Korespondenční Seminář

z Programování

Dokud existují počítače, bude existovat i **KSP**čko.
Že jsi o něm ještě neslyšel(a)? V tom případě
si zkus odpovědět na následující otázky:

- Zajímáš se o počítače?
- Rád(a) soutěžíš?
- Chceš se dozvědět něco nového?
- Chceš poznat nové lidi?
- Chceš užitečně vyplnit volný čas?
- Hledáš výzvu pro svoji hlavu?

Odpověděl(a) sis alespoň jednou „ano“? Pak hledáme
právě Tebe. Do KSP se může zapojit každý.
Máš-li chuť, otoč list …

Na této stránce najdeš odpovědi na základní otázky o KSP, vesmíru a vůbec.

Co všechno znamená KSP?

Korejská strana práce, Katedra správního práva, Klub severských psů, nebo třeba Korespondenční seminář z programování! Korejští kynologové mají smůlu, zůstaneme u posledního.

Korespondenční seminář z programování?

Celostátní a celoroční soutěž v programování pro studenty středních škol a vyšších ročníků základních škol.

Jak tato soutěž probíhá?

Jeden ročník je rozdělen na 5 sérií, přičemž v každé obdrží účastníci zadání přibližně 8 úloh (buď poštou nebo po Internetu). Na vyřešení série bývá několik týdnů času, takže můžeš řešit v klidu v teple domácího krbu, v MHD nebo o nudné hodině ve škole.

Opravená řešení ti později pošleme poštou spolu se vzorovými řešeními nebo si je můžeš stáhnout z našich stránek.

Jaké jsou úlohy?

Úlohy jsou převážně čistě algoritmické. Rychlejší a lépe popsané algoritmy mají přednost před programy hýřícími barvami.

Jak se počítají výsledky?

Úlohy jsou za určitý počet bodů dle obtížnosti, do výsledků se každému započítá 5 nejlépe vyřešených úloh ze série. Začátečníky bodujeme mírněji, za drobné chyby ztrácejí méně bodů než zkušení řešitelé. Celkové hodnocení je tvořeno součtem bodů ze všech sérií.

Vůbec nevím, co napsat do řešení. Co s tím?

Nalistuj si konec letáku, kde jsme pro Tebe přichystali stručný návod.

Jak rozeznám lehké a těžké úlohy?

Jednak se můžeš kouknout na body, jež by měly přibližně odpovídat obtížnosti (samozřejmě záleží na znalostech a jak komu úloha sedne), druhak najdeš u některých úloh následující značky:

Takto označenou úlohu považujeme za řešitelnou i pro začátečníky, zkušení řešitelé ji jistě dají levou zadní. Pro její vyřešení by neměly být potřeba žádné speciální znalosti.

Aby si i pokročilí přišli na své, zařazujeme někdy do zadání těžkou úlohu, která se může stát leckomu noční můrou. Na její pokoření jsou často potřeba hlubší znalosti algoritmů a datových struktur, odměnou je však vyšší bodový zisk.

Univerzální odpověď na všechny ostatní otázky: 42.

Této úloze říkáme *praktická*, jelikož není potřeba popsat algoritmus, jen ho naprogramovat a odevzdat přes Internet. Bližší informace naleznete přímo v jejím zadání.

V každém ročníku KSP rozebíráme na pokračování nějaké zajímavé informatické téma do hloubky. Poslední úloha série je pokračováním takového *seriálu* – obsahuje kromě samotného zadání ještě text, ve kterém se můžeš dozvědět o tématu něco nového. Jelikož díly seriálu na sebe navazují, vyplatí se mít nastudované i předchozí série.

Protože chápeme, že k „uvaření“ řešení jsou často potřeba znalosti základních algoritmů a datových struktur, obvykle též přikládáme do každé série tzv. *kuchařku*, ze které se můžeš takové věci naučit. Často je také v zadání úloha, již lze řešit algoritmem z kuchařky. A pozor – další kuchařky najdete na našich webových stránkách.

Dostanu za řešení nějakou odměnu?

Nejlepších 30 řešitelů zveme na začátku dalšího školního roku (obvykle v září) na týdenní **soustředění**, na kterém se v rychlém tempu střídají hry a odborný program. Vyspíte se až doma!

Dále každý, kdo překoná zhruba 50% hranici bodů, se stane úspěšným řešitelem a jako takovému mu budou **odpuštěny přijímačky** na Matfyz!

Jsi-li začínající řešitel, můžeš také jet na jarní soustředění (v dubnu či květnu), kde učíme základy programování a algoritmů.

A co když se stanu nejlepším z nejlepších?

Tři nejlepší řešitelé 24. ročníku se stanou na rok **králi KSP**. Navíc obdrží libovolnou knihu dle svého ctěného výběru (v případě 2. a 3. nejlepšího jen českou) a dalších výsad budou požívat i na podzimním soustředění (extra moučníky!).

Co budu dělat o prázdninách?

První série se odevzdává až koncem října. Během prázdnin se tak můžeš kochat přírodou, surfovat, lézt po horách anebo řešit nultou sérii! V této originální sérii jsme přichystali několik netradičních úložek, jejichž řešení můžete odevzdávat na našich stránkách až do 20. srpna.

Kde se dozvím více a jak se přihlásím?

Další informace a přihlášku nalezneš na

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

Dotazy (ale ne řešení úloh) můžeš posílat na

ksp@mff.cuni.cz

Hodně štěstí!

Milí řešitelé a řešitelky!

Držíte v ruce první leták 24. ročníku KSP. Každá série letos obsahuje 8 úloh a z nich se 5 nejlépe vyřešených započítává do celkového bodového hodnocení. Proti loňskému ročníku tedy jedna úloha v každé sérii přibyla.

V každé sérii budou nově dvě lehké úlohy pro začátečníky za menší počet bodů.

Nově je také možno být přijat na MFF UK za úspěšné řešení KSP. Úspěšným řešitelem se stává ten, kdo získá za celý ročník alespoň 50 % bodů. Za letošní rok půjde získat maximálně 300 bodů, takže hranice pro úspěšné řešitele je 150.

Upozorňujeme letošní maturanty, že termín odevzdání páté série bude pravděpodobně příliš pozdě na to, aby pátou sérii dohánělí chybějící body. Diplom úspěšného řešitele ale můžeme v případě potřeby zaslat i dříve, budete-li mít dost bodů.

Termín odevzdání první série je stanoven na pondělí 24. října v 8:00 SELČ, což znamená, že papírové řešení byste měli podat na poštu do středy 19. října.

Řešení přijímáme elektronicky na stránce *https://ksp.mff.cuni.cz/submit/*. Chcete-li s námi komunikovat bezpečně, můžete si ověřit náš HTTPS certifikát – zde je jeho SHA1 hash: `7F:53:E7:00:60:F2:24:93:8F:52:51:EC:1E:A8:34:54:86:69:32:7D`.

Také nám řešení můžete poslat klasickou poštou na adresu

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

Před tím ale vyplňte přihlášku (a to i tehdy, když jste se KSPčka účastnili loni) na *http://ksp.mff.cuni.cz/,* kde najdete i další informace o tom, jak KSP funguje. Na webu máme také fórum, kde se můžete na cokoli zeptat. Nebo nám můžete napsat na e-mail *ksp@mff.cuni.cz.*

První série čtyřiadvacátého ročníku KSP

```
FD 60 BK 120 FD 60 RT 90 FD 50
LT 90 FD 60 BK 120 FD 60
```

```
10  PRINT "e"
```

```
++++++++++[>++++++++++<-]>--.
```

```
Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook.
Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook.
Ook. Ook. Ook. Ook. Ook! Ook? Ook! Ook!
Ook. Ook? Ook. Ook. Ook. Ook. Ook. Ook.
Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook.
Ook. Ook. Ook. Ook. Ook. Ook. Ook? Ook.
Ook? Ook! Ook. Ook? Ook. Ook. Ook. Ook.
Ook. Ook. Ook. Ook. Ook. Ook. Ook. Ook.
Ook. Ook. Ook. Ook. Ook! Ook. Ook? Ook.
```

```
print split /[^g-q]/, lc sub {};
```

```
#include <stdio.h>
int main() {
  printf("%c", (107^25)-70&99);
  return 0;
}
```

```
h(X,[]) :- put(X), !.
h(X,[A|B]) :- Y is X + A, h(Y,B).
?- h(42, [5, 6, 7, 8, 7]).
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SERIE1.
PROCEDURE DIVISION.
    DISPLAY 'S'.
    STOP RUN.
```

```
(defun gen (i j)
   (if (<= i j)(cons i (gen (1+ i) j)) nil))
(defun split (s)
   (mapcar (lambda (i) (substring s (1- i) i))
           (gen 1 (length s))))
(defun GAPS (x)
   (princ (caddr (split (symbol-name x)))))
(defun Q (x)
   (eval (list x (list 'quote x))))
(Q 'GAPS)
```

```
class Program {
  static void Main() {
    System.Console.Write((char)0x21);
  }
}
```

Toto je jen malý náhled do zvěřince programovacích jazyků. Dokážete rozpoznat jednotlivé jazyky? Umíte zjistit, co programy v nich udělají a co z toho vznikne dohromady? Pokud ne, může vám pomoci malý výlet do historie.

První předzvěstí programovacího jazyka byl v roce 1801 vynález tkalcovského stavu ovládaného děrnými štítky (kartami z tvrdšího papíru, jež obsahovaly data zakódovaná do děr). Jednalo se však spíše o kód než jazyk.

Dalším významným počinem se stalo v půli 19. století napsání prvního programu, a to na počítání Bernoulliho čísel. Kupodivu ho nenapsal muž, ale Ada Lovelace v korespondenci s Charlesem Babbagem pro jeho analytický stroj.

Až doba elektromechanických počítačů z konce 30. let a ze 40. let 20. století odstartovala rychlý vývoj programovacích jazyků. Hybatelem pokroku byla nepřekvapivě druhá světová válka, zejména touha prolomit šifry nepřátelské strany.

Tehdy se programovalo pomocí přepínačů či děrných štítků (v podstatě sekvencí 0 a 1) a pro každý počítač jinak (by-

lo jich tehdy naštěstí jen několik na světě). Problémy tohoto způsobu se však objevily celkem rychle, především v množství chyb, jež raní programátoři udělali.

To vedlo k vývoji vyšších jazyků. První návrh, pojmenovaný Plankalkül, vymyslel Konrad Zuse (autor elektromechanických počítačů Z1, Z2, Z3 a Z4), nikdy ho však neimplementoval.

Po strojových kódech přišly „assemblery", které nahrazovaly binární zápisy anglickými slovy jako „add" a „xor".

24-1-1 Podvádíme s XORem — 8 bodů

Představte si, že jste s kolegou z práce dostali obrovskou hromadu hardwarových součástek, přičemž každá má nějakou cenu danou přirozeným číslem. Chcete je rozdělit na dvě části, aby byl v obou stejný součet cen.

Kolega však není váš kamarád, a tak ho zkusíte podvést. Vy rozdělíte součástky na dvě hromádky a on si součty překontroluje programem v assembleru, jenže nebude tušit, že dělá operaci XOR místo sčítání (což jste mu nenápadně prohodili).

Operace XOR (exkluzivní OR, neboli vylučovací nebo) pracuje se dvěma čísly po bitech tak, že ve výsledném čísle je na i-tém místě jednička, když byla jednička na i-tém místo právě v jednom ze vstupních čísel (tzn. ne v obou).

Příklad (čísla jsou v binárním zápisu, v závorce desítkově):

```
    11001001 (201)
XOR 01100101 (101)
    --------------
 =  10101100 (172)
```

Máte tedy seznam přirozených čísel, který chcete rozdělit tak, aby XOR všech prvků byl v obou částech stejný, ale rozdíl součtů co největší (menší připadne přirozeně kolegovi).

K této úloze není potřeba vymyslet algoritmus nebo napsat program, jde spíše o nalezení způsobu rozdělování čísel.

Prvním z moderních vyšších programovacích jazyků, jež se stále používají, je FORTRAN (FORmula TRANslator) z roku 1955, původně určený pro vědeckotechnické výpočty. Stal se předchůdcem dnešních imperativních jazyků, v nichž se program zapisuje jako posloupnost příkazů s přesně daným pořadím vyhodnocení.

Brzy ho následoval zcela odlišný LISP (LISt Processor), první funkcionální jazyk. Zlí jazykové mu přezdívají Lots of Irritating Superfluous Parentheses (spousta otravných nadbytečných závorek), protože téměř každý příkaz je ohraničen kulatými závorkami.

Funkcionální se podobným jazykům říká, protože zachází s výpočtem jako s vyhodnocováním matematické funkce. Představují jeden z přístupů deklarativního programování, v němž se na rozdíl od imperativního jen určuje, co se má udělat, kdežto imperativní jazyk popisuje i postup.

Druhým rozšířeným deklarativním přístupem je logické programování, které vzniklo začátkem 70. let. Nejznámějším zástupcem je Prolog, v němž jsou jednotlivé části programu v podstatě logické formule.

Koncem 50. let do rodiny imperativních jazyků přibyl ALGOL (ALGOrithmic Language) uzpůsobený pro přehlednější zápis algoritmů. Jako první přišel s bloky příkazů, které byly vyznačeny slovy begin a end.

Že jste už begin a end někde viděli? Ano, z ALGOLu se koncem 60. let vyvinul Pascal, nejdříve určený pro výuku programování, ovšem dodnes rozšířený i v komerční sféře.

Od 60. let se s jazyky doslova roztrhl pytel. Jmenujme jen ty významné, rozšířené nebo alespoň něčím zajímavé.

V roce 1964 byl vytvořen BASIC (Beginner's All-purpose Symbolic Instruction Code), stejně jako FORTRAN a ALGOL imperativní. Při jeho vytváření byl kladen důraz na snadné používání a podobnost angličtině. Jeho dialekty a pokračovatelé jako Visual BASIC jsou dodnes hojně používané.

24-1-2 Rozházené řádky v BASICu — 7 bodů

Programátorský šotek měl veselou náladu, a tak přeházel řádky ve vašem už značně dlouhém programu v BASICu. Naštěstí je na řádcích na začátku napsáno jejich číslo (na rozdíl od starých verzí BASICu, kde se typicky číslovalo po desítkách, čísla v tomto programu začínají od 1 a přibývají po jedné).

Soubor lze spravit pouze prohazováním dvojic řádků, ale vy se jako správní programátoři nechcete moc nadřít a rádi byste provedli co nejméně prohození, abyste dostali původní program.

Vymyslete algoritmus, který dostane na vstupu posloupnost N čísel od 1 do N, v níž se žádné neopakuje (tedy permutaci), a má určit, na kolik nejméně prohození 2 řádků lze dostat seřazenou posloupnost od 1 do N.

Příklad: pro permutaci `3, 10, 8, 4, 6, 5, 9, 1, 2, 7` je správnou odpovědí 6.

Z konce 50. a začátku 60. let pochází také zvláštní programovací jazyk APL založený na matematické notaci. Programy v něm jsou typicky jednořádkové a vyhodnocují se striktně zprava doleva (tedy v APL neexistuje nic jako priorita operátorů).

Ptáte se, jak se program dokáže vejít na jednu řádku? Docela pěkně, když jsou operátory jednoznakové, jen je pro ně třeba použít tolik znaků, že většina není na běžných klávesnicích. Pár příkladů: ÷ (dělení), ⍴ (zjištění rozměrů pole), více jich můžete najít v předloňském seriálu.[1]

Na počátku 70. let vznikl v Bellových laboratořích současně se systémem Unix jazyk C vyvinutý z B (B už však nepředcházelo žádné A, zato jazyk D z přelomu tisíciletí navazuje na C).

C bylo navrženo více nízkoúrovňově, a tudíž je vhodné na systémové a výkonově náročné aplikace. Po svých předchůdcích zdědilo označování bloků znaky `{` a `}`.

24-1-3 Turnaj jazyků — 12 bodů

Když už jsme si tu několik programovacích jazyků představili, můžeme mezi nimi uspořádat velký turnaj, jehož se zúčastní i jazyk budoucnosti, BestLang. Ten je přirozeně zcela nejlepší a vždy vyhraje.

V každém kole turnaje je zadána úloha, přední odborníci na jednotlivé jazyky v každém napíší řešení a do dalšího kola postupují ty jazyky, jejichž programy doběhly do dvojnásobku času nejlepšího řešení.

Jak bylo řečeno, BestLang vyhraje. Ale chce vyhrát s co nejvíce body a ty se počítají za každé kolo vzorcem

$$\frac{\textit{počet vyřazených} \cdot 100\,000}{\textit{počet na začátku kola}}.$$

Dělení ve vzorci je celočíselné (počítá se dolní celá část) a *počet na začátku kola* obsahuje i BestLang. Celkový počet bodů určuje součet bodů ze všech kol.

[1] `http://ksp.mff.cuni.cz/viz/22-4-7`

BestLang je navíc tak dobrý, že si může v každém kole vybrat, kolik jazyků vyřadí (všechna řešení v ostatních jazycích doběhnou ve skoro stejném čase a BestLang dokáže zjistit, v jakém). V kole nemusí vyřadit žádný jazyk.

Máte daný počet jazyků (N) a počet kol (K), přičemž $N \leq 1\,000$ a $K \leq 1\,000$, a úkolem je najít takovou posloupnost počtu vyřazených v jednotlivých kolech, aby BestLang získal co nejvíce bodů.

V jakémkoliv kole může klidně vyřadit všechny, ale po posledním kole musí zůstat v turnaji sám.

Příklady: pro $N = 500$ a $K = 3$ je řešením `437, 55, 7`, pro $N = 15$ a $K = 8$ je to `3, 3, 2, 2, 1, 1, 1, 1`.

Pojďme si povědět ještě něco o programovacích jazycích obecně – hlavně o tom, jaké jsou jejich druhy. Mezi nejvýraznější patří rozdělení na nízkoúrovňové (více se blížící strojovému kódu, tedy jazyku počítače) a vyšší (bližší člověku).

Zástupcem první skupiny je např. assembler. Dalších nízkoúrovňových není tolik, pokud nebudeme hledět na odlišnosti dané hardwarem, a programátoři se s nimi setkávají málo, většina vývoje i dělení se proto týká jen vyšších jazyků.

Při procházení historických jazyků jsme už nakousli dělení na jazyky imperativní (program je posloupnost příkazů) a deklarativní (zapisuje se, co se má spočítat, a nezáleží tolik na tom, jak). Známými deklarativními jazyky jsou LISP, Haskell (oba funkcionální) a Prolog (logické programování).

Všechny imperativní jazyky jsou dnes procedurální, ačkoliv zprvu neobsahovaly podprogramy, neboli procedury či funkce. Často se proto musel používat příkaz `goto` (skok na jiné místo v programu), což vedlo k nepřehlednosti. Dnes už se `goto` téměř nepoužívá, i když v programovacích jazycích často bývá.

O objektový přístup obohatil programování jazyk Simula určený pro diskrétní simulace. Obsahuje objekty, třídy, dědičnost, a dokonce i garbage collection *(automatickou správu paměti).*

Z hlediska rychlosti provádění kódu a pohodlnosti při psaní jsou dvěma protipóly jazyky kompilované (kompilátorem převáděné do strojového kódu) a interpretované (program, jemuž se říká interpretr, čte kód a rovnou ho provádí, což bývá pomalejší).

Pokud vám na předchozím odstavci něco nesedí, pak je to dobře. Jazyk je totiž jen forma zápisu, a tak existují interpretry jazyka C, typického zástupce kompilovaných, a naopak kompilátory pro skriptovací jazyk Python.

24-1-4 Složitá složitost — 7 bodů

Ačkoliv si v této sérii vyprávíme o programovacích jazycích, při řešení úloh KSPčka nám o ně obvykle moc nejde – víc než použitý jazyk, ať už kompilovaný nebo interpretovaný, nás zajímá nás tzv. asymptotická časová složitost. Ta je zcela nezávislá na jazyce a umožňuje rozlišit, jak je který algoritmus rychlý, aniž bychom ho museli spouštět na skutečném počítači.

Více se o složitosti dozvíte z kuchařky na konci letáku. Její přečtení doporučujeme před řešením této úlohy všem, kdo ještě nikdy složitost neurčovali nebo jim stále přijde poněkud složitá. Ostatním může kuchařka sloužit pro osvěžení znalostí před novým ročníkem.

V této úloze jsme si pro vás připravili pseudokód (zjednodušený kód) algoritmu. Jeho úkolem je setřídit zadané pole čísel, čili jeho prvky přerovnat do vzestupného pořadí. Vaším úkolem pak je určit časovou a paměťovou složitost tohoto algoritmu a zdůvodnit, proč tomu tak je. Zajímá nás složitost v nejhorším případě.

Ještě poznámka pro pořádek: pole indexujeme od 1 a parametr `n` říká, kolik v něm je uloženo prvků.

```
Funkce Setřiď(pole, n):
   odm = odmocnina z n zaokrouhlená dolů
   i = 1
   Dokud i <= n:
      změna = true
      Dokud změna je true:
         změna = false
         Pro j od i do min(i+odm-2, n-1):
            Jestliže pole[j] > pole[j+1]:
               Prohoď pole[j] a pole[j+1]
               změna = true
      i = i + odm
   vysl = vytvoř pole délky n
   zač = vytvoř pole délky odm+1
   Do všech prvků pole zač vlož 0
   Pro i od 1 do n:
      vysl[i] = nekonečno
      minIndex = 1
      j = 1
      k = 1
      Dokud j <= n:
         Jestliže zač[k] < odm a j+zač[k] <= n:
            a = pole[j + zač[k]]
            Jestliže a < vysl[i]:
               vysl[i] = a
               minIndex = k
         j = j + odm
         k = k + 1
      zač[minIndex] = zač[minIndex] + 1
   Vrať vysl
```

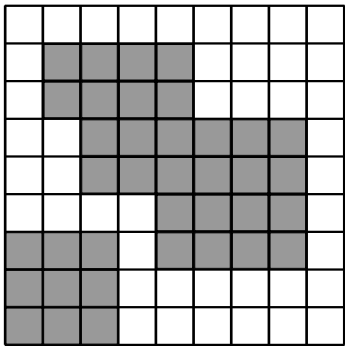
Zvláštní kategorii tvoří výukové jazyky, snažící se být co nejjednoduššími a zároveň poutavými pro děti. Někdy jsou v nich textové příkazy nahrazeny ikonkami, z nichž se vytváří program metodou táhni a pusť.

V Logu, starém již přes 40 let, se ovládá želva, která chodí po ploše, a když spustí ocásek, kreslí za sebou stopu. Tomuto způsobu kreslení se dodnes říká želví grafika.

24-1-5 Razítková grafika — 12 bodů

Představme si vytváření grafiky trochu podobné želví, ale s jedinou operací – otisk čtvercového razítka. Kreslit budeme na klasickou bitmapu (čtvercovou síť pixelů) jedinou barvou, například černou na bílé pozadí.

Dostali jste černobílý obrázek o rozměrech $N \times M$ pixelů a vaším úkolem je určit, pomocí jakého největšího razítka mohl být vytvořen. Žádný pixel nesmí být orazítkován dvakrát.



Na obrázku je příklad vstupní bitmapy. Největší razítko, kterým se dá vytvořit, má velikost 1 pixel, razítkem se 2 pixely by nešel nakreslit čtverec 3×3 vlevo dole.

Tato úloha je praktická a řeší se ve vyhodnocovacím systému CodEx.[2] Přesný formát vstupu a výstupu, povolené jazyky a další technické informace jsou uvedeny v CodExu přímo u úlohy.

Dalším jazykem pro děti je v ČR vytvořený Karel (pojmenovaný po Karlu Čapkovi), v němž se na čtvercové síti ovládá robot. Domácího původu je i Baltík obsahující postavičku čaroděje, který chodí po ploše a čaruje na políčka obrázky.

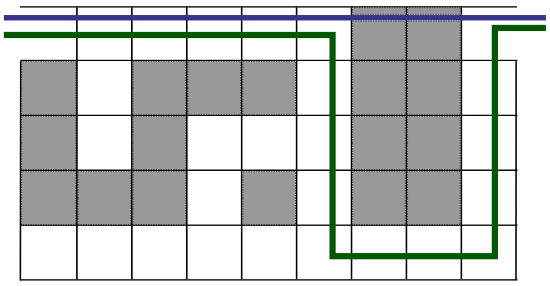
24-1-6 V bludišti s krumpáčem — 9 bodů

V Baltíkovi i v Karlovi je typickou úlohou pro začátečníky procházení bludiště. Postavička chodí po čtvercové (popř. obdélníkové) síti, musí se vyhýbat zdem, ale občas se může rozhodnout nějakou zbourat.

Máte mapu velkého bludiště na čtvercové síti s rozměry $N \times M$. Políčko je buďto prázdné, nebo je na něm zeď. Úkolem je najít pro postavičku nejrychlejší cestu od vchodu do bludiště k východu (je tam jen jeden) s tím, že zbourání zdi stojí stejně času jako ujít $K-1$ políček (takže políčko se zdí postavička projde celkově za stejný čas jako K prázdných).

Pro jednoduchost stačí vypsat dobu na projití nalezené trasy. Můžete také předpokládat, že K je nejvýše 10.

Příklad bludiště vidíte níže na obrázku. Pro K menší než 5 se vyplatí probourat dvě zdi, pro K větší než 5 je lepší zdi obejít a pro $K = 5$ jsou stejně dobré obě cesty.



Léta vývoje programovacích jazyků přinesla i mnohé plody, jež jsou hezké, zajímavé či alespoň vtipné, leč v praxi naprosto nepoužitelné. Jedním z nejznámějších je Brainfuck, který si své jméno skutečně zasluhuje.

Běh programu v něm si lze představit jako operace nad polem bytů, přičemž je k dispozici jen jeden ukazatel na aktivní buňku, s níž jedinou lze pracovat bez změny ukazatele. K tomu všemu stačí 8 instrukcí reprezentovaných 8 znaky, ostatní se ignorují.

ZOO takovýchto esoterických jazyků je opravdu pestrá. Obsahuje nejen příbuzné Brainfucku (např. Ook!), ale třeba i Malbolge, který se snaží, aby bylo programování v něm co nejobtížnější, INTERCAL, v němž je nutno mimo jiné o provedení příkazu prosit, avšak ne moc, a Whitespace, který využívá jen znaků mezera, tabulátor a nová řádka.

Jelikož má Brainfuck stejnou výpočetní sílu jako jiné běžné jazyky (populárně řečeno), jako demonstraci užitečnosti uveďme jazyk HQ9+ se 4 instrukcemi pokrývajícími typické testovací úlohy pro jazyky, leč nic jiného:

`h` – *vypíše „Hello, world!“,*

`q` – *zobrazí zdrojový kód programu,*

`9` – *vypíše text k písničce „99 Bottles of Beer on the Wall“ (ano, má 100 slok),*

`+` – *zvýší o 1 hodnotu akumulátoru.*

Že programování je umění (dokonce abstraktní) a psát není potřeba, dokazuje Piet, pojmenovaný po holandském malíři Pietu Mondrianovi. Některá jeho abstraktní díla vypadají skoro stejně jako programy v Pietu, reprezentované bitmapou s maximálně 20 barvami.

Pokud vás netradiční programování zaujalo, pěknou sbírku roztodivných jazyků najdete na specializované wiki.[3]

Jak bude vypadat vývoj jazyků v budoucnosti? Někteří tvrdí, že nic nového, převratného do 10 let nepřijde a na špičce se udrží stávající jazyky, které se budou jen pomalu vyvíjet a dále přejímat prvky z funkcionálních jazyků. Třeba nás ale někdo překvapí!

Jedním z nových trendů, jež se nyní rychle rozvíjí, je paralelismus, vycházející z myšlenky rozdělit dlouho trvající výpočty mezi několik procesorů nebo počítačů. Třeba se dají takto prolamovat některé jednodušší šifry.

24-1-7 Distribuované výpočty — 10 bodů

Firma Hack & Crack vlastní N (tedy mnoho) počítačů vzájemně propojených mezi sebou (ne nutně každý s každým). Rozhodla se, že prolomí šifru americké armády, a na výpočet nasadila veškeré síly.

Uběhlo pár dní a programátoři z hrůzou zjistili, že je ve výpočtu chyba a musí se přerušit. Postupně tedy vypínají počítače, chtějí však, aby se v každém okamžiku mohly všechny běžící počítače spolu domluvit (tj. mezi každými dvěma lze přes nějaké jiné poslat zprávu).

Pomozte jim najít pořadí, v němž mají vypínat počítače (očíslované od 1 do N). Na vstupu kromě N dostanete i seznam dvojic kabelem propojených počítačů (propojení je obousměrné).

Můžete předpokládat, že na začátku lze poslat zprávu mezi každými dvěma počítači. Je-li řešení více, stačí najít jen jedno.

Příklad: ve firmě je 7 počítačů a propojené jsou `1-2, 1-3, 2-3, 3-4, 3-5, 3-6, 3-7, 5-6, 6-7`. Řešením je například vypínat v pořadí `4, 5, 7, 6, 3, 1, 2` nebo `2, 5, 6, 7, 4, 1, 3` a nebo mnoha jinými způsoby.

Ve vzdálené budoucnosti by se klidně mohlo programovat v angličtině nebo i v češtině. Hello world by vypadal třeba takto:

```
Vypiš „Dobrou noc, světe!" (bez uvozovek)
a skonči.
```

Co byste říkali na takovýto program?

```
Vyřeš všechny úlohy z 1. série.
Zapiš jejich řešení po jednom do PDF.
Odevzdej řešení přes web KSP.
```

Zatím jediným alespoň trochu funkčním překladačem češtiny se zdají být čeští programátoři. Dokážete napsat kompilátor pro počítač, který bude dobrý alespoň jako člověk, co neprogramuje?

Povídání o jazycích sepsal Pavel Veselý.

[2] `http://ksp.mff.cuni.cz/zaciname/codex.html`

[3] `http://esolangs.org/wiki/`

24-1-8 Pojďte pane, budeme si hrát — 14 bodů

Letos se bude seminářem jako červená nit proplétat seriál o hrách a jejich matematickém a výpočetním řešení. To důležité, co si z něj můžete odnést, je přehled o tom, jakým způsobem současné počítače získávají náskok před lidským rozumem a v jakém vztahu mohou koexistovat chytrá pozorování a hrubá výpočetní síla.

Definovat si, co znamená *hra,* zní nanejvýš otravně, ale je to pojem tak obecný, že s nějakým vymezením začít musíme. Nebo od nás čekáte, že budeme studovat, jak počítačem řešit schovávanou?

- Mějme právě dva hráče.
- Hráči se střídají v tazích.
- Každý tah se vybírá z konečné sady možností. Piškvorky tedy budeme nazývat hrou jen pro předem omezenou velikost čtverečkovaného papíru.
- Oba hráči znají celou informaci o hře, takže žádný z nich neskrývá karty.
- Průběh hry je závislý pouze na tazích hráčů, takže neexistuje náhoda a nehází se kostkou.

Můžeme začít!

Matematika funguje

Přestože víme, že počítače jsou v šachách lepší než lidé, neplatí, že by šachy byly vyřešená hra – neví se totiž, že by nějaká strategie zaručovala vítězství proti libovolnému oponentovi.

Existují hry, jako je *anglická dáma,* které vyřešené jsou, ale tak nějak „suše". Máme v nich strategii, která zaručí, že nikdy neprohrajeme, nejde však o elegantní matematický nápad, jako spíš o velmi dlouhý seznam (či spíš strom) pravidel.

Vzhledem k tomu, jak arbitrární jsou pravidla oblíbených her, nedá se ani čekat, že by pro ně někdo někdy takový hezký matematický nápad našel. Existují ale hry *matematické.* Říká se jim tak, protože mají pravidla formulovatelná v řeči matematiky tak snadno či příznivě, že očekáváme, že by krásná řešení mít mohly.

Jednu takovou matematickou hru si ukážeme. Překvapivě, tuto hru lidé občas stále hrají – a hráli dlouho předtím, než byla jako důležitá matematická hra rozpoznána.

Mějme tři hromádky nerozlišitelných žetonů. Hráči se střídají v tom, že z jedné hromádky seberou a zahodí libovolné nenulové množství žetonů. Prohrává ten, na kterého žádný žeton nezbude.

Když si tuto hru zkusíte zahrát, zjistíte, že úplně triviální není. Má však elegantní matematické řešení, které nám dává rychlou metodu, jak určit, jestli má hráč na tahu zajištěnou výhru a pokud ano, jak by měl táhnout.

Zjednodušme si situaci a redukujme počet hromádek na dvě. Jak hrát tuto hru je nasnadě, ale rozmyslete si to. Pokušení číst dál, aniž byste řešení našli, je třeba odolat, protože spoilery v matematice hrají stejně zápornou roli, jako spoilery u filmů s důležitým zvratem.

Takovou hru má vyhranou první hráč na tahu právě tehdy, je-li na hromádkách rozdílný počet žetonů. Táhnout bude tak, že sebere z početnější hromádky tolik žetonů, aby počet dorovnal. Protivníkovi tak nezbude, než rovnost opět porušit.

To se bude opakovat do té doby, než hráč, co dostává situaci s rozdílným počtem žetonů, dostane jednu z hromádek prázdnou – vyhraje pak sebráním celé druhé hromádky. Hráči, co dostává situaci s tím samým počtem žetonů, se něco takového evidentně stát nemůže – jedním tahem nikdy dvě neprázdné hromádky neodstraní.

Dobře tedy! Při třech hromádkách budeme hledat podobné *smutné stavy* hry,

- do nějakého z nich bude mít jeden hráč možnost druhého vždy uvrhnout z každého stavu, který smutný není,
- které budou zahrnovat prázdnou (prohrávací) pozici,
- a všechny tahy ze smutných pozic vedou do pozic, které smutné nejsou.

Řešením je vyjádřit si počty žetonů na hromádkách jako binární čísla a provést po číslicích jejich XOR (ten jsme milou náhodou zavedli už v úloze 24-1-1). Stav jako smutný označíme tehdy, vyjde-li nám nula.

Úkol 1 [5b]: Ověřte, že taková definice splňuje tři požadavky, které jsme měli.

Uvědomte si, že tímto pozorováním získáme strategii, jak hru se třemi hromádkami vyhrát pokaždé, když nejsme ve smutném stavu, kdy nad námi naopak bude moci vždy vyhrát protivník.

Můžeme si také všimnout, že popsaná strategie pro dvě hromádky je ve skutečnosti ten samý postup. Ještě zajímavější je, že nám strategie funguje pro libovolný konečný počet hromádek!

Generování možných tahů

V druhé části seriálu se zaměříme na hry, které efektivně vyřešit neumíme. Zřejmě se nebudeme snažit, aby za nás počítač *pochopil,* jaké strategie jsou dobré, protože počítač je v chápání opravdu nemožný. Co mu naopak velmi jde, je procházení všech možností.

Máme výchozí situaci a chceme udělat první půltah (půltah je zahrání jednoho hráče a tah je půltah hráče společně s následujícím půltahem protihráče). Můžeme si spočítat, jak bude vypadat deska po každém z možných půltahů, a uvažovat nad tím, je-li to pro nás dobrá pozice. Asi ale tušíte, že z toho mnoho nezjistíme. Potřebujeme rozmýšlet na více tahů dopředu.

Dobře. Nagenerujeme všechny možné situace desky po 8 půltazích. Třeba rekurzivně:

Funkce *generuj* (*pozice*, *hloubka*, kdo je *na tahu*):

Pokud je $hloubka = 0$ nebo je pozice vyhrávající
či prohrávající:
    Vypiš *pozice*.

Pro všechny možné tahy t hráče, který je *na tahu*,
z *pozice*:
    generuj (pozice po tahu t, $hloubka - 1$, druhý hráč)

Co nevidíme, je tam pozice, ve které jsme vyhráli!

Slavnostně si vybavíme, jaká sekvence půltahů vede do této pozice a zahrajeme první z nich. Ale co se nestalo, protivník se svou brzkou záhubou nesouhlasí a hraje jinam, do pozice, která pro nás nevypadá dobře.

Minimax aneb „O tom nerozhoduješ!"

Byli jsme nerozvážní. Nemůžeme si jen tak vybrat, kam se dostat – musíme počítat s tím, že naše možnost ovlivňovat hru je jaksi poloviční a navíc že protivník je inteligentní a druhá polovina tahů povede proti našemu zájmu.

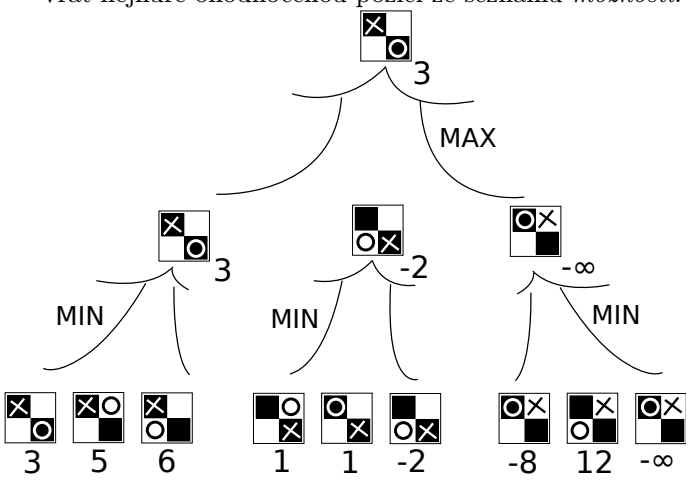
Nalezení prostého maxima z nalezených pozic se tedy vyvarujeme. Využijeme zvoleného rekurzivního způsobu generování tahů a budeme si vracet maxima tam, kde máme volbu, a minima tam, kde volbu nemáme a kde očekáváme, že půjde protihráč proti nám.

Funkce *generuj* (*pozice*, *hloubka*, kdo je *na tahu*):

Pokud je *hloubka* $= 0$ nebo je pozice vyhrávající či prohrávající:
  Vrať *pozice*.

Zavedeme prázdný seznam *možnosti*.
Pro všechny možné tahy t hráče, který je *na tahu*, z *pozice*:
  Přidej do *možnosti* hodnotu
  generuj (pozice po tahu t, *hloubka* $- 1$, druhý hráč).

Pokud jsem *na tahu* já:
  Vrať nejlépe ohodnocenou pozici ze seznamu *možnosti*.
Jinak:
  Vrať nejhůře ohodnocenou pozici ze seznamu *možnosti*.



Tomuto algoritmu se obvykle říká *minimax*. Abychom ho mohli implementovat, potřebujeme počítači vysvětlit, jak poznat, která situace je pro nás lepší, než jiná. Obvyklou volbou je napsat funkci, která pozici přiřadí hodnotu z množiny reálných čísel obohacených o hodnoty $+\infty$ a $-\infty$, které slouží jako indikace „výhry“ a „prohry“. Vysoká kladná čísla budou znamenat, že je pozice velmi dobrá, záporná, že je pozice slabá.

Kvality této *ohodnocovací funkce* budou do značné míry ovlivňovat kvalitu výsledného algoritmu. Uvědomme si, že minimax zaručuje, že se dostaneme do relativně dobře ohodnocené situace, pokud si ale ohodnocovací funkce spletla dobrý skutečný stav věcí se špatným, prohráváme.

Pokud bychom mohli minimax spustit neomezeně půltahů hluboko, do konce každé hry, stačilo by, aby ohodnocovací funkce poznala vítězství a prohru, a náš algoritmus by hrál nejlépe možně – pokud by mohl vyhrát, vyhrál by. To však není realistické očekávat – s prohledáváním o každý půltah hlouběji se běh násobně zpomalí.

Ohodnocovací funkce může pozici zkoumat podrobně a strávit nad tím hodně času, minimax ale potom nestihne postoupit do tak hluboké úrovně, jako kdyby bylo ohodnocování pozic povrchní a nespolehlivé. To, jak pečlivé zkoumání stojí za to zvolit, záleží především na povaze řešené hry.

Druhá důležitá funkce generuje možné půltahy. Ve většině her je jen několik málo půltahů rozumných a u části nerozumných půltahů to můžeme algoritmicky rozeznat ihned, aniž bychom pouštěli minimax.

Kupříkladu v piškvorkách nemá cenu hrát příliš daleko od bojiště. Nemůžeme si být jisti, jestli některá optimální strategie s takovým dalekým tahem nepočítá, ale ani ve hrách velmi dobrých hráčů se nic takového nevyskytuje a my si tak jen na základě tohoto pozorování můžeme dovolit násobně zmenšit počet vygenerovaných tahů.

Dobře zvolit, které situace z generátoru pouštět a které ne, je důležité rozhodnutí, protože jím omezujeme, jak moc se nám bude strom volání větvit, tedy (opět) jak hluboko budeme moci propočítávat.

Úkol 2 [7+2b]: Napište ohodnocovací funkci a generátor možných tahů pro hru šestvorky na desce 15×15. Tato hra se od běžných piškvorek liší ve dvou „drobnostech“:

- Vyhrává linie šesti, ne pěti značek.
- Hráči pokládají hned dvě své značky ve svém půltahu místo jedné, s výjimkou úplně prvního tahu začínajícího hráče, při kterém se pokládá značka toliko jedna. Je to hezké pravidlo, protože činí hru vyrovnanější.

Od 21. srpna (tedy od skončení nulté série) do uzávěrky série této bude na stránkách semináře aréna, ve které se budou vaše funkce zasazené do minimaxového algoritmu bít. Dozvíte se tam i technické detaily. Zde uvádíme jen,

- že čas na vyhodnocení jednoho tahu bude zhruba deset sekund,
- že pozici budete dostávat jako obyčejné dvojrozměrné pole a že žádné jiné informace nebudete smět použít (nepůjde si ukládat data mezi ohodnoceními)
- a že programovacím jazykem bude Python.

S účastí není potřeba zvlášť spěchat, protože doba, po kterou se bude algoritmus soutěže účastnit, závěrečné vyhodnocení sama od sebe neovlivní. Přirozeně je dobrý nápad zkusit si včas, jak si na tom stojíte, a tak se v případě neuspokojivého výsledku motivovat k další práci.

Všechna řešení, která překonají námi připravenou dvojici funkcí, dostanou sedm bodů. Zbylé dva body rozdělíme podle umístění v tabulce.

Lukáš Lánský

Recepty z programátorské kuchařky

Časová a paměťová složitost

V této kuchařce se můžete dočíst o základech časové a paměťové složitosti. Po přečtení byste měli být schopni sami rozebrat složitost jednoduchých algoritmů. To se hodí třeba při návrhu algoritmů a řešení algoritmických úloh, které můžete potkat například v KSP.

Nejdříve si ujasníme, co to ta složitost vlastně je, a ukážeme si pár příkladů. Pak si řekneme, s jakou přesností budeme složitost chtít určovat, a zavedeme si asymptotickou složitost. Na závěr si ukážeme běžné třídy složitosti.

Základní přehled

Pokud řešíme nějakou programátorskou úlohu, často nás napadne více různých řešení a potřebujeme se rozhodnout, které z nich je „nejlepší“. Abychom to mohli posoudit, potřebujeme si zavést měřítka, podle kterých budeme různé algoritmy porovnávat. Nás u každého algoritmu budou zajímat dvě vlastnosti: čas, po který algoritmus běží, a paměť, kterou při tom spotřebuje.

Čas nebudeme měřit v sekundách (protože stejný program na různých počítačích běží rozdílnou dobu), ale v počtu provedených operací. Pro jednoduchost budeme předpokládat,

že aritmetické operace, přiřazování, porovnávání, apod. nás stojí jednotkový čas. Ona to není úplná pravda, tyto operace se ve skutečnosti přeloží na procesorové instrukce, které se teprve zpracovávají. Ale nám postačí vědět, že těch instrukcí bude vždy konstantní počet. A později se dozvíme, proč nám na takové konstantě nezáleží.

Množství použité paměti můžeme zjistit tak, že prostě spočítáme, kolik bytů paměti náš program použil. Nám obvykle bude stačit menší přesnost, takže všechna čísla budeme považovat za stejně velká a velikost jednoho prohlásíme za jednotku prostoru.

Jak čas, tak paměť se obvykle liší podle toho, jaký vstup náš program zrovna dostal – na velké vstupy spotřebuje více času i paměti než na ty malé. Budeme proto oba parametry určovat v závislosti na velikosti vstupu a hledat funkci, která nám tuto závislost popíše. Takové funkci se odborně říká *časová (případně paměťová, někdy též prostorová) složitost* algoritmu/programu.

Nyní si na příkladu ukážeme, jak se časová a paměťová složitost dá určovat intuitivně, a pak si vše podrobně vysvětlíme.

Představme si, že máme danou posloupnost N celých čísel, ze které chceme vybrat maximum. Použijeme algoritmus, který za maximum prohlásí nejprve první číslo posloupnosti. Pak toto maximum postupně porovnává s dalšími čísly posloupnosti a pokud je některé větší, učiní z něj nové maximum. Zapsat bychom to mohli třeba takto:

```
posl[1...N] = vstup
max = posl[1]
Pro i = 2 až N:
   Jestliže posl[i] > max:
      max = posl[i]
Vypiš max
```

Není těžké nahlédnout, že algoritmus provede maximálně $N - 1$ porovnání. Intuitivně časová složitost bude lineárně záviset na N, protože porovnání dvou čísel nám zabere „jednotkový čas“ a paměťová složitost bude také na N záviset lineárně, protože si každé číslo z posloupnosti budeme uchovávat v paměti. Pokud bychom si nepamatovali celou posloupnost, ale vždy jen poslední přečtený člen, stačilo by nám jen konstantně mnoho proměnných, takže paměťová složitost by klesla na konstantní (nezávislou na N) a časová by zůstala stejná.

Jiný příklad: Mějme dané číslo K. Naším úkolem je vypsat tabulku všech násobků čísel od 1 do K:

```
Pro i = 1 až K:
   Pro j = 1 až K:
      Vypiš i*j a mezeru
   Přejdi na nový řádek
```

Tabulka má velikost K^2 a na každém jejím políčku strávíme jen konstantní čas. Proto časová složitost bude záviset na čísle K kvadraticky, tedy bude K^2. Paměťová složitost bude buď konstantní, pokud hodnoty budeme jen vypisovat, anebo kvadratická, pokud si tabulku budeme ukládat do paměti. Můžeme si také všimnout, že tabulku nemusíme vypisovat celou, ale bude nám stačit jen její dolní trojúhelníková část – i tak budeme muset spočítat $(K \cdot K - K)/2 + K = K^2/2 + K/2$ hodnot, což je stále řádově kvadratické vzhledem ke K.

U výběru algoritmu tedy bereme v potaz čas a paměť. Který z těchto faktorů je pro nás důležitější, se musíme rozhodnout vždy u konkrétního příkladu. Často také platí, že čím více času se snažíme ušetřit, tím více paměti nás to pak stojí. To kvůli chytré reprezentaci dat v paměti a různým vyhledávacím strukturám, o kterých se můžete dočíst v našich dalších kuchařkách. Nás u valné většiny algoritmů bude nejdříve zajímat časová složitost a až poté složitost paměťová. Paměti mají totiž dnešní počítače dost, a tak se málokdy stane, že vymyslíme algoritmus, který má dokonalý čas, ale nestačí nám na něj paměť. Ale přesto doporučujeme dávat si na paměťová omezení pozor.

Než se pustíme do podrobnějšího vysvětlování, ještě si ukážeme tzv. „metodu kouknu a vidím“, kterou můžeme použít na určování časové složitosti u těch nejjednodušších algoritmů. Spočívá jen v tom, že se podíváme, kolik nejvíc obsahuje náš program vnořených cyklů. Řekněme, že jich je k a že každý běží od 1 do N. Potom za časovou složitost prohlásíme N^k.

Vzhledem k čemu budeme složitosti určovat?

Složitosti obvykle určujeme vzhledem k velikosti vstupu (počet čísel, případně znaků na vstupu). Tento počet si označme N. Časovou i paměťovou složitost pak vyjádříme vzhledem k tomuto N. To je vidět třeba na výběru maxima v předchozím textu.

Pokud by existovalo několik vstupů stejné velikosti, pro které náš algoritmus běží různě dlouho, bude časová složitost popisovat ten nejhorší z nich (takový, na kterém algoritmus poběží nejpomaleji). Stejně tak pro paměťovou složitost použijeme ten ze vstupů délky N, na který spotřebujeme nejvíce paměti. Dostaneme tzv. složitosti v nejhorším případě. Podrobněji si o tom povíme později.

Někdy se nám hodí určit složitost v závislosti na více než jedné proměnné. Pokud bychom například chtěli vypisovat všechny dvojice podstatného a přídavného jména ze zadaného slovníku, strávíme tím čas, který bude záviset nejen na celkové velikosti slovníku, ale i na tom, kolik obsahuje podstatných a kolik přídavných jmen. Rozmyslete si, jaká složitost vyjde, pokud víte, že velikost slovníku je S, podstatných jmen je A a přídavných jmen B.

Častým příkladem, kde si velikost vstupu potřebujeme rozdělit do více proměnných, jsou algoritmy pracující s grafy (viz grafová kuchařka).[4] V případě grafů obvykle vyjadřujeme složitost pomocí proměnných N a M, kde N je počet vrcholů grafu a M je počet jeho hran. I pro více proměnných vybíráme nejhorší případ.

Ne vždy ale určujeme složitosti v závislosti na velikosti vstupů. Například pokud je velikost vstupu konstantní, složitost určíme vzhledem k hodnotám proměnných na vstupu. Třeba u příkladu s tabulkou násobků jsme složitost určili vzhledem k velikosti tabulky, kterou jsme dostali na vstupu. Jiným příkladem může být vypsání všech prvočísel menších než dané N.

Asymptotická složitost

V této části textu se budeme věnovat pouze časové složitosti. Všechna pravidla, která si řekneme, pak budou platit i pro paměťovou složitost.

U určování časové složitosti nás bude především zajímat, jak se algoritmy chovají pro velké vstupy. Mějme například algoritmus A o časové složitosti $4N$ a algoritmus B

[4] http://ksp.mff.cuni.cz/tasks/20/cook3.html

o složitosti N^2. Tehdy je sice pro $N = 1, 2, 3$ algoritmus B rychlejší než A, ale pro všechna větší N ho už algoritmus A předběhne. Takže pokud bychom si měli mezi těmito algoritmy zvolit, vybereme si algoritmus A.

U složitosti nás obvykle nebude zajímat, jak se chová na malých vstupech, protože na těch je rychlý téměř každý algoritmus. Rozhodující pro nás bude složitost na maximálních vstupech (pokud nějaké omezení existuje) anebo složitost pro „hodně velké vstupy“. Proto si zavedeme tzv. **asymptotickou časovou složitost.**

Představme si, že máme algoritmus se složitostí $n^2/4+6n+12$. Pod asymptotikou si můžeme představit, že nás zajímá jen nejvýznamnější člen výrazu, podle kterého se pak pro velké vstupy chová celý výraz. To znamená, že:

- Konstanty u jednotlivých členů můžeme škrtnout (např. $6n$ se chová podobně jako n). Tím dostáváme $n^2 + n + 1$.
- Pro velká n je $n+1$ oproti n^2 nevýznamné, tak ho můžeme také škrtnout. Dostáváme tak složitost n^2. Obecně škrtáme všechny členy, které jsou pro dost velké n menší než nějaký neškrtnutý člen.

Tahle pravidla sice většinou fungují, ale škrtat ve výpočtech přece nemůžeme jen tak. Proto si nyní zavedeme operátor $\mathcal{O}$ (velké O), díky kterému budeme umět popsat, co přesně naše „škrtání“ znamená, a používat ho korektně.

Definice: Mějme funkce $f : \mathbb{N} \to \mathbb{R}^+$ a $g : \mathbb{N} \to \mathbb{R}^+$. Řekneme, že $f \in \mathcal{O}(g)$, pokud $\exists n_0 \in \mathbb{N}$ a $\exists c \in \mathbb{R}^+$ tak, že $\forall n \geq n_0$ platí $f(n) \leq c \cdot g(n)$.

Nyní slovy: Mějme funkce f a g funkce z přirozených do kladných reálných čísel. Řekneme, že funkce f patří do třídy $\mathcal{O}(g)$, pokud existují konstanty n_0 a c takové, že f je pro dost velká n (totiž pro $n \geq n_0$) menší než $c \cdot g(n)$.

Někdy také píšeme, že $f = \mathcal{O}(g)$ nebo říkáme, že program má složitost $\mathcal{O}(f)$.

A zde je použití: $n^2/4+6n+12 \in \mathcal{O}(n^2)$, protože například pro $c = 10$ platí pro všechna $n > 1$ (tedy $n_0 = 2$):

$$n^2/4 + 6n + 12 \leq 10n^2.$$

Pokud vám tento způsob nevyhovuje a více se vám líbí metoda pomocí „škrtání“, tak ji klidně používejte, akorát všude pište $\mathcal{O}(\ldots)$. Někdy také říkáme, že se konstanty a méně významné členy v $\mathcal{O}$ ztrácí.

Ještě poznamenejme, že operátor $\mathcal{O}(\ldots)$ znamená asymptotický horní odhad funkce. Takže pokud funkce patří do $\mathcal{O}(N)$, tak pak patří i do $\mathcal{O}(N^2)$, $\mathcal{O}(N^3)$, ...

Nejhorší a průměrný případ

Opět si vše vysvětlíme jen na časové složitosti.

Velká část algoritmů běží pro různé vstupy stejné velikosti různou dobu. U takových algoritmů pak můžeme rozlišovat složitost v nejhorším případě (tu už známe), v nejlepším případě a třeba i průměrnou časovou složitost.

Vše si ukážeme na algoritmu BubbleSort (bublinkovém třídění), o kterém se můžete dočíst v kuchařce o třídících algoritmech.[5] Funguje tak, že se dívá na všechny dvojice sousedních prvků a kdykoliv je dvojice ve špatném pořadí, tak ji prohodí. Zde je pseudokód algoritmu:

```
BubbleSort(pole, N):
   Opakuj:
      setříděno = 1
      Pro i = 1 až N-1:
         Jestliže pole[i] > pole[i+1]:
            p = pole[i]
            pole[i] = pole[i+1]
            pole[i+1] = p
            setříděno = 0
   Skonči, až bude setříděno = 1
```

Časová složitost v nejhorším případě činí $\mathcal{O}(N^2)$ – v každém průchodu vnějším cyklem nám totiž největší hodnota „probublá“ na konec a ostatní se posunou o jednu pozici doleva. Rozmyslete si, proč. Průchodů je proto nejvýše $N-1$ a každý z nich trvá $\mathcal{O}(N)$. Tento nejhorší případ může doopravdy nastat, pokud necháme setřídit klesající posloupnost. Tam provedeme přesně $N-1$ průchodů.

Naopak v nejlepším případě bude časová složitost pouze $\mathcal{O}(N)$. To nastane, pokud na vstupu dostaneme už setříděnou posloupnost. U té algoritmus pouze zkontroluje všechny dvojice a pak se ihned zastaví.

Průměrná časová složitost nám udává, jak dlouho náš algoritmus běží průměrně. Co to ale znamená, není snadné definovat ani spočítat. U třídícího algoritmu bychom mohli počítat průměr přes všechny možnosti, jak mohou být prvky na vstupu zamíchané (tedy přes všechny jejich permutace). To nám někdy může dát přesnější odhad chování algoritmu.

Zrovna u BubbleSortu a mnoha jiných algoritmů vyjde průměrná časová složitost stejně jako složitost v nejhorším případě. Jedním z nejznámějších příkladů algoritmu, který je v průměru asymptoticky lepší, je třídící algoritmus QuickSort (opět viz třídicí kuchařka). Jeho průměrná časová složitost činí $\mathcal{O}(N \cdot \log N)$, zatímco v nejhorším případě může běžet až kvadraticky dlouho.

Často používané složitosti

Na závěr si ukážeme často se vyskytující časové složitosti algoritmů (ty paměťové jsou obdobné). Seřadili jsme je od nejrychlejších a ke každé připsali příklad algoritmu.

$\mathcal{O}(1)$ – *konstantní* (třeba zjištění, jestli je číslo sudé)

$\mathcal{O}(\log N)$ – *logaritmická* (binární vyhledávání); všimněte si, že na základu logaritmu nezáleží, protože platí $\log_a n = \log_b n / \log_b a$, takže logaritmy o různých základech se liší jen konstanta-krát, což se „schová do $\mathcal{O}$-čka“.

$\mathcal{O}(N)$ – *lineární* (hledání maxima z N čísel)

$\mathcal{O}(N \cdot \log N)$ – *lineárně-logaritmická* (nejlepší algoritmy na třídění pomocí porovnávání)

$\mathcal{O}(N^2)$ – *kvadratická* (BubbleSort)

$\mathcal{O}(N^3)$ – *kubická* (násobení matic podle definice)

$\mathcal{O}(2^N)$ – *exponenciální* (nalezení všech posloupností délky N složených z nul a jedniček; pokud je chceme i vypsat, dostaneme $\mathcal{O}(N \cdot 2^N)$)

$\mathcal{O}(N!)$ – *faktoriálová*, $N! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot N$ (nalezení všech permutací N prvků, tedy třeba všech přesmyček slova o N různých písmenech)

Složitosti ještě často rozdělujeme na *polynomiální* a *nepolynomiální*. Polynomiální říkáme těm, které patří do $\mathcal{O}(N^k)$ pro nějaké k. Naopak nepolynomiální jsou ty, pro něž žádné takové k neexistuje.

Do polynomiálních algoritmů patří například i algoritmus se složitostí $\mathcal{O}(\log N)$. A to proto, že $\mathcal{O}(\log N) \subset \mathcal{O}(N)$

[5] http://ksp.mff.cuni.cz/tasks/20/cook2.html

(každý algoritmus, který seběhne v čase $\mathcal{O}(\log N)$, seběhne i v $\mathcal{O}(N)$).

Nepolynomiální jsou z naší tabulky třídy $\mathcal{O}(2^N)$ a $\mathcal{O}(N!)$. Takové algoritmy jsou extrémně pomalé a snažíme se jim co nejvíce vyhýbat.

Pro představu o tom, jak se složitost projevuje na opravdovém počítači, se podíváme, jak dlouho poběží algoritmy na počítači, který provede 10^9 (miliardu) operací za sekundu. Tento počítač je srovnatelný s těmi, které dnes běžně používáme. Podívejme se, jak dlouho na něm poběží algoritmy s následujícími složitostmi:

funkce / $n =$	10	20	50	100	1 000	10^6
$\log_2 n$	3.3 ns	4.3 ns	4.9 ns	6.6 ns	10.0 ns	19.9 ns
n	10 ns	20 ns	30 ns	100 ns	1 μs	1 ms
$n \cdot \log_2 n$	33 ns	86 ns	282 ns	664 ns	10 μs	20 ms
n^2	100 ns	400 ns	900 ns	100 μs	1 ms	1 000 s
n^3	1 μs	8 μs	27 μs	1 ms	1 s	10^9 s
2^n	1 μs	1 ms	1 s	10^{21} s	10^{292} s	$\approx \infty$
$n!$	3 ms	10^9 s	10^{23} s	10^{149} s	10^{2558} s	$\approx \infty$

Pro představu: 1 000 s je asi tak čtvrt hodiny, 1 000 000 s je necelých 12 dní, 10^9 s je 31 let a 10^{18} s je asi tak stáří Vesmíru. Takže nepolynomiální algoritmy začnou být velmi brzy nepoužitelné.

Dnešní menu servírovali
Karel Tesař a Martin Mareš

Jak řešit úlohy – Často kladené dotazy

„U Elektrona Svatýho, co myslel tímhle?“

„A časovou složitost má kde?“

„Proč by tohle mělo fungovat?“

Takové a mnohé další dotazy si my, organizátoři KSP, klademe při opravování došlých řešení. Bohužel, někdy na ně odpovědi nenajdeme, a proto ze zvědavosti strhneme několik bodů. Sice se mnozí řešitelé časem naučí, co dělat mají a co ne, aby získali co nejvíce bodů, ale chvíli to trvá a některé při tom ztratíme.

Pro ulehčení nováčkům jsme tedy připravili tento návod. Představme si modelovou úložku. Úkolem je spočítat největšího společného dělitele dvou čísel (předstírejte na chvíli, že jste to ještě nikdy neřešili). Na ní si ukážeme postup, jak dojít ke správnému řešení, a také pár tipů, co nedělat.

Napřed je samozřejmě potřeba řešení vymyslet. První, co nás pravděpodobně napadne, je zkusit vydělit obě čísla tím menším z nich. Pokud po dělení je nějaký nenulový zbytek, pak to není největší společný dělitel a my zkusíme číslo o 1 menší. A tak dále, dokud nepotkáme takové, které beze zbytku dělí obě. To je samozřejmě společný dělitel a je první, kterého jsme potkali, takže je největší.

Takový postup má mnohé výhody – je jednoduchý, zcela očividně vrátí správný výsledek, a navíc máme jistotu, že někdy skončí (zastavíme se určitě nejpozději u jedničky). Ale jde to i rychleji (co znamená „rychleji“, můžete najít výše v kuchařce).

Zapomeňme na rozklad na prvočísla, který je na první pohled příliš komplikovaný, než aby měl šanci na úspěch. Vezmeme dvě zadaná čísla. Pokud se rovnají, pak jsou (obě) největším společným dělitelem. Pokud ne, to větší z nich zmenšíme o to menší a pokračujeme stejně.

Navíc pokud bude jedno číslo obrovské a druhé maličké, budeme to maličké odečítat opakovaně, až získáme… zbytek po dělení většího menším. To by mohlo výpočet ještě zrychlit.

Dobrá, postup máme. Co teď? Nyní je vhodné napsat vlastní program v nějakém jazyce. My organizátoři ho sice nepožadujeme „povinně“, ale pomůže odhalit nedostatky (či nedomyšlené „zrady“) v algoritmu.

Například na našem příkladě bychom zjistili, že zatímco odečítací metoda funguje, metoda zbytková se bude pokoušet dělit nulou (alespoň tak, jak jsme ji popsali). V nějakou chvíli již bude v menším čísle uložený výsledek. Při odčítání dojdeme postupně ke stejnému číslu – ale při dělení získáme zbytek 0 jedinou operací. V takovou chvíli je třeba skončit. Navíc nám program pomůže pochopit méně jasné části popisu.

Program by vypadal třeba takto (zapíšeme ho v Pascalu):

```
var x, y: integer;
begin
   read(x, y);
   while (x<>0) and (y<>0) do
      if x>y then x := x mod y
             else y := y mod x;
   writeln(x+y);
end.
```

(všimněte si malého triku: když je na konci jedno z čísel `x`, `y` nulové, tak `x+y` je rovno tomu nenulovému).

Nakonec je třeba vytvořit text řešení. Co by měl obsahovat? Určitě popis algoritmu, a to takový, aby kdokoliv, kdo umí jen trochu programovat, podle něho byl program schopný napsat. Při tomto popisu lze použít nějaký již existující algoritmus jako stavební kámen, například se odkázat na nějakou knížku nebo programátorské kuchařky z webu KSPčka.

Pokud tento popis bude nejasný nebo nejednoznačný, pokusíme se nějakou myšlenku vykoukat z přiloženého programu, avšak už za nedostatečný popis nejspíš pár bodů ztratíte.

Další částí by mělo být nějaké zdůvodnění, proč vlastně program počítá, co se po něm chce. Určitě nám ještě nevěříte, že popsaná magie s odčítáním funguje. My mnohým tvrzením, která nám dojdou, také ne (některému až tak moc, že si dáme práci ho vyvrátit).

Co by bylo důkazem v tomto případě? Třeba následující textík (zapsaný opravdu důkladně, jako formální důkaz, obvykle však stačí myšlenka):

Tvrdíme, že v každém kroku algoritmu nahradíme větší z čísel x, y číslem $|x-y|$ a zachováme přitom všechny společné dělitele dvojice x, y, tím pádem samozřejmě i největšího společného dělitele.

Jakmile se algoritmus zastaví (což zajisté učiní), držíme v ruce dvě čísla, z nichž jedno je nula (a ta je beze zbytku dělitelná čímkoliv), a tedy největší číslo, které dělí obě, je to druhé, nenulové.

Zbývá tedy dokázat, že jeden krok algoritmu zachovává všechny společné dělitele. Mějme nějakého společného dělitele d čísel x, y. Navíc předpokládejme, že $x > y$, takže x budeme nahrazovat číslem $z = x \bmod y$ (kdyby $x < y$, tak

jen prohodíme x s y). Co z toho víme:

$$x = x' \cdot d$$
$$y = y' \cdot d$$
$$z = x - t \cdot y$$

pro nějaká x', y', t. Číslo z tedy můžeme upravovat takto: $z = x - ty = x'd - ty'd = (x' - ty')d$. Takže z je také dělitelné číslem d.

Nechť naopak nějaké d dělí dvojici z, y. Pak víme:

$$z = z' \cdot d$$
$$y = y' \cdot d$$
$$x = z + t \cdot y,$$

takže $x = z'd + ty'd = (z' + ty')d$ je také dělitelné číslem d. Zjistili jsme tedy, že společní dělitelé dvojic x, y a z, y jsou titíž.

Nakonec je potřeba odhadnout, jak dobrý algoritmus jsme vymysleli. K tomu slouží odhady časové a paměťové složitosti. Jelikož jsou velmi důležité, věnovali jsme jim letos celý jeden díl receptů z programátorské kuchařky, který najdete hned nad tímto textem.

Pokud máte určování složitostí v malíčku nebo jste si přečetli kuchařku, můžete se podívat na pokračování vzorového řešení úložky s největším společným dělitelem:

Paměťová složitost našeho algoritmu je zcela očividně konstantní – máme jen dvě proměnné na čísla, v nich provádíme veškeré operace.

Časová bude chtít trošku odhadovat. Jednak, co je velikostí vstupu? Tou bude součet velikostí obou čísel, tedy $n = x + y$ (délka výpočtu totiž nezávisí na počtu čísel na vstupu – tam je jich vždy stejně – ale na jejich hodnotách). V každém kroku se jedno z čísel sníží alespoň o 1 a nikdy se nedostaneme do záporných čísel. Takže bychom mohli klidně psát, že časová složitost je $\mathcal{O}(n)$ – určitě náš program nepoběží déle.

To je sice pravda, ale moc jsme se nevytáhli – stejnou časovou složitost měl i původní algoritmus se zkoušením všech potenciálních dělitelů. Tak co teď? Vymyslet lepší algoritmus? Ne, my na to půjdeme šalamounsky – vymyslíme lepší důkaz.

Opět předpokládejme, že přecházíme od dvojice x, y ke dvojici z, y, kde $z = x \bmod y$. Dokážeme, že $z \leq x/2$, takže každým krokem algoritmu se aspoň jedno z čísel zmenší aspoň dvakrát. Přitom kroků, kdy se dvakrát zmenšilo původní x, může být celkem nejvýš $\lfloor \log_2 x \rfloor$, a analogicky pro y. Proto je celková časová složitost $\mathcal{O}(\log_2 x + \log_2 y) = \mathcal{O}(\log n)$.

A proč je $z \leq x/2$? Rozebereme dvě možnosti: buďto je $y \leq x/2$, ale pak stačí využít toho, že zbytek po dělení je vždy menší než dělitel, tedy $z < y \leq x/2$. A nebo je $y > x/2$, ale pak $z = x - y \leq x - x/2 = x/2$.

Na závěr dodejme, že popsanému algoritmu na počítání největšího společného dělitele se říká Eukleidův.

Několik špatných pokusů

Minulá část obsahovala popis, jak vypadá správné řešení. Nyní zmíníme několik chyb, se kterými se celkem pravidelně při opravování setkáváme.

Jednou (a asi nejvážnější) z nich je, když nám přijde pouze zdrojový kód, který je občas (ale sporadicky) komentovaný a není k tomu žádný popis. Popis má být hlavní částí řešení, zdrojový kód pouze doplňkem.

Opačný extrém je příliš podrobné (a komplikované) vyprávění či slohová práce. Opravdu neplatí, že čím delší text, tím více bodů. Úlohy v KSP jsou dělané tak, aby se daly jednoduše popsat na stránku nebo dvě. Řešení o 20 stranách je tak dlouhé, že v něm prostě něco špatně být musí.

Další, celkem běžný, problém je špatně pochopitelný popis. Zkuste si text po sobě přečíst – s vědomím, že člověk, který ho bude číst, možná vaše řešení vůbec nezná a nic o něm neví. Nejlépe s odstupem několika hodin či dní.

Do této oblasti patří i pravopis (někdy špatně umístěná nebo chybějící čárka ve větě může úplně změnit význam) a v případě psaní ručně i čitelnost rukopisu (za to, co nepřečteme, body nedáme).

A, samozřejmě, plný počet bodů nedostanete ani za řešení, které nefunguje.

Co dělat když...

Mnoho lidí KSP řešit nezačne, přestože by je třeba i bavilo. Většinou proto, že narazí na nějaký problém, který ale obvykle není tak neřešitelný, jak vypadá.

Pokud vám některá úloha přijde příliš těžká, nezoufejte. Snažíme se dávat i „šťavnaté“ úložky pro pokročilejší řešitele – samozřejmě ne všechny, některé jsou lehké. K jejich rozpoznání vkládáme do zadání značky, jejichž popis najdete na začátku letáku.

A co v případě, když vás napadne jen pomalé řešení? Je na něm sice jasně vidět, že jsme při zadávání mysleli na něco rychlejšího, ale vy na to ne a ne přijít. Rozhodně napište alespoň to pomalé – zatímco za nefunkční řešení nedáváme skoro nic, za pomalejší řešení dáváme docela dost (tedy, podle toho, jak moc pomalejší je).

Nakonec malý tip. Zkuste začít řešit s předstihem. Velmi pomáhá, když je pár dní času na to, aby pěkné řešení „napadlo“.