako ostatně již několikráte, stávalo, že se úlohy *KSP* stanou náplní středoškolské výuky informatiky. Nemáme nic proti tomu, abyste si úlohy na některé z hodin rozebrali, ale nepovažujeme za vhodné využívat výsledků dosažených v semináři jako kritéria pro hodnocení prospěchu. Nastanou-li nějaké problémy, seznamte své učitele s obsahem tohoto odstavce.

Rozhodnete-li se zapojit do tohoto semináře, tak k první zásilce, již nám zašlete, připojte *přihlášku,* jejíž vzor je uveden níže. Můžete ji poslat jak poštou, tak e-mailem na níže uvedenou adresu. Prosíme o vyplnění Vaší přesné adresy, na kterou si přejete zasílat zadání dalších sérií a ostatní „seminářovou korespondenci“, a to pokud možno včetně poštovního směrovacího čísla. Adresu školy není nutno uvádět, jméno školy slouží pouze pro snazší orientaci ve výsledkové listině. Ročník, který navštěvujete, ovšem můžeme použít při výběru účastníků soustředění; vzhledem ke zmatku v dnešních čtyř- až osmiletých gymnasiích jsme byli nuceni zavést ročník *normalizovaný* odpovídající standardnímu čtyřletému gymnasiu – číslo 4 tedy značí maturitní ročník, 3 první ročník před maturitním atd. Mohou ovšem vznikat i zdánlivě nesmyslná čísla jako -1 (osmý ročník základní školy či tercie osmiletého gymnasia), těch se však není třeba nikterak obávat. Máte-li e-mailovou adresu, uveďte i tu (pokud budete chtít zasílat řešení elektronickou cestou (viz dále), musíme ji znát).

Každá úloha, kterou nám budete posílat, by měla být vypracována na *zvláštním listě* (nebo více listech), nejlépe formátu A4, ježto různé úlohy obvykle opravují různí lidé. První list by měl začínat jednoduchou *hlavičkou* dle následujícího vzoru (všechny údaje nahraďte svými vlastními!):

Tomáš Marný *18-1-1*
4.B, Gymnasium Nezamyslice *2 listy*

(počet listů můžete vynechat, je-li první list zároveň posledním) a listy následující opatřete alespoň číslem listu a pokud možno je připněte sponkou v levém horním rohu k ostatním listům téže úlohy. Posílat diskety s programy nemá smysl – pokud zadání požaduje napsání programu, přiložte jeho výpis k textu řešení. Též si dávejte pozor, abyste své zásilky opatřili patřičným počtem *poštovních známek* – nedoplacené zásilky jsou tradičně honorovány prémií ve výši -10 bodů.

Své přihlášky, vyřešené úlohy, vzkazy pro opravovatele a případné dotazy či připomínky zasílejte na adresu

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

Také můžete pro komunikaci s námi používat Internet: přihlášky a vzkazy organizátorům můžete posílat e-mailem na adresu *ksp@mff.cuni.cz*, řešení úloh je možné odevzdávat na adrese *http://ksp.mff.cuni.cz/submit/*. Ovšem pozor na to, že tohoto způsobu mohou využívat jen „zaregistrovaní“ řešitelé – pokud byste chtěli zaslat řešení elektronicky, pošlete přihlášku mailem (s dostatečným předstihem alespoň týdne). To se týká i řešitelů minulého ročníku.

Zadání, vzorová řešení i výsledkové listiny, archiv předchozích ročníků, fotky ze soustředění a jiné zajímavosti najdete na webových stránkách semináře na *http://ksp.mff.cuni.cz/*.

Rádi bychom Vás ještě upozornili na další zajímavou programátorskou soutěž. Je to kategorie P Matematické olympiády. Úlohy domácího kola získáte ve škole od svých učitelů matematiky nebo informatiky a najdete je také na Webu na *http://mo.mff.cuni.cz/p/*. Vyřešené úlohy domácího kola se odevzdávají ve škole obvykle do konce listopadu. Úspěšní řešitelé tohoto kola postoupí do kola oblastního a nejlepší z nich pak do kola celostátního s možností další účasti v mezinárodních soutěžích – středoevropské a mezinárodní olympiádě.

Hodně štěstí!

(ty stejně závisí na přesné definici operací), takže čas $32n^2$ pro nás bude totéž jako n^2. Dokonce i $t(n) = n^2 + 42n - 5$ je totéž, protože pro $n \geq 42$ je $t(n) \leq 2n^2$. Ovšem $T(n) = n^3/1000$ už roste rychleji, ačkoliv to bude poznat až pro velká n (konkrétně $n > 1000$). Proto pokud je časová složitost polynom, záleží jenom na jeho stupni: rozlišujeme složitost lineární ($c \cdot n$), kvadratickou ($c \cdot n^2$), kubickou ($c \cdot n^3$) atd.

T: To je krásně jednoduché. Žádné další nejsou?

T: Jsou, a kolik! Někdy třeba narazíš na algoritmy, které jsou rychlejší než lineární. Takové si ani nemohou stihnout přečíst celý vstup, ovšem pokud ho dostanou připravený v poli, nemusí to vadit. Třeba vyhledávání půlením intervalů má složitost řádu $\log n$ (logaritmickou; všimni si, že na základu logaritmu nezáleží, protože $\log_a x = \log_b x / \log_b a$, a proto se různé logaritmy liší zase jenom konstanta-krát a my přeci na konstanty nevěříme). Naopak některé želvovité algoritmy oplývají složitostí řádu 2^n (exponenciální), takže už pro docela malá n můžeme na výsledek čekat pár hodin. Často také potkáš složitosti typu $n \cdot \log n$ (tu mají mnohé třídící algoritmy a je mezi n a n^2) nebo $n \cdot \log^2 n$ (ta je mezi $n \cdot \log n$ a n^2).

T: Občas jsem také narazil na cosi jako $\mathcal{O}(n^2)$. Co to je?

T: To je taková zkratka. Aby informatici nemuseli pořád psát „když zanedbáme multiplikativní konstanty, složitost lze shora omezit funkcí řádu n^2“, napíší raději „složitost je $\mathcal{O}(n^2)$“. Pokud to chceš vědět přesně: když napíšeme $f(n) = \mathcal{O}(g(n))$, znamená to, že existuje nějaká konstanta $c > 0$ taková, že pro všechna dost velká n (přesněji od nějakého n_0 dál) je $f(n) \leq c \cdot g(n)$. Šikovná věcička, ale pozor, je trochu nebezpečná. Vypadá totiž jako rovnost, ale nemůžeš obě strany prohodit ($\mathcal{O}(g(n)) = f(n)$ prostě nedává smysl) a navíc je to vlastně nerovnost – pokud napíšu $f(n) = \mathcal{O}(n^3)$, říkám tím pouze, že (až na konstanty atd.) f roste nejvýše jako n^3 – ve skutečnosti může růst mnohem pomaleji. Čili i o lineárním algoritmu můžeme říci, že má složitost $\mathcal{O}(n^3)$ – bude to pravda, ale moc trefné to není. Pro úplnost dodávám, že existuje podobná značka Ω pro opačnou nerovnost, takže to, že nějaký algoritmus má složitost alespoň kvadratickou, mohu také napsat jako $\Omega(n^2)$.

T: Jakou má tedy složitost ten můj algoritmus?

T: Tahle úloha zrovna do našeho způsobu měření velikosti vstupu moc nezapadá, protože vstupem je jediné číslo a doba výpočtu závisí jen na jeho hodnotě. Proto budeme časovou složitost studovat raději jako funkci té hodnoty.

T: (vítězoslavně) Takže je logaritmická!

T: Málem. For-cyklem opravdu projdeš $\mathcal{O}(\log n)$-krát (jednou pro každou číslici výsledku a těch je $1 + \lfloor \log_2 n \rfloor$). Jenže jelikož si číslice skládáš do stringu, nesmíš zapomenout na to, že operace se stringem, třeba takové přiřazení stringů, nemá konstantní časovou složitost, nýbrž lineární v délce stringu. První přiřazení tedy bude trvat čas c, druhé $2c$, třetí $3c$ až poslední řádově $\log n \cdot c$. To je dohromady $\mathcal{O}(\log^2 n)$.

T: To je zrada! Jak můžu poznat, co trvá jak dlouho?

T: Pokud používáš jednoduché operace (toho druhu, jaké jsme před chvilkou považovali za elementární), můžeš se spolehnout na to, že trvají konstantně dlouho, čili $\mathcal{O}(1)$. Ale jakmile začneš využívat nějakých pokročilejších funkcí svého oblíbeného programovacího jazyka, už to začne být složitější. Tehdy je nejlepší představit si, jak taková funkce uvnitř funguje, a podle toho určit složitost. Možná bude implementována chytřeji a rychleji, než to dovedeš ty, ale málokdy hůře. A pokud si nedokážeš fungování takové funkce představit, je lepší se jí obloukem vyhnout.

T: A nešlo by tu úlohu vyřešit rychleji? Jak bys to napsal Ty?

T: Určitě šlo, stačí si všimnout toho, že namísto skládání čísel do stringu bys je mohl skládat do pole předem známé délky ... nebo ještě lépe obrátit směr běhu cyklu a jednotlivé číslice vypisovat průběžně. Vypadalo by to asi takhle (rovnou Ti ukážu, jak k tomu napsat popis algoritmu):

Úlohu budeme řešit tak, že si všimneme, že n mod 2 je rovno poslední číslici dvojkového zápisu čísla n a celočíselným dělením čísla n číslem 2^i odřízneme i posledních dvojkových číslic. Proto (n div 2^i) mod 2 dává i-tou číslici dvojkového zápisu. Tento algoritmus má časovou složitost $\mathcal{O}(\log n)$ (která je optimální, protože rychleji nelze ani vypsat výstup) a paměťovou složitost $\mathcal{O}(1)$.

```
var x,n:integer;
begin
   read(n);
   x := 1;
   while 2*x<=n do x:=2*x;
   { x je nejbližší nižší mocnina 2 }
   while x>0 do begin
      write(n div x mod 2); { číslice řádu x }
      x := x div 2;
   end;
   writeln;
end.
```

T: To je krásně jednoduché!

T: To je. Většina úloh z KSP-čka se dá vyřešit na pár řádků. Samozřejmě napsat opravdu hezké a krátké řešení je velké umění a asi Ti ještě bude nějaký čas trvat, než se to naučíš (asi tomu nebudeš věřit, ale i autoři vzorových řešení často celý program několikrát přepisují, než se jim začne líbit). Ovšem pokud Tvé řešení má tisíc řádků a je plné objektů a volání exotických knihoven, skoro určitě je od správného na hony daleko a vysloužíš si za něj spíš než body další hrst sarkastických poznámek.

T: Když už jsi zmínil prostorovou složitost, ta se měří jak?

T: Prostorová (jinak také paměťová) složitost algoritmu udává množství paměti, které algoritmus spotřebuje, opět v závislosti na velikosti vstupu. Obvykle se do ní nepočítá velikost vstupu a výstupu, pokud algoritmus vstup pouze čte a do vstupu pouze zapisuje. Většinou se měří v paměťových buňkách, přičemž každá buňka pojme jedno číslo, jeden znak nebo jeden ukazatel (string už zabere tolik buněk, kolik je jeho délka, plus jednu buňku navíc na uchování té délky).

T: (s úsměvem takřka ďábelským) Ha! Takže skoro všechna vzorová řešení KSP vlastně mají konstantní časovou i prostorovou složitost, protože mají velikost vstupu omezenou nějakou konstantou.

T: Ne tak zhurta. Technicky vzato, máš pravdu. Ale ty konstanty tam jsou jenom proto, že dynamické alokování polí nebo seznamů podle skutečné velikosti vstupu je otročina, na které většinou není nic zajímavého a každý si ji snadno domyslí. Takže tam raději napíšeme konstantu, aby v programu bylo dobře vidět na ty opravdu důležité části.

T: A není to s tou prostorovou složitostí trochu podvod? Nemohl bych si třeba všechny proměnné zakódovat po bitech

```
uses crt;
var cislo,index:integer;  { číslo a index }
      vysl:string;        { výsledek }
begin
repeat
clrscr; textcolor(red);   { smažeme obrazovku }
write('Zadej cislo: '); textcolor(white);
read(cislo);              { přečteme číslo }
until cislo>=0; { nedáme pokoj než zadá kladné }
for index:=0 to trunc(ln(cislo)/ln(2)) do begin
   r:=trunc(exp(ln(2)*index)); { mocnina }
   if (cislo div r) mod 2 = 0 then
      vysl := '0'+vysl    { výsledek je 0 }
   else vysl := '1'+vysl; { jednička }
   end;                   { konec cyklu }
write(vysl);              { výsledek }
end.
```

T: *(zděšeně)* To tedy zírám. Uff. Určitě jsi to ani nevyzkoušel ... vlastně ani nezkompiloval.

T: (udiveně) No, nevyzkoušel. Ale jaks' to poznal?

T: Předně: zkompilovat by Ti to ani nešlo, protože proměnná `r` není nikde deklarovaná. Ale ani pak to nebude fungovat, zapomínáš totiž inicializovat `vysl`.

T: Dobře, to jsou přeci maličkosti. Hlavní je, že to funguje.

T: Jenže nefunguje. Třeba pro nulu nastane běhová chyba, protože z nuly neexistuje logaritmus, a pro mocniny dvojky to také většinou nevyjde, protože kvůli zaokrouhlovacím chybám bude podíl logaritmů o maličko menší než celé číslo, a tak ho `trunc` ořízne na o 1 méně, než je správně, a zapomeneš vypsat první číslici. Zkrátka pokud opravdu nevíš, co děláš, je daleko lepší s celočíselnými vstupy zacházet jenom celočíselně, žádný `real`.

T: To je fakt. Ale na to mně ani neupozornili. Zato to žalostné zavytí na začátku programu ... Co tím vlastně chtěli říct?

T: Že je zbytečné věnovat tolik místa v programu všelijakému pozlátku, jako je třeba mazání obrazovky nebo barvičky. Angličané tomu výstižně říkají „bells and whistles". Všechno to jsou věci, které by skutečného uživatele možná potěšily, ale v KSP jsou spíš na obtíž, protože se mezi nimi ztrácí to, o co opravdu jde. Stejně tak není potřeba testovat korektnost vstupu.

T: A co znamenají ty otazníky u komentářů? Vždyť jsem s nimi dal takovou práci.

T: Ono jich sice je spousta, ale mají jednu vadu: jsou úplně k ničemu. Kterým endem končí který cyklus, každý vidí (či spíš by viděl, kdybys program rozumně odsazoval), stejně tak, že `read` čte číslo ze vstupu. A popis v úvodu je úplně totéž. Ovšem naproti tomu třeba není zřejmé, co znamenají ty kejkle s logaritmy a exponenciálami, a ty jsi vůbec neokomentoval.

T: (trochu zmateně) A co tam tedy mám psát?

T: Zkus začít tím, že vysvětlíš, jakou základní myšlenku Tvůj algoritmus má, případně dokážeš, že opravdu funguje (pokud to není zřejmé), a program pak napíšeš co nejpřímočařeji – když se budeš držet popisu algoritmu a proměnné budeš pojmenovávat výstižně, hned bude vidět, co program dělá, i když v něm moc komentářů nebude. Výstižně ovšem vůbec nemusí znamenat dlouze – pro obyčejnou indexovací proměnnou je opravdu nejlepší název *i*. Komentuj jenom místa, která nejsou všem jasná, kde třeba používáš nějaký trik. Zkus si třeba představit, že chceš svému kamarádovi, vysvětlit, jak program funguje. A také si to po sobě nezapomeň pozorně přečíst, gramatické a pravopisné chyby v češtině jsou úplně stejně špatně, jako syntaktické chyby v programu.

T: Jak Tě tak poslouchám, to by tam ani ten program nemusel být, ne?

T: Máš pravdu, nemusel. Kdybys popis algoritmu napsal tak, že si každý dokáže detailně představit, jak by program vypadal, je program opravdu zbytečný. Jenže málokdy poznáš s jistotou, že to tak je, takže je lepší programy pokaždé psát, třeba tím usnadníš práci někomu, kdo se bude už druhou hodinu marně snažit Tvůj popis algoritmu pochopit. A nakonec i Tebe psaní a ladění programu donutí všechno si pořádně rozmyslet, zvlášť různé okrajové případy, jako třeba ty nuly.

T: A když používám nějaký standardní algoritmus?

T: Pokud je dobře známý (třeba proto, že byl v minulé kuchařce), tak ho samozřejmě podrobně vysvětlovat nemusíš. Stačí, když řekneš, jaký algoritmus použiješ a kde se dá najít. I v programu ho pak můžeš vynechat (byť se tím připravíš o možnost ladění).

T: OK, a co ta poznámka o složitosti? Tu už mi tam psali minule, tak jsem si to přesně změřil a opravdu to vždycky doběhne do sekundy. Co to tedy ta časová složitost je a k čemu slouží?

T: Prakticky každá úloha se dá řešit mnoha různými algoritmy. A jak už to chodí, nejsou všechny stejně dobré – často se liší tím, že některý pro nějaký vstup doběhne ihned, zatímco jiný se pro tentýž vstup loudá želvím krokem celé roky. Pravda, oba nakonec vydají správný výsledek, ale asi budeš souhlasit, že ten druhý je vcelku nanic.

Takže když vymyslíš nějaké řešení, měl by ses hned sám sebe zeptat, jak je rychlé, čili jak dlouho pro který vstup poběží. Obvykle doba běhu nijak zvlášť nezávisí na konkrétních číslech na vstupu, jenom na jejich počtu (velikosti vstupu). Také nemá smysl měřit se stopkami v ruce časy v sekundách, už proto, že by pro různé počítače vycházely naprosto různě. Takže si raději zvolíme nějaké elementární operace a za časovou složitost prohlásíme funkci, která nám pro každou velikost vstupu řekne, kolik operací program spotřebuje pro vstupy této velikosti. A kdyby se to pro různé vstupy téže velikosti lišilo, prostě si z časů pro danou velikost vybereme maximum.

T: Hmm, a co to přesně jsou ty elementární operace?

T: Běžné aritmetické operace – sčítání, odčítání, násobení, porovnávání; také základní řídící konstrukce, jako jsou třeba skoky a podmíněné skoky. Zkrátka to, co normální procesor zvládne jednou nebo několika instrukcemi. *(šeptem)* Ale psst, ono to je tak trochu jedno – ať si vymyslíš libovolnou rozumnou sadu operací, stejně se počet operací v programu změní jen konstanta-krát a na tom, jak za chvíli uvidíme, moc nezáleží. Ovšem to *rozumnou* je důležité, nesmíš si za základní operaci zvolit třeba zkopírování celého pole nebo dokonce jeho setřídění, to by Ti nikdo neuvěřil.

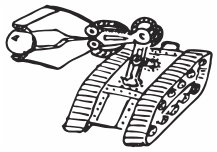
T: (udiveně) A není strašně těžké spočítat, kolik těch operací bude?

T: Kdybys to chtěl spočítat přesně, tak to opravdu těžké bude. Ale ono málokdy záleží na přesném počtu, obvykle se zajímáme jenom o to, jak rychle ten počet roste s velikostí vstupu. Všelijaké konstanty přitom budeme ignorovat

Svá řešení nám zasílejte do 17. října 2005 buďto elektronicky na *http://ksp.mff.cuni.cz/submit/*, nebo klasickou poštou na adresu:

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

18-1-1 Dimenze X — 5 body



Vědci z ústavu teoretické fyziky MFF UK si usmysleli, že zachrání svět a jednou provždy zatočí s tajemnou a nebezpečnou Dimenzí X. Sestrojili přístupový portál a vyslali do něj dva roboty, z nichž každý nese slušný náklad plutonia. Jistě si sami dovedete představit za jakým účelem a sami tušíte, že oba kusy plutonia je potřeba dát na místě k sobě.

Jaké však bylo překvapení vědců, když po odeslání robotů zjistili, že Dimenze X je pouze jednorozměrná, tedy obyčejná přímka, navíc táhnoucí se prokazatelně od severu k jihu. Do výpočtů se navíc vloudila chybička a roboti dopadli na naprosto neznámé místo na přímce, každý někam jinam. Na obou místech přistání zůstala po robotech hromada šrotu, která z nich nárazem odpadla, mimo jiné komunikační a senzorová jednotka. Paměťový obvod se také částečně poškodil, kompas naštěstí vydržel. To znamená, že roboti se spolu nemůžou nijak na dálku domluvit a jejich setkání se tak značně komplikuje. Ale Vy to přeci tak nenecháte!

V každém kroku se robot může posunout buďto o 1 metr na sever nebo o 1 metr na jih. Robot naštěstí pozná, že právě přechází přes hromadu šrotu (ale už nepozná, jestli přes svou, nebo druhého robota). Kromě nákladu plutonia (který pochopitelně nesmí zahodit) už neunese vůbec nic dalšího. Paměťový obvod je poškozený, takže čím méně si toho bude robot muset pamatovat, tím lépe. Navrhněte nouzovou posloupnost povelů – tedy vlastně algoritmus (programovat ho ale nemusíte) – pro roboty takovou, aby do sebe roboti na přímce zaručeně po nějaké době narazili. Ale pozor, oba dva roboti dostanou jednu a tutéž posloupnost povelů.

18-1-2 Úřad — 6 bodů

Bylo nebylo, na jistém nejmenovaném úřadě jistého nejmenovaného města v jisté nejmenované republice byli velmi líní úředníci. Lidé přicházeli se svými žádostmi, přiznáními, potvrzeními, stížnostmi a dalšími písemnostmi za úředníky, kteří s otráveným výrazem vyslechli, co od nich kdo chce. Potom se slovy „Náš úřad to přezkoumá" spis založili do šanonu, aby ho z něj už nevytáhli až do skonání světa. Ale to jim nestačilo a po čase začali zařazovat i samotné šanony a pořadače do jiných šanonů, pořadačů, fasciklů, s deskami různých barev, a takové objemné svazky potom ukládali do police.

Jednou přišel do úřadu kontrolor. Ne ovšem proto, aby zjistil, jestli úředníci dobře vyřizují žádosti, nýbrž jestli správně archivují spisy. Stoupnul si k polici se spisy a ze zadní strany postavil nervózního ředitele, aby zleva doprava četl barvy desek a jestli je to otevírací deska nebo uzavírací. Úkolem kontrolora je zjistit, jestli jsou šanony do sebe dobře zabalené. Očíslujme si jednotlivé barvy a pokud je před číslem minus, je to deska uzavírací, jinak otvírací. Například tohle zjevně nejsou dobře zabalené desky: 4, 5, −4. Stejně tak tyto: 7, 8, −7, 8. Nicméně proti těmto nelze nic namítat: 1, 5, 6, −6, −5, −1, 6, −6.

Pomozte kontrolorovi zdeptat pana ředitele! Napište program, který odpoví, jestli desky jsou či nejsou dobře zabalené. Program dostane na vstupu přirozené číslo K, což je maximální počet barev šanonů, a počet nahlášených desek N (otevíracích i uzavíracích dohromady). Poté následuje N čísel c, $1 \leq |c| \leq K$ udávajících barvy šanonové desky, kladné číslo c značí otevírací desku barvy c, číslo $-c$ uzavírací desku barvy c. Program by měl vypsat ano nebo ne podle toho, jestli jsou desky správně zabalené. Jinými slovy, pokud si různé typy šanonů představíme jako různé typy závorek, pak je Vaším úkolem otestovat, zda je zadaná posloupnost závorek správně uzávorkovaná.

Příklad: Pro $K = 3$, $N = 8$ a desky $2, 1, 2, -2, -1, -2, 3, -3$ by měl Váš program vypsat odpověď *ano*, zatímco pro desky $1, 2, 3, 1, -1, -2, -3, -1$ je správná odpověď *ne*.

18-1-3 Keřík — 10 bodů

Housenka běláska je neuvěřitelně žravý tvor. Když se jednou tak potulovala po zahrádce plné salátu, kedluben, řepy a spousty dalších laskomin, narazila na velmi chutně vypadající keř.

Keř sestává z význačných bodů, na kterých rostou šťavnaté lístky, a větví mezi nimi, kde neroste nic. U každého bodu je zadáno, kolik lístků na něm roste, a seznam jiných význačných bodu, do kterých z něj vede větev. Dále víte, že keř je zkrátka strom, a tedy mezi každými dvěma význačnými body vede právě jedna cesta.

Housenka by ráda začala svou hostinu v nějakém význačném bodu keře a postupně se po větvích přesouvala na jiné body, nikdy však do místa, kde už jednou byla. A chtěla by se co nejvíce najíst. Pomozte jí a napište program, který k zadanému keři najde nejvýživnější cestu v něm, tedy cestu mezi nějakými dvěma body, na které se neopakuje žádný význačný bod a která obsahuje největší možný počet lístků.

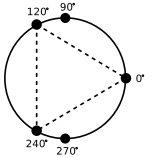
Program dostane na vstupu číslo N, což je počet význačných bodů, a N řádků popisujících jednotlivé vrcholy (očíslujeme si je 1 až N). První číslo na i-tém řádku udává počet lístků, druhé číslo počet větví v vedoucích z i-tého bodu, načež následuje v čísel udávajících, kam ty větve vedou. Program by měl vypsat na výstup nejvýživnější cestu, pokud je takových cest více, tak libovolnou z nich.

Příklad: Pro čtyři význačné body, které obsahují po řadě 1, 2, 3 a 4 lístečky, a pro větvičky $1 \leftrightarrow 2, 2 \leftrightarrow 3, 2 \leftrightarrow 4$ je nejvýživnější cesta $3 \rightarrow 2 \rightarrow 4$ s 9 lístečky.

18-1-4 Dortík — 11 bodů

Klub Sběratelů Pamětin právě slaví své K-té narozeniny. Proto jeho členové vytáhli ze svých nekonečných sbírek narozeninový dort s N svíčkami (jiný zrovna nedokázali najít). Rádi by na něm zapálili právě K svíček a jelikož sběratelé pamětin mají velmi vytříbené estetické cítění, musí tyto svíčky ležet ve vrcholech pravidelného K-úhelníku.

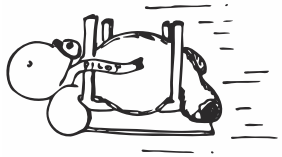
Napište pro ně program, který na vstupu dostane čísla N a K a dále polohy všech N svíček v libovolném uspořádání. Všechny svíčky leží na kružnici, takže jejich polohu můžeme jednoznačně popsat úhlem, který svírá spojnice středu kružnice a svíčky s osou x. Výsledkem programu by měl být nalezený pravidelný svíčkový K-úhelník, případně zpráva, že žádný takový neexistuje.



Příklad: 5 svíček na pozicích 0°, 240°, 270°, 120° a 90° obsahuje pravidelný svíčkový trojúhelník, ale neobsahuje pravidelný svíčkový čtyřúhelník.

18-1-5 Matlalové — 15 bodů

A nyní zprávy ze Země: „Institut SETI konečně potvrdil existenci mimozemského života! Frakce Marťanů a létajících talířů (MALTAL) oznámila zachycení vysílání potvrzující existenci skutečných Marťanů, kteří používají opravdové létající talíře!“ „Ty Matlalové jsou ale natvrdlí, jaké létající talíře? Budeme tam muset zaletět a objasnit to!“



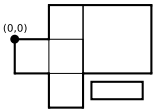
Na Zemi přistáli mimozemšťani přímo na zahradě před frakcí MALTAL. „Marťanové!“ „Matlalové! My vůbec nepoužíváme létající talíře! Dříve jsme létali na létajících kobercích, ale protože se na nich není jak držet, často jsme z nich ve vesmíru padali, a tak nyní používáme létající stoly!“

Na zahradě opravdu stály létající stoly, některé dokonce stály na ostatních (při pohledu shora se překrývaly). Ovšem díky marťanskému způsobu navigace byly hrany všech stolů rovnoběžné se světovými stranami. Protože se už ale seběhl dav, který si chtěl Marťany a jejich dopravní prostředky prohlédnout, rozhodli se Matlalové postavit kolem létajících stolů plot. Protože se ale některé ze stolů překrývají, není vůbec jasné, jak by měl být dlouhý. A protože Vás Matlalové už znají (vědí, jak jste zatočili s Dimenzí X), požádali Vás o pomoc.

Program dostane na vstupu N, počet marťanských létajících stolů, a jejich jednotlivé polohy. Každý marťanský stůl je zadán pomocí svého „levého horního“ rohu (při pohledu shora) a svou délkou a šířkou v metrech, přičemž jednotlivé stoly se mohou překrývat. Vaším úkolem je říci, kolik metrů plotu bude třeba k oplocení území, na kterém marťanské stoly přistály. Plot musí oplotit všechny zadané stoly, musí vést vždy po jejich hranici, ale nikdy nesmí vést uvnitř nějakého létajícího stolu (matematicky řečeno chcete zjistit obvod sjednocení všech létajících stolů). Počítejte s tím, že jak souřadnice, tak délky a šířky marťanských stolů mohou být reálná čísla.

Příklad: Pro $N = 4$ a létající stoly

- levý horní roh $(0, 0)$, délka 20 m a šířka 10 m,
- levý horní roh $(10, 10)$, délka 10 m a šířka 30 m,
- levý horní roh $(20, 10)$, délka 20 m a šířka 20 m,
- levý horní roh $(22.5, -12.5)$, délka 15 m a šířka 5 m,



je hledaná délka plotu 180 m.

18-1-6 Kompilované komplikátory — 11 bodů

V letošním ročníku seriálu si budeme povídat o kompilátorech. Samozřejmě toho nestihneme příliš mnoho (o kompilátorech existuje několik tlustých knih a stovky článků), ale měli bychom získat alespoň obecnou představu o tom, jak kompilátory fungují.

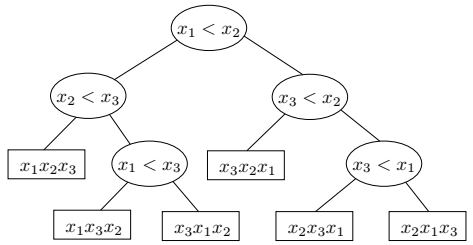
Začněme trochou historie. Před dávnými časy se počítače programovaly přímo ve strojovém kódu – tj. programátor psal (nebo děroval) řady čísel. Napsat takto jakýkoliv rozsáhlejší program bylo samozřejmě velmi obtížné, o hledání a opravování chyb ani nemluvě. Proto se brzy objevily nástroje, které tuto činnost usnadňovaly – assembler, který alespoň umožňoval zapisovat instrukce v čitelnější formě a pojmenovat si proměnné, a později i vyšší programovací jazyky.

Jedním ze zásadních problémů překladačů programovacích jazyků byla rychlost (tedy spíše pomalost) jimi produkovaného kódu. Nějakou dobu trvalo, než optimalizace, které překladače provádí, dosáhly takové úrovně, aby výsledný kód byl srovnatelný s tím, co napíše programátor v assembleru. Po docela dlouhou dobu strašily v programátorských učebnicích poučky typu „místo $3 * x$ pište $x + x + x$“, které měly kompilátoru zabránit, aby použil „drahou“ instrukci na násobení. Dnes již naštěstí takovéto obludnosti nejsou potřeba (a pouze učiní program těžším ke čtení) – dost často jsou přímo v assembleru napsané programy pomalejší než zkompilované.

Většina seriálu bude věnována tomu, jakými optimalizacemi se tohoto výsledku dosáhne. Nicméně nejprve bychom si měli popsat, jak kompilátor vypadá a s čím vlastně pracuje. Níže popsané schéma kompilátoru bylo obecně přijato a v nějaké podobě ho používají zřejmě všechny v současnosti existující kompilátory.

Prvním krokem při kompilaci je lexikální analýza. Jejím úkolem je vstup (což je nějaká posloupnost znaků) převést na posloupnost objektů, které nesou nějaký smysl. Například, pokud je na vstupu řetězec „`bla = bla + 42`“, pro překladač je nezajímavé, že je to znak 'b', následovaný znakem 'l', atd. Výsledek lexikální analýzy by nám v tomto případě řekl, že na vstupu je identifikátor proměnné (`bla`), následovaný operátorem přiřazení, dalším identifikátorem, operátorem sčítání, a číslem (42). Výstupem tedy je posloupnost tzv. *tokenů*. Každý z nich má typ udávající, o jaký objekt se jedná (zda to je identifikátor, číslo, atd.), a případně hodnotu (název identifikátoru, hodnota čísla, ...).

činnost algoritmu si můžeme popsat tzv. *rozhodovacím stromem*. Zde je příklad rozhodovacího stromu pro tříprvkové pole:



Každý vrchol obsahuje porovnání dvou prvků x a y, v levém podstromu daného vrcholu je činnost algoritmu pokud $x < y$, v pravém podstromu činnost při $x \geq y$. V listech je už jisté správné pořadí prvků.

Každému algoritmu odpovídá nějaký rozhodovací strom a každý průběh činnosti algoritmu odpovídá průchodu rozhodovacím stromem od kořene do nějakého listu. Naším cílem bude ukázat, že v libovolném rozhodovacím stromu (a tedy i libovolném odpovídajícím algoritmu) bude existovat cesta z kořene do nějakého listu (neboli výpočet algoritmu) délky $N \log N$.

Kolik maximálně hladin h, a tedy i jaká nejdelší cesta se v takovém stromu může vyskytnout? Náš strom má tolik listů, kolik je možných pořadí tříděných prvků, tedy právě $N!$. Různým pořadím totiž musí odpovídat různé listy, jinak by algoritmus netřídil (předpokládáme přeci, že to, jak má prvky prohazovat, může zjistit jenom jejich porovnáváním), a naopak každé pořadí prvků jednoznačně určuje cestu do příslušného listu. Na nulté hladině je jediný vrchol, na každé další hladině se oproti předchozí počet vrcholů nejvýše zdvojnásobí, takže na i-té hladině se nachází nejvýše 2^i vrcholů. Proto je listů stromu nejvýše 2^h (některé listy mohou být i výše, ale za každý takový určitě chybí jeden vrchol na h-té hladině). Z toho víme, že platí:

$$2^h \geq \textit{počet listů} \geq N!,$$

a proto:

$$h \geq \log_2(N!).$$

Logaritmus faktorálu se těžko počítá přesně, ale můžeme si ho zdola odhadnout pomocí následující známé nerovnosti:

$$n^n \geq n! \geq n^{n/2}.$$

Dosazením získáme:

$$h \geq \log_2(N!) \geq \log_2(N^{N/2}) \geq \frac{N}{2} \log_2 N.$$

Vidíme tedy, že pro každý třídící algoritmus existuje vstup, na kterém se bude muset provést alespoň $N \log N$ kroků.

Poznámky na okraj:

- Zkuste si též rozmyslet (drobnou modifikací předchozího důkazu), že ani *průměrný* čas třídění nemůže být lepší než $N \log N$.
- Odvodit průměrnou složitost QuickSortu vlastně není zase tak těžké. Zkusme následující úvahu: Pokud by pivot nebyl přesně medián, ale alespoň se nacházel v prostřední třetině setříděného úseku, byla by složitost stále $\mathcal{O}(N \log N)$, jen by se zvýšila konstanta v $\mathcal{O}$-čku. Kdybychom pivot volili náhodně, ale po rozdělení prvků si zkontrolovali, jestli pivot padl do prostřední třetiny, a pokud ne (jeden z úseků by byl moc velký a druhý moc malý), volbu bychom opakovali, v průměru by nás to stálo konstantní počet pokusů (pozorování z řešení úlohy 16-1-5: pokud čekáme na událost, která nastává náhodně s pravděpodobností p, stojí nás to v průměru $1/p$ pokusů; zde je $p = 1/3$), takže celková složitost by v průměru vzrostla jen konstantně. Původní QuickSort sice žádné takové opakování volby neprovádí a rovnou se zavolá rekurzivně na velký i malý úsek, ale opět se po v průměru konstantně mnoha iteracích velký úsek zredukuje na nejvýše 2/3 původní velikosti a třídění malých úseků jednotlivě nezabere víc času, než kdyby se třídily dohromady.
- Kdybychom u QuickSortu použili rekurzivní volání jen na menší interval, zatímco ten větší bychom obsloužili přenastavením proměnných a skokem na začátek právě prováděné procedury, zredukovali bychom paměťovou složitost na $\mathcal{O}(\log N)$, jelikož každé další rekurzivní volání zpracovává alespoň dvakrát menší úsek než to předchozí. Časové složitosti tím však nepomůžeme.
- Počet přihrádek u RadixSortu vůbec nemusí být konstanta – pokud např. chcete třídit N čísel v rozsahu $1 \ldots N^k$, stačí si zvolit $\ell = N$ a fází bude jenom k. Pro pevné k tak dosáhneme lineární časové složitosti.
- Nerovnost $n! \geq n^{n/2}$, kterou jsme použili v dolním odhadu složitosti třídění, můžeme dokázat snadným trikem: $n! = \sqrt{1^2 \cdot 2^2 \cdot \ldots \cdot n^2} = \sqrt{n \cdot 1} \cdot \sqrt{(n-1) \cdot 2} \cdot \ldots \cdot \sqrt{2 \cdot (n-1)} \cdot \sqrt{1 \cdot n} \geq \sqrt{n} \cdot \sqrt{n} \cdot \ldots \cdot \sqrt{n} = (n^{1/2})^n = n^{n/2}$. Pokud nevidíte, proč $\geq$, uvažte, že výraz pod odmocninou je tvaru $(n-k)(k+1) = nk + n - k^2 - k = n + k(n-k-1)$ a poslední závorka je pro $0 \leq k < n$ a $n \geq 1$ vždy nezáporná.

Dnešní menu Vám servírovali
Tomáš Valla, Martin Mareš a Dan Kráľ

Jak se stát vítězem KSP za 10 minut, jinak též F.A.Q.

Osoby na scéně:

T: Tomáš Marný, začínající účastník KSP. Právě sedí u počítače v městské knihovně, jehož kabely se ztrácejí kdesi mezi regály. Za okny je ospalé podzimní odpoledne.

T: Tomáš Marný, dlouholetý organizátor KSP. Tentýž počítač v téže knihovně, totéž ospalé odpoledne, jen o 10 let později. Tedy i tentýž Tomáš, ale ještě o tom neví.

Začínáme ...

T: (vztekle odhazuje svazek řešení, kterým probleskují pestrobarevné poznámky opravovatelů, a nemaje si komu postěžovat, bezmyšlenkovitě ťuká do klávesnice) Sakryš! Už zase jenom dva body, to mi ti mizerové snad dělají naschvál, vždyť to přece mám úplně správně!

T: *(odkládá knížku o geometrii L-prostoru a jelikož začíná tušit, co se děje, opatrně vyťukává odpověď)* Ahoj Tomáši, tady je také Tomáš. Asi mne ještě neznáš, i když já tebe znám docela dobře. Oni ti organizátoři jsou docela svérázní lidé, jenže dva body za dobré řešení by nedal ani jeden z nich. Pojď se na to podívat. Co jsi jim to vlastně poslal?

T: (trochu udiveně, protože nečekal, že mu na jeho výkřik někdo odpoví, a ještě méně, že by jeho problémy někoho zajímaly) No, byla to taková děsně jednoduchá úloha s převodem čísla do dvojkové soustavy. Hned mi bylo jasné, jak na to. Co říkáš tomuhle:

```
{ Převod čísel do dvojkové soustavy. Funguje
  jednoduše: přečte si číslo a pak ho vypisuje
  ve dvojkové soustavě. Doběhne do sekundy. }
```

Je však třeba si pamatovat, že pokud se pivot volí náhodně, může rekurze dosáhnout hloubky N a časová složitost algoritmu až $\mathcal{O}(N^2)$ – představme si, že se pivot v každém rekurzivním volání nešťastně zvolí jako největší prvek z tříděného úseku. V naší implementaci QuickSortu nebudeme pivot volit náhodně, ale vždy použijeme prostřední prvek tříděného úseku.

```
procedure QuickSort(var A:Pole; l,r:integer);
var i,j,k,x: integer;
begin
  i:=l; j:=r;
  k:=A[(i+j) div 2];
  repeat
    while A[i]<k do i:=i+1;
    while A[j]>k do j:=j-1;
    if i<=j then
      begin
        x:=A[i]; A[i]:=A[j]; A[j]:=x;
        i:=i+1;
        j:=j-1;
      end;
  until i >= j;
  if j>l then QuickSort(A, l, j);
  if i<r then QuickSort(A, i, r);
end;
```

Ještě si předvedeme dva třídící algoritmy, které jsou vhodné, pokud tříděné objekty mají některé další speciální vlastnosti. Prvním z nich je *třídění počítáním (CountSort)*. To lze použít, pokud tříděné objekty obsahují pouze klíče a možných hodnot klíčů je málo. Tehdy stačí si spočítat, kolikrát se který klíč vyskytuje, a místo třídění vytvořit celé pole znovu na základě toho, kolik jednotlivých objektů obsahovalo pole původní. My si tento algoritmus předvedeme na příkladu třídění pole celých čísel z intervalu $\langle D, H \rangle$:

```
const D = 1;
      H = 10;
procedure CountSort(var A: Pole);
var C: array[D..H] of integer;
    i,j,k: integer;
begin
  for i:=D to H do C[i]:=0;
  for i:=1 to N do C[A[i]]:=C[A[i]] + 1;
  k:=1;
  for i:=D to H do
    for j:=1 to C[i] do
      begin
        A[k]:=i;
        k:=k+1;
      end;
end;
```

Časová složitost takovéhoto algoritmu je lineární v N, ale nesmíme zapomenout přičíst ještě velikost intervalu, ve kterém se prvky nacházejí ($K = H - D + 1$), protože nějaký čas spotřebujeme i na inicializaci pole počítadel. Celkem tedy $\mathcal{O}(N + K)$.

Pokud by tříděné objekty obsahovaly vedle klíčů i nějaká data, můžeme je místo pouhého počítání rozdělovat do přihrádek podle hodnoty klíče a pak je z přihrádek vysbírat v rostoucím pořadí klíčů. Tomuto algoritmu se říká *přihrádkové třídění (BucketSort)* a my si popíšeme jeho víceprůchodovou variantu *(RadixSort),* která je vhodnější pro větší hodnoty K. V první fázi si čísla rozdělíme do přihrádek (skupin) podle nejméně významné cifry a spojíme do jedné posloupnosti, v druhé fázi čísla roztřídíme podle druhé nejméně významné cifry a opět spojíme do jedné posloupnosti, atd. Je důležité, aby se uvnitř každé přihrádky zachovalo pořadí čísel v posloupnosti na začátku fáze, tj. posloupnost uložená v každé přihrádce je vybranou podposloupností posloupnosti ze začátku fáze. Tvrdíme, že na konci i-té fáze obsahuje výsledná posloupnost čísla utříděná podle i nejméně významných cifer. Zřejmě i-té nejméně významné cifry tvoří rostoucí posloupnost, neboť podle nich jsme právě v této fázi rozdělovali čísla do přihrádek, a pokud dvě čísla mají tuto cifru stejnou, jsou uložena v pořadí dle jejich $i-1$ nejméně významných cifer, neboť v každé přihrádce jsme zachovali pořadí čísel z konce minulé fáze. Na závěr poznamenejme, že místo čísel podle cifer lze do přihrádek rozdělovat též textové řetězce podle jejich znaků, atp.

Časová složitost této varianty RadixSortu, pokud třídíme celá čísla od 1 do K a v každém kroku je rozdělujeme do ℓ přihrádek, je $\mathcal{O}((N + \ell) \log_\ell K)$, tedy $\mathcal{O}(N)$, pokud K a ℓ jsou konstanty. My si předvedeme implementaci algoritmu pro $K = 255$ a $\ell = 2$ (čísla budeme roztřiďovat podle bitů v jejich binárním zápisu).

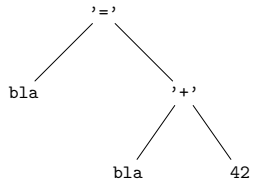
```
const K=255;
procedure RadixSort(var A: Pole);
var P0,P1: Pole;
    k1,k2: integer;
    i: integer;
    bit: integer;
begin
  bit:=1;
  while bit<=K do
    begin
      k1:=0; k2:=0;
      for i:=1 to N do
        if (A[i] and bit)=0 then
          begin
            k1:=k1+1; P0[k1]:=A[i];
          end
        else
          begin
            k2:=k2+1; P1[k2]:=A[i];
          end;
      for i:=1 to k1 do A[i]:=P0[i];
      for i:=1 to k2 do A[k1+i]:=P1[i];
      bit:=bit shl 1;
    end;
end;
```

Na závěr našeho povídání o třídících algoritmech si ukážeme, že třídit obecné údaje, se kterými neumíme provádět nic jiného, než je navzájem porovnávat, rychleji než $\mathcal{O}(N \log N)$ nejen nikdo neumí, ale také ani umět nemůže. Libovolný třídící algoritmus založený na porovnávání a prohazování prvků totiž musí na některé vstupy vynaložit řádově alespoň $N \log N$ kroků. (RadixSort na první pohled tento výsledek porušuje, na druhý však už ne, když si uvědomíme, o jak speciální druh tříděných dat se jedná.)

Třídící algoritmus v průběhu své činnosti nějak porovnává prvky a nějak je přehazuje. Provedeme myšlenkový experiment. Pozměníme algoritmus tak, že nejdříve bude pouze porovnávat, podle toho zjistí, jak jsou prvky v poli uspořádány, a když už si je jistý správným pořadím, prvky najednou popřehází. Předpokládejme pro jednoduchost, že všechny tříděné údaje jsou navzájem různé. Porovnávací

padně nějaké další informace (např. u identifikátorů řetězec, udávající jeho jméno, u čísel jejich hodnotu, apod.).

Druhým krokem je syntaktická analýza. Jejím úkolem je rozebrat strukturu programu. Výsledkem syntaktická analýzy výrazu „`bla = bla + 42`“ je to, že se jedná o přiřazení, na jehož levé straně je proměnná a na pravé straně je operátor sčítání aplikovaný na proměnnou a číslo. Povšimněte si, že v syntaktické analýze zpravidla záleží pouze na typech tokenů, nikoliv na další informacích u nich uložených – nezáleží na tom, jak se proměnná jmenuje, nebo jaká je hodnota čísla (i když samozřejmě tyto informace nesmíme ztratit). Výsledek se obvykle reprezentuje jako *syntaktický strom*. Vrcholy tohoto stromu odpovídají tokenům, synové vrcholu pak operandům příslušného výrazu nebo příkazu. Například vrchol obsahující plus bude mít jako syny sčítané výrazy, vrchol obsahující příkaz `if` bude mít jako syny podmínku, `then` větev a `else` větev. Syntaktický strom výrazu „`bla = bla + 42`“ vypadá takto:



Syntaktická analýza se samozřejmě provádí podle syntaxe daného programovacího jazyka. Syntaxe bývá nejčastěji zadána pomocí bezkontextové gramatiky. Co to přesně znamená, se můžete dočíst v loňském seriálu, my si pouze ukážeme příklad syntaxe jednoduchého aritmetického výrazu (obsahujícího sčítání, odčítání, násobení, dělení a závorky), ze které bychom měli získat hrubou představu:

```
výraz     →  výraz '+' výraz_nás
          |  výraz '-' výraz_nás
          |  výraz_nás

výraz_nás →  výraz_nás '*' operand
          |  výraz_nás '/' operand
          |  operand

operand   →  číslo
          |  proměnná
          |  '(' výraz ')'
```

Zde `výraz_nás` je výraz, jehož operátor má alespoň tak vysokou prioritu, jako násobení. Tedy podle této gramatiky například `operand` výrazu je buď číslo, nebo proměnná, nebo uzávorkovaný výraz.

Existují nástroje, kterým se zadá takováto bezkontextová gramatika a automaticky vytvoří kód, který provádí syntaktickou analýzu popsaného jazyka. Druhou variantou je napsat si syntaktickou analýzu „ručně“, což je o něco pracnější, ale snáze se popisuje ošetřování chyb a jiných speciálních případů.

Dalším krokem bývají jednoduché optimalizace (například zjednodušování výrazů), které se snadno provádí na syntaktických stromech. Pro většinu optimalizací je však reprezentace pomocí syntaktických stromů poměrně nevhodná. Například operandy výrazu mohou být libovolně složité a mohou mít různé vedlejší efekty (změny hodnot proměnných, volání libovolných funkcí, apod.), které činí práci s nimi dost nepohodlnou. Proto se program dále převádí do tzv. *mezikódu*, což bývá jazyk podstatně jednodušší, často podobný assembleru. To, jak přesně mezikód vypadá, se překladač od překladače dost liší, často se v jednom překladači používá i více mezijazyků, které se typicky čím dál tím víc blíží výslednému assembleru. Jednoduchý mezijazyk by mohl vypadat například takto:

Program je posloupnost příkazů. Příkazy jsou následující:

- `assign` *proměnná výraz* – přiřadí do proměnné hodnotu výrazu. Výraz musí být jednoduchý, tedy musí to být proměnná, číslo, nebo nějaká aritmetická operace, jejíž operandy jsou buď proměnné nebo čísla. Takže *výraz* může být třeba x, 42, $x + y$, ale už $x + y + 1$ je příliš složité.
- `label` *jméno* – označuje label, na který se dá skákat.
- `goto` *jméno* – skočí na label se jménem *jméno*.
- `if` *podmínka jméno*$_1$ *jméno*$_2$ – pokud je podmínka pravdivá, skočí na label se jménem *jméno*$_1$, jinak na label se jménem *jméno*$_2$. Podmínka musí být jednoduchá, tj. pouze porovnání dvou proměnných nebo čísel.

Například kousek programu v Pascalu

```
sum := 0;
for i := 1 to 10 do
  sum := sum + 2 * i;
```

se do mezikódu přeloží jako

```
assign sum 0
assign i 1
label loopbeg
assign tmp (2 * i)
assign sum (sum + tmp)
assign i (i + 1)
if (i <= 10) loopbeg loopend
label loopend
```

Nad takovýmto mezikódem se provádí většina optimalizací. Povšimněte si, že v mezikódu nezáleží na tom, jaký jazyk vlastně překládáme – stejný mezikód bychom dostali při překladu následujícího programu v C:

```
sum = 0;
for (i = 1; i <= 10; i++)
  sum += 2 * i;
```

Všechny optimalizace nad mezikódem tedy můžeme sdílet mezi překladači různých programovacích jazyků. Část překladače, která závisí na použitém programovacím jazyce a která končí typicky překladem do mezikódu, se nazývá *front-end*. Část, v níž se provádí optimalizace víceméně nezávislé jak na překládaném jazyce, tak na assembleru, do nějž program překládáme, se nazývá *middle-end*.

Poslední částí překladače je *back-end*. Tato část přepisuje program z mezikódu do assembleru, a je tedy závislá na architektuře, pro kterou je výsledný kód určen (jiný back-end se používá pro počítače založené na procesorech typu x86, které asi všichni znáte, jiný pro další, „exotičtější“ procesory). V back-endu se často provádí optimalizace specifické pro danou architekturu, například scheduling (přerovnávání instrukcí tak, aby se daly provádět paralelně), přiřazování proměnných do registrů a další.

Překladač se tedy skládá z jednoho či více front-endů pro různé programovací jazyky, middle-endu, a jednoho či více back-endů pro různé architektury. Samozřejmě ve skutečnosti rozdělení nebývá takto jasné. I v middle-endu musíme například znát časy provádění instrukcí, abychom mohli dobře optimalizovat, a tedy middle-end musí něco vědět

o cílových architekturách. Pokud je dobře navržen popis architektury, lze také často sdílet některé části back-endů, a některé optimalizace se tak přesouvají spíše do middle endu.

Úloha:

Abychom si pouze nepovídali, zkusíme si v této sérii prakticky napsat jednoduchý front-end k překladači. Abychom se vyhnuli komplikacím, měl by umět pouze překládat výrazy popsané gramatikou uvedenou v textu seriálu, do popsaného mezikódu. Výsledek výrazu by měl být přiřazen do proměnné `result`. Například výraz

```
x * (x + y) - z / bla / neco * 5
```

by měl být přeložen jako

```
assign tmp1 (x + y)
assign tmp2 (x * tmp1)
assign tmp3 (z / bla)
assign tmp4 (tmp3 / neco)
assign tmp5 (tmp4 * 5)
assign result (tmp2 - tmp5)
```

Recepty z programátorské kuchařky

I letos Vám kromě úloh budeme servírovat také recepty z programátorské kuchařky. Některé si vypůjčíme z dřívějších ročníků KSP, ale i k těm se budeme snažit připsat něco nového, aby si i zkušenější řešitelé přišli na své. V letošní první kuchařce si povíme o třídících algoritmech. Co to znamená? Pojem *třídění* je možná maličko nepřesný, nehodláme data (čísla, záznamy, řetězce a jiné) rozdělovat do nějakých tříd, ale přerovnat je do správného pořadí, protože se seřazenými údaji se mnohem lépe pracuje, například pokud v nich pak potřebujeme vyhledávat. Takové uspořádávání dat je denním chlebem každého programátora, a tak není divu, že třídící algoritmy jsou jedny z nejstudovanějších. My však nebudeme do nějakých velkých detailů a specialit příliš zabíhat. Zkrátka a dobře – budeme chtít třídit údaje rychle, úsporně a radostně.

Obvykle třídíme exemplář datové struktury typu pascalského záznamu. V takové datové struktuře bývá obsažena jedna význačná položka, *klíč*, podle které se záznamy řadí. Malinko si náš život zjednodušíme a budeme předpokládat, že třídíme záznamy obsahující pouze klíč, který je navíc celočíselný – budeme tedy třídit pole celých čísel. Pomocí počtu tříděných čísel N pak budeme vyjadřovat časovou (a paměťovou) složitost jednotlivých algoritmů, které si předvedeme.

Metody třídění můžeme rozdělit do dvou hlavních skupin, a to na *vnitřní třídění*, kdy si můžeme dovolit všechna data načíst do (rychlé) paměti počítače, a na *vnější třídění*, kdy již třídění musíme realizovat opakovaným čtením a vytvářením diskových souborů. V tomto dílu se omezíme pouze na algoritmy vnitřního třídění a tříděné pole si nadeklarujeme takto:

```
const N = 100;
type Pole = array[1..N] of integer;
```

Nejjednodušší třídící algoritmy patří do skupiny *přímých metod*. Všechny mají několik společných rysů: Jsou krátké, jednoduché a třídí přímo v poli (nepotřebujeme pomocné pole). Tyto algoritmy mají většinou časovou složitost $\mathcal{O}(N^2)$. Z toho vyplývá, že jsou použitelné tehdy, když tříděných dat není příliš mnoho. Na druhou stranu pokud je dat opravdu málo, je zbytečně složité používat některý z komplikovanějších algoritmů, které si předvedeme později.

Stručně si přiblížíme tři nejznámější algoritmy pro třídění přímými metodami. *Třídění přímým výběrem (SelectSort)* je založeno na opakovaném vybírání nejmenšího čísla z dosud nesetříděných čísel. Nalezené číslo prohodíme s prvkem na začátku pole a postup opakujeme, tentokrát s nejmenším číslem na indexech $2, \ldots, N$, které prohodíme s druhým prvkem v poli. Poté postup opakujeme s prvky s indexy $3, \ldots, N$, atd. Je snadné si uvědomit, že když takto postupně vybíráme minimum z menších a menších intervalů, setřídíme celé pole (v i-tém kroku nalezneme i-tý nejmenší prvek a zařadíme ho v poli na pozici s indexem i).

```
procedure SelectSort(var A: Pole);
var i,j,k,x: integer;
begin
  for i:=1 to N-1 do
    begin
      k:=i;
      for j:=i+1 to N do
        if A[j]<A[k] then k:=j;
      x:=A[k]; A[k]:=A[i]; A[i]:=x;
    end;
end;
```

Pro úplnost si ještě řekněme pár slov o časové složitosti právě popsaného algoritmu. V i-tém kroku musíme nalézt minimum z $N-i+1$ čísel, na což spotřebujeme čas $\mathcal{O}(N-i+1)$. Ve všech krocích dohromady tedy spotřebujeme čas $\mathcal{O}(N+(N-1)+\ldots+3+2+1) = \mathcal{O}(N^2)$.

Třídění přímým vkládáním (InsertSort) funguje na podobném principu jako třídění přímým výběrem. Na začátku pole vytváříme správně utříděnou posloupnost, kterou postupně rozšiřujeme. Na začátku i-tého kroku má tato utříděná posloupnost délku $i-1$. V i-tém kroku určíme pozici i-tého čísla v dosud utříděné posloupnosti a zařadíme ho do utříděné posloupnosti (zbytek utříděné posloupnosti se posune o jednu pozici doprava). Není těžké si rozmyslet, že každý krok lze provést v čase $\mathcal{O}(N)$. Protože počet kroků algoritmu je N, celková časová složitost právě popsaného algoritmu je opět $\mathcal{O}(N^2)$.

```
procedure InsertSort(var A: Pole);
var i,j,x: integer;
begin
  for i:=2 to N do
    begin
      x:=A[i];
      j:=i-1;
      while (j>0) and (x<A[j]) do
        begin
          A[j+1]:=A[j];
          j:=j-1;
        end;
      A[j+1]:=x;
    end;
end;
```

(Upozornění: v našich příkladech předpokládáme, že máme v překladači zapnuto tzv. zkrácené vyhodnocování logických výrazu, třeba v předchozím while-cyklu se při `j=0` hodnoty `x` a `A[0]` již neporovnávají.)

Bublinkové třídění (BubbleSort) pracuje jinak než dva dříve popsané algoritmy. Algoritmu se říká „bublinkový“, protože podobně jako bublinky v limonádě „stoupají“ vysoká čísla v poli vzhůru. Postupně se porovnávají dvojice sousedních prvků, řekněme zleva doprava, a pokud v porovnávané dvojici následuje menší číslo po větším, tak se tato dvě čísla prohodí. Celý postup opakujeme, dokud probíhají nějaké výměny. Protože algoritmus skončí, když nedojde k žádné výměně, je pole na konci algoritmu setříděné.

```
procedure BubbleSort(var A: Pole);
var i,x: integer;
    zmena: boolean;
begin
  repeat
    zmena:=false;
    for i:=1 to N-1 do
      if A[i] > A[i+1] then
        begin
          x:=A[i]; A[i]:=A[i+1]; A[i+1]:=x;
          zmena:=true;
        end;
  until not zmena;
end;
```

Správnost algoritmu nahlédneme tak, že si uvědomíme, že po i průchodech while-cyklem bude posledních i prvků obsahovat největších i prvků setříděných od nejmenšího po největší (rozmyslete si, proč tomu tak je). Popsaný algoritmus se tedy zastaví po nejvýše N průchodech a jeho celková časová složitost v nejhorším případě je $\mathcal{O}(N^2)$, neboť na každý průchod spotřebuje čas $\mathcal{O}(N)$. Výhodou tohoto algoritmu oproti předchozím dvěma algoritmům je, že pokud je pole na začátku setříděné, tak algoritmus spotřebuje jen lineární čas, $\mathcal{O}(N)$.

Lepší třídící algoritmy pracují v čase $\mathcal{O}(N \log N)$. Jedním z nich je *Třídění sléváním (MergeSort)*, založené na principu slévání (spojování) již setříděných posloupností dohromady. Představme si, že již máme dvě setříděné posloupnosti a chceme je spojit dohromady. Jednoduše stačí porovnávat nejmenší prvek z každé posloupnosti, který jsme dosud nedali do nově vytvářené posloupnosti, a menší z těchto prvků do nové posloupnosti přidat. Je zřejmé, že ke slití dvou posloupností potřebujeme čas úměrný součtu jejich délek.

My si zde popíšeme a předvedeme modifikaci algoritmu MergeSort, která používá pomocné pole. Algoritmus lze implementovat při zachování časové složitosti i bez pomocného pole, ale je to o dost pracnější. Existuje též modifikace algoritmu, která má počet fází (viz dále) v nejhorším případě $\mathcal{O}(\log N)$, ale pokud je již pole na začátku setříděné, proběhne pouze jediná a v takovém případě má algoritmus časovou složitost $\mathcal{O}(N)$. My si však zatajíme i tuto variantu.

Algoritmus pracuje v několika *fázích*. Na začátku první fáze tvoří každý prvek jednoprvkovou setříděnou posloupnost a obecně na začátku i-té fázi budou mít setříděné posloupnosti délky 2^{i-1}. V i-té fázi tedy vždy ze dvou sousedních 2^{i-1}-prvkových posloupností vytvoříme jedinou délky 2^i. Pokud N není násobkem 2^i, bude délka poslední posloupnosti zbytek po dělení N číslem 2^i. Zastavíme se, pokud $2^i \geq N$, tj. po $\lceil \log_2 N \rceil$ fázích. Protože v i-té fázi slijeme $\lceil N/2^i \rceil$ dvojic nejvýše 2^{i-1}-prvkových posloupností, je časová složitost jedné fáze $\mathcal{O}(N)$. Celková časová složitost popsaného algoritmu je pak $\mathcal{O}(N \log N)$.

```
procedure MergeSort(var A: Pole);
var P: Pole;        { pomocné pole }
    delka:integer;  { délka setříděných posl. }
    i: integer;     { index do vytvářené posl. }
    i1,i2: integer; { index do slévaných posl. }
    k1,k2: integer; { konce slévaných posl. }
begin
  delka:=1;
  while delka<N do
    begin
      i1:=1; i2:=delka+1; i:=1;
      k1:=delka; k2:=2*delka;
      while i<=N do
        begin
          if k2>N then k2:=N;
          while (i1<=k1) or (i2<=k2) do
            if (i1>k1) or
               ((i2<=k2) and (A[i1]<=A[i2]))
            then
              begin
                P[i]:=A[i1]; i:=i+1; i1:=i1+1;
              end
            else
              begin
                P[i]:=A[i2]; i:=i+1; i2:=i2+1;
              end;
          i1:=k2+1; i2:=k2+delka;
          k1:=k2+delka;
          k2:=k2+2*delka;
        end;
      A:=P;
      delka:=2*delka;
    end;
end;
```

V čase $\mathcal{O}(N \log N)$ pracuje také algoritmus jménem *QuickSort*. Tento algoritmus je založen na metodě Rozděl a panuj. Nejprve si zvolíme nějaké číslo, kterému budeme říkat *pivot*. Více si o jeho volbě povíme později. Poté pole přeuspořádáme a rozdělíme je na dvě části tak, že žádný prvek v první části nebude větší než pivot a žádný prvek v druhé části naopak menší. Prvky v obou částech pak setřídíme rekurzivním zavoláním téhož algoritmu. Musíme ale dát pozor, aby v každém kroku obě části byly neprázdné (a rekurze tedy byla konečná). Je zřejmé, že po skončení algoritmu bude pole setříděné.

Malá zrada spočívá ve volbě pivota. Pro naše účely by se hodilo, aby po přeházení prvků levá i pravá část pole byly přibližně stejně velké. Nejlepší volbou pivota by tedy byl *medián* tříděného úseku, tj. prvek takový, jenž by byl v setříděném poli přesně uprostřed. Přeuspořádání jistě zvládneme v lineárním čase a pokud by pivoty na všech úrovních byly mediány, pak by počet úrovní rekurze byl $\mathcal{O}(\log N)$ a celková časová složitost $\mathcal{O}(N \log N)$ (na každé úrovni rekurze je součet délek tříděných posloupnosti nejvýše N). Ačkoli existuje algoritmus, který medián pole nalezne v čase $\mathcal{O}(N)$, v QuickSortu se obvykle nepoužívá, jelikož konstanta u členu N je příliš velká v porovnání s pravděpodobností, že náhodná volba pivota algoritmus příliš zpomalí. Většinou se pivot volí náhodně z dosud nesetříděného úseku – zkrátka se sáhne někam do pole a nalezený prvek se prohlásí za pivot. Dá se ukázat, že takovýto algoritmus s velmi vysokou pravděpodobností poběží v čase $\mathcal{O}(N \log N)$. Důkaz tohoto tvrzení je trošičku trikový a lze jej nalézt např. v knize Kapitoly z diskrétní matematiky od pánů Matouška a Nešetřila.