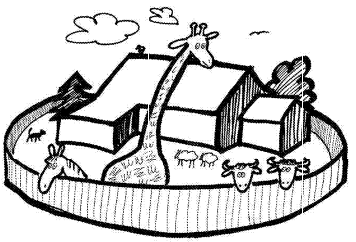
INFORMATIKA

Oplocení farmy

(Úlohy z MO – kategorie P, 26. část)

PAVEL TÖPFER

Matematicko-fyzikální fakulta UK, Praha



Stejně jako v předchozím dílu našeho seriálu, také tentokrát si ukážeme řešení jedné zajímavé úlohy z krajského kola 59. ročníku Matematické olympiády – kategorie P (školní rok 2009/2010). Rovněž tato úloha pochází z dílny slovenských organizátorů programátorské olympiády z Fakulty matematiky, fyziky a informatiky Univerzity Komenského v Bratislavě. Jako obvykle se nejprve seznámíme s úplným zadáním soutěžní úlohy.

⋆⋆⋆

Přemysl se doslechl, že se v zemědělství točí velké peníze, a rozhodl se, že v něm také začne podnikat. Netrvalo dlouho a už vlastnil rozlehlou farmu, na níž pěstoval množství zajímavých plodin. Přemyslova zemědělská půda má obdélníkový tvar a je rozdělena na $R \times S$ stejně velkých čtvercových políček. Na každém políčku se pěstuje jedna plodina.

Nedávno se farma ocitla v nebezpečí, neboť v jejím okolí se přemnožili zajíci, kteří si rádi pochutnávají na pěstovaných rostlinách. Proto se Přemysl rozhodl, že na farmě nechá postavit plot, který ochrání plody jeho práce. Jelikož v okolí není velká konkurence v oblasti stavebnictví, Přemyslovi se podařilo sehnat pouze jeden kontakt – firmu Čtverce s. r. o., která

se specializuje na stavební práce čtvercového charakteru. Firma Čtverce s. r. o. nabídla, že postaví na farmě plot, který ochrání zvolenou čtvercovou část pozemku.

Přemysl přemýšlí, kde má nechat plot postavit. Plot může chránit čtvercové území libovolné velikosti. Musí ale vést po hranicích mezi políčky, takže každé políčko ochrání buď celé, nebo vůbec. Přemysl navíc požaduje, aby plot chránil políčka s alespoň dvěma různými plodinami. Chce mít totiž jistotu, že nezůstane na trhu jenom s jedním produktem. Pomozte Přemyslovi zjistit, kolik má možností na postavení plotu.

Formát vstupu

Na prvním řádku vstupu jsou zadány rozměry pozemku – počet řádků R a počet sloupců S, a dále počet plodin K, které je možné na farmě pěstovat. Platí $1 \leq R, S \leq 2500$, $1 \leq K \leq 1\,000\,000\,000$. Na každém z dalších R řádků vstupu je vždy uvedeno S čísel – popis, které plodiny se pěstují na jednotlivých políčkách. Plodiny jsou označeny čísly od 0 do $K - 1$.

Formát výstupu

Vypište jediné číslo – kolika způsoby může Přemysl postavit na farmě plot, který ohradí čtvercovou oblast, na níž se pěstují aspoň dvě různé plodiny.

Příklad

Vstup:

```
3 6 10
1 0 0 0 1 7
2 0 0 0 7 7
3 0 0 0 7 7
```

Výstup:

```
8
```

V situaci popsané v tomto příkladu máme pět možností, jak lze postavit plot kolem oblasti velikosti 2×2, a tři možnosti, jak lze postavit plot kolem oblasti velikosti 3×3.

⋆⋆⋆

Oplocená čtvercová oblast má jistě stranu délky alespoň 2, neboť v oblasti velikosti 1×1 by nebylo možné pěstovat dvě různé plodiny. Délka její strany nemůže být větší než minimum z hodnot R, S, jelikož větší čtverec se na pozemek farmy ani nevejde. Primitivní řešení úlohy může vypadat tak, že postupně vyzkoušíme všechny velikosti oplocené oblasti z uvedeného rozmezí od 2 do $\min(R, S)$ a pro každou velikost všechna možná umístění takové oblasti na pozemku. Umístění lze jednoznačně určit například polohou pravého dolního rohu oblasti, takže počet takových umístění shora odhadneme výrazem $R \cdot S$. Každou uvažovanou volbu oplocení otestujeme tak, že celou oplocenou oblast projdeme a zkontrolujeme, zda obsahuje aspoň dvě různé plodiny – to vyžaduje provést práci úměrnou počtu políček obsažených ve zkoumané čtvercové oblasti. Celková časová složitost takového řešení je tedy $O(R \cdot S \cdot (\min(R, S))^3)$.

S tak velkou časovou složitostí se samozřejmě nespokojíme a budeme hledat způsob, jak výpočet urychlit. Značná neefektivita popsaného primitivního řešení spočívá v tom, že jednotlivá políčka na pozemku farmy testujme opakovaně mnohokrát, jelikož jsou součástí mnoha různě velkých a různě umístěných čtvercových oblastí. Tomuto opakovanému procházení políček budeme chtít zabránit. Při návrhu lepšího řešení úlohy využijeme techniku dynamického programování, která spočívá v tom, že nejprve vyřešíme vhodně zvolené dílčí podúlohy, pomocí jejich výsledků pak řešíme podúlohy další, atd., až se dostaneme k řešení celé původní úlohy. S metodou dynamického programování se můžete blíže seznámit například v učebnici [1] nebo ve studijním textu [2], využili jsme ji již vícekrát i v seriálu úloh z MO kategorie P v našem časopise.

Ještě jednou si připomeneme, že každou čtvercovou oblast farmy můžeme jednoznačně identifikovat polohou jejího pravého dolního rohu a délkou její strany. Dále si uvědomíme, že snadněji se nám budou počítat jednobarevné čtverce, tedy čtvercové oblasti, na nichž se pěstuje jediná plodina. Hledaný počet všech různých vícebarevných čtverců (tzn. čtvercových oblastí farmy obsahujících vždy aspoň dvě různé plodiny) pak určíme jednoduše tak, že od počtu všech čtverců odečteme počet jednobarevných čtverců. Kdybychom pro každý bod (r, s) znali počet jednobarevných čtverců, jejichž pravý dolní roh se nachází právě na políčku (r, s), dokázali bychom hledaný výsledek snadno spočítat. Počet všech čtverců (jednobarevných i vícebarevných dohromady) s pravým dolním rohem (r, s) je totiž roven minimu z hodnot r, s. Počet vícebarevných čtverců pro pravý dolní

roh (r, s) spočítáme tedy jako rozdíl $\min(r, s)$ minus počet jednobarevných čtverců pro (r, s)

Symbolem $K(r, s)$ si označíme počet všech jednobarevných čtverců, jejichž pravý dolní roh leží na políčku (r, s). Hodnota $K(r, s)$ je zároveň rovna délce strany největšího jednobarevného čtverce, který má pravý dolní roh na (r, s). Je totiž zřejmé, že jakmile je některý čtverec s pravým dolním rohem na políčku (r, s) vícebarevný, jsou vícebarevné i všechny větší čtverce umístěné na (r, s), neboť ty ho celý obsahují.

Nyní si ukážeme, jak lze hodnoty $K(r, s)$ efektivně počítat. Uvažujme tři sousední políčka $(r, s-1)$, $(r-1, s)$ a $(r-1, s-1)$. Je-li alespoň na jednom z těchto políček pěstována jiná plodina než na políčku (r, s), potom je hodnota $K(r, s)$ jistě rovna jedné, jelikož čtverec s délkou strany dva už je vícebarevný. V případě, že se plodiny pěstované na uvedených třech políčkách shodují s plodinou na políčku (r, s), potom hodnotu $K(r, s)$ můžeme jednoduše vyjádřit podle vztahu

$$K(r, s) = 1 + \min(K(r-1, s), K(r, s-1), K(r-1, s-1))$$

Zbývá zdůvodnit, proč tento vztah platí. Čtverec A s pravým dolním rohem na políčku (r, s) a se stranou délky $x + 1$ v sobě obsahuje čtverce B, C, D se stranou velikosti x a s pravým dolním rohem po řadě na políčkách $(r, s-1)$, $(r-1, s)$ a $(r-1, s-1)$. Navíc sjednocení ploch těchto tří čtverců B, C, D tvoří celou plochu původního čtverce A, až na rohové políčko (r, s). Z předpokladu víme, že pravé dolní rohové políčko všech čtyř uvažovaných čtverců má stejnou barvu. Jsou-li tedy tři čtverce B, C, D jednobarevné, mají nutně všechny stejnou barvu a tutéž barvu má i celý čtverec A. V našem případě $x = \min(K(r-1, s), K(r, s-1), K(r-1, s-1))$ a tedy skutečně $K(r, s) = x + 1$.

Protože k výpočtu hodnot $K(r, s)$ stačí znát vždy jenom hodnoty $K(r-1, s)$, $K(r, s-1)$, $K(r-1, s-1)$, můžeme postupovat po pozemku farmy po řádcích shora dolů a v rámci každého řádku zleva doprava. Každou hodnotu $K(r, s)$ přitom spočítáme v konstantním čase s využitím hodnot, které už známe. Naše řešení úlohy má proto časovou složitost $O(R \cdot S)$. Z hlediska časové složitosti je toto řešení jistě asymptoticky optimální, neboť jenom samotné zpracování vstupních dat vyžaduje provedení takovéhoto množství operací. Kdybychom si načetli a uložili celý vstup, program by měl také paměťovou složitost $O(R \cdot S)$. Paměťovou složitost ale můžeme snížit na $O(R)$, když si uvědomíme, že stačí udržovat v paměti pouze dva

řádky vstupu a dva řádky počítaných hodnot $K(r, s)$, jelikož předcházející řádky již nebudeme k dalšímu výpočtu potřebovat.

```
program OploceniFarmy;

var R, S, K: integer; {rozměry pozemku, počet plodin}
    Pocet: longint;   {výsledný počet možností}
    D, M: array[0..2500, boolean] of integer;
       { D[s] -- plodina pěstovaná na políčku
         M[s] -- počet jednobarevných čtverců
         dva exempláře pole pro poslední dva řádky pozemku }
    i, j: integer;
    novy: boolean;

function min(a, b: integer): integer;
begin
if a < b then
  min:=a
else
  min:=b
end;

begin
read(R, S, K);
{Nahoře a vlevo doplníme políčka s indexem 0,
na nich je pěstována fiktivní plodina číslo K}
for j:=0 to S do
  D[j,false]:=K;
novy:=true;
for i:=1 to R do
  begin
  D[0,novy]:=K;
  for j:=1 to S do
    begin
    read(D[j,novy]);
    if (D[j-1,novy] <> D[j,novy])
    or (D[j,not novy] <> D[j,novy])
    or (D[j-1,not novy] <> D[j,novy]) then
```

```
      M[j,novy]:=1
    else
      M[j,novy]:=1 + min(min(M[j-1,novy], M[j,not novy]),
                         M[j-1,not novy]);
    Pocet:=Pocet + min(i,j) - M[j,novy]
    end;
  novy:=not novy   {posun řádku}
  end;
writeln(Pocet)
end.
```

Literatura

[1] *Töpfer, P.*: Algoritmy a programovací techniky, Prometheus, Praha 2007 (2. vydání)

[2] Korespondenční seminář z programování MFF UK, studijní materiály, http://ksp.mff.cuni.cz/tasks/21/cook5.html

(Autorkou úvodní ilustrace je Mgr. Jaroslava Čermáková.)

Proč při programování nemusí platit některá pravidla známá z hodin matematiky

DENIS MAINZ – VÁCLAV VRBÍK

Pedagogická fakulta Západočeské univerzity v Plzni

Při úvodních hodinách programování lze poměrně zajímavým způsobem ilustrovat a doplnit výklad o reálných číslech z hodin matematiky. V programování bývá zvykem rozlišovat čísla na obor čísel „celých“ a „reálných“. Je tedy vhodné seznámit žáky s tímto faktem hned v úvodu, aby

si vytvořili představu o vyskytujících se problémech, jež jsou s ním obvykle spejené. K tomuto účelu využijeme následující jednoduché příklady, vytvořené v programovacím jazyku Java. Tyto mohou být s patřičnými úpravami sestaveny i v jiných programovacích jazycích.

Pro práci s celými čísly v jazyce Java se obvykle používá datový typ *int*. Následující jednoduchý příklad nevykazuje po spuštění žádný problém. Pokud však žáci použijí pro proměnné A, B hodnoty $A := 320000$; $B := 280000$, bude vytištěna hodnota mezivýsledku – proměnné $V1 = -594313216$. Nechte žáky experimentovat s jinými hodnotami proměnných A, B.

```
public class    Cis1{
  public static   void main(String[] args){
    int A = 32;
        int B = 28;
        int V1 = A * B;
        System.out.println(V1);
  }
}
```

Je málo pravděpodobné, že žáci sami objeví příčinu tisku nesprávného mezivýsledku pro některé hodnoty A, B. Vysvětlete jim tedy, že pro zobrazení celých čísel v počítači slouží v případě Javy „paměť“ o velikosti 32 bitů. Kladná čísla jsou zobrazena v paměti počítače ve dvojkové soustavě, záporná čísla se obvykle ukládají v doplňkovém kódu. Nejvyšší bit slouží pro znaménko (je-li roven 0, je číslo kladné, je-li roven 1, je číslo záporné). Lze tedy zobrazit celá čísla v intervalu od -2^{31} do $2^{31} - 1$, tj. od $-2\,147\,483\,648$ do $2\,147\,483\,647$. Žákům je třeba vysvětlit, že na rozdíl od matematiky se při programování pracuje s oborem celých čísel v jistém intervalu.

Pro práci s reálnými čísly je nejčastěji využíván datový typ *double*. Tento datový typ přináší řadu omezení. Slouží pro práci s reálnými čísly jen v jistém intervalu přípustných hodnot a čísla se zobrazují jen na jistý předem omezený počet platných cifer. V počítači jsou tato čísla obvykle uložena ve dvojkové soustavě v podobě znaménka, mantisy a exponentu. Mantisa obsahuje číslo v tzv. normalizovaném tvaru, nejvyšší řád mantisy je nenulový a desetinná tečka je umístěna před nejvyšším řádem mantisy. Exponent udává, o kolik míst je třeba posunout řádovou čárku. Je-li hod-

nota exponentu kladná, je absolutní hodnota zobrazovaného čísla větší než 1 a desetinnou čárku je nutné posunout doprava. Je-li hodnota exponentu záporná, je absolutní hodnota zobrazovaného čísla menší než 1 a desetinnou čárku posuneme doleva. V tomto okamžiku však není naším cílem seznámení s detailním způsobem vyobrazení reálných čísel v počítači. Důležité je pochopení skutečnosti, že pro zobrazení mantisy slouží omezený počet bitů (je tak určen maximální počet platných cifer, na které se číslo zobrazuje) stejně jako v případě exponentu, pro který je rovněž vyhrazen pouze omezený počet bitů (je tak určen rozsah hodnot, které lze zobrazit). Uvedený typ prezentace reálných čísel se nazývá *zobrazení v pohyblivé řádové čárce*. V případě jazyka Java slouží u datového typu double pro zobrazení mantisy 52 bitů, pro znaménko 1 bit a exponentu náleží 11 bitů. Lze tak zobrazit čísla v rozsahu od $-4.94065645841246544 \cdot 10^{-324}$ do $1.7976931348623157 \cdot 10^{308}$ přibližně na 15 až 16 platných cifer.

Při výpočtech s reálnými čísly je dále nutné si stále uvědomovat způsob jejich reprezentace v počítači. Jinak hrozí, že teoreticky správné algoritmy budou při jejich naprogramování dávat vlivem zaokrouhlovacích chyb nesprávné výsledky. Později se žáci seznámí s některými algoritmy numerické matematiky, u kterých bude nutná analýza zaokrouhlovacích chyb a vysvětleny metody jejich eliminace. Přesto je nutné žákům hned na počátku ukázat několik problémů, se kterými by se mohli setkat. Z důvodu omezeného paměťového místa pro uložení reálných čísel, je možné zobrazit v omezeném intervalu jen konečný počet různých hodnot (z matematického hlediska je počet všech reálných čísel nekonečný). Omezen je počet platných cifer a přípustný rozsah řádů cifer reálného čísla. Reálná čísla jsou tedy v programu nahrazena racionálními čísly z jistého intervalu a s pevně daným počtem platných cifer. S čísly, která nepatří do uvedeného intervalu, nelze vůbec pracovat. Z matematického hlediska tedy můžeme **přesně** zobrazit v počítači jen čísla z jisté konečné, i když značně rozsáhlé podmnožiny *racionálních čísel*, označme ji M. Ostatní reálná čísla x (tj. $x \in M$) jsou při uložení do paměti zaokrouhlena a nahrazena vždy nejbližším číslem $x_0 \in M$ (jde o nejbližší hodnotu datového typu *double*). K ozřejmení problémů, proč některé výpočty počítače dávají „chybné“ výsledky, poslouží následující série příkladů.

První příklad ilustruje situaci, kdy při přičtení velmi malého čísla k číslu 1 se hodnota tohoto velmi malého čísla v součtu vůbec neprojeví.

```
public class    Cis2{
```

```
  public static   voidmain(String[]args){
    double X = 1.0;
    double Y = 0.00000000000000007;
    double V = X + Y;
    System.out.println("X = "+ X);
    System.out.println("Y = "+ Y);
    System.out.println("V = "+ V);
  }
}
```

Po provedení příkazu výstupu se vytiskne:

```
X = 1.0
Y = 7.0E-17
V = 1.0
```

Příčinu, proč se hodnota Y v součtu vůbec neprojeví, lze spatřovat v příliš velkém rozdílu mezi exponenty čísel X a Y. Při zobrazování výsledku V v počítači vzniká potřeba „rozšířit mantisu“, ale protože to technicky není možné, bity se „vpravo mimo rozsah mantisy“ zanedbají.

Další příklad je obdobou předchozího. Hodnota proměnné X je v tomto příkladu 10^{16} a přičtení jedničky se v součtu vůbec neprojeví. Příklad zároveň ilustruje, že by se při programování neměl provádět test na rovnost čísel datového typu reprezentujícího obor reálných čísel.

```
public class    Cis3{
  public static   void main(String[]args){
    double X = 1E16;
    if (X == X+1){
      System.out.println("X = X + 1");
    }
  }
}
```

Při experimentování s předchozími příklady mohou žáci dospět k chybné představě, že problém zaokrouhlovacích chyb vyplývá z velikosti zobrazovaného čísla. Často si myslí, že počítač nedovede rozlišit pouze čísla, jejichž absolutní hodnota rozdílu je velmi malá. Následující příklad ilustruje, že problém zaokrouhlovacích chyb ve skutečnosti plyne z omezeného počtu platných cifer zobrazovaného čísla.

```
public class    Cis4{
  public static   void main(String[]args){
    double X = 1.00000000000000001E+20;
    double Y = 1.00000000000000000E+20;
    System.out.println(X+","+Y);
  }
}
```

Po spuštění programu se v obou případech na výstupu vypíše 1.0E20. Hodnoty proměnných X, Y se zobrazí stejným způsobem, i když hodnota proměnné X bude např. 1.00000000000000001000E+20. K zajímavému jevu dojde také v případě, že z oblasti čísel za desetinnou čárkou odebereme u proměnné X v původním zadání jednu nulu. Výsledkem pak při výpisu uvedené proměnné bude číslo 1.0000000000000002E20.

Následující příklad ilustruje situaci, kdy při počítání s reálnými čísly typu *double* nemusí vždy platit asociativní zákon.

```
public class    Cis5{
  public static   void main(String[]args){
    double X = 1;
    double Y = 1E16;
    double Z = -1E16;
    System.out.println(X+(Y+Z));
    System.out.println((X+Y)+Z);
  }
}
```

V prvním případě se vytiskne: 1.0
Ve druhém případě se vytiskne: 0.0

Ukázkou zobrazení racionálního čísla $\frac{1}{3}$ v počítači může být následující příklad:

```
public class    Cis6{
  public static   void main(String[]args){
    double X = 1./3.;
    System.out.printf("%.11f"  ,  X);
```

```
    System.out.println();
    for (int  i = 1; i < 20; i++){
      X = (X - 0.3) * 10;
      System.out.printf(" %.11f",  X);
      System.out.println();
    }
  }
}
```

Při prvním příkazu výstupu se zobrazí hodnota proměnné X: 0.33333333333. To by mohlo znamenat, že zobrazení racionálního čísla (vytiskne se ve tvaru desetinného čísla s teoretickou hodnotou $0,\overline{3}$) v počítači je přesné. V následujícím cyklu for dochází nejprve k odečtení hodnoty 0.3 od původní hodnoty proměnné X a následnému vynásobení číslem 10. Zobrazený výsledek by tedy měl být stále stejný (0.33333333333). Žákům je třeba tuto operaci ještě před spuštěním programu vysvětlit, například pomocí zápisu na tabuli, tím způsobem, že vlastně v cyklu odečítáme první trojku za desetinnou tečkou a po vynásobení číslem 10 posouváme všechny další trojky o jedno místo doleva. Protože desetinný rozvoj je v našem případě teoreticky nekonečný, mělo by se stále zobrazovat stejné číslo. Skutečnost je však taková, že se zobrazí následující hodnoty:

0,33333333333	0,33333333272	0,32716542764
0,33333333333	0,33333332717	0,27165427641
0,33333333333	0,33333327165	− 0,28345723590
0,33333333333	0,33333271654	− 5,83457235903
0,33333333333	0,33332716543	− 61,34572359029
0,33333333333	0,33327165428	
0,33333333327	0,33271654276	

Žáci mohou s programem experimentovat, neměli by však měnit výstupní formát v příkazu výstupu (zobrazení na 11 desetinných míst). Celá záhada spočívá v tom, že $\frac{1}{3} \notin M$ a po zadání hodnoty příkazu *double* $X = 1./3.$ se v počítači nezobrazí $\frac{1}{3}$, ale jiné číslo $x_0 \in M$, byť k jedné třetině blízké. Tímto vzniká chyba zobrazení čísla, která se při každém dalším násobení logicky také násobí, a následně zvětšuje.

Závěrem partie o použití celých a reálných čísel je nutné zdůraznit, že pokud je možné použití celých čísel (datový typ *int*, případně další celočíselné typy), je vhodné tento typ neopouštět. Jakýkoli i dočasný přechod k typu double (případně dalším reálným typům) znamená především nebezpečí vzniku zaokrouhlovacích chyb. Co nejdříve je nutné seznámit žáky se skutečností, že pro operaci dělení celých čísel existuje operace „celočíselné dělení“, kdy výsledkem operace s celými čísly je opět celé číslo.

Na závěr si ukážeme úlohu navazující na problematiku zobrazení čísel v počítači a varianty jejího řešení. Jde o výpočet faktoriálu. Jedná se o jednu z nejfrekventovanějších úloh, která se objevuje v různých fázích výuky programování.

```
public class  Faktorial1{
  public static   void main(String[]args){
    Scanner sc = new
    Scanner(System.in);
    System.out.println("Zadej přirozené číslo: ");
    int N = sc. nextInt();
    int Fakt = 1;
    for (int  i = 0; i < n; i++){
      Fakt = Fakt * (i + 1);
    }
    System.out.println(N+"! = "+ Fakt);
  }
}
```

Program je přesným přepisem správného algoritmu, který je uváděn v nejedné publikaci a na mnoha internetových stránkách věnujících se programování v tom, či onom programovacím jazyce. Z tohoto důvodu by mohl málokdo předpokládat, že nebude správně řešit danou úlohu. Výsledky získané tímto programem jsou následující:

$11! = 39\,916\,800 \qquad 12! = 479\,001\,600 \quad 13! = 1\,932\,053\,504$

Pro ověření správnosti výsledků může být použita např. vědecká kalkulačka, která je součástí příslušenství operačního systému Windows. Žáci dojdou k závěru, že výsledek programu se s hodnotou na kalukalčce neshoduje v 13!. Ti vnímavější se mohou dokonce dostat do situace, kdy už si nebudou jisti ani výsledkem vědecké kalkulačky. V takových případech

už pomůže snad jen papír, tužka a násobení pod sebe.

Tento moment, kdy počítač nepočítá správně, je možné využít k podrobnějšímu osvětlení pojmu datový typ. Typ *int* je charakterizován nejen množinou přípustných hodnot, ale i množinou přípustných operací, které lze s daty daného typu provádět. Na rozdíl od matematiky je množina hodnot typu *int* konečná (představuje interval $-2\,147\,483\,648$ až $2\,147\,483\,647$). Hodnota $13! = 6\,227\,020\,800$ leží mimo uvedený interval a počítač ji tudíž neumí zobrazit.

Otázkou však stále zůstává, jak sestavit nebo vylepšit program tak, aby počítal skutečně správné výsledky nejen do 12!, ale také hodnoty mnohem vyšší, např. 50!. Žáci přestanou pohlížet na počítač jako na neomylného počtáře a začnou se zajímat o řešení programu, který by jim umožnil počítat správný faktoriál. První možností, kterou obvykle zvolí, je „zvětšení" intervalu zobrazitelných hodnot. Seznámíme tedy žáky s největším celočíselným datovým typem, který můžeme v Javě použít, a tím je *long*, jehož rozsah je -9 223 372 036 854 775 808 až 9 223 372 036 854 775 807. Z následujících výsledků můžeme pozorovat, že je situace se zadáváním větších hodnot o něco lepší:

$13! = 6227020800$	...
$14! = 87178291200$	...
$15! = 1307674368000$	...
$16! = 20922789888000$	...
$17! = 355687428096000$	...
$18! = 6402373705728000$	$65! = -9223372036854775808$
$19! = 121645100408832000$	$66! = 0$
$20! = 2432902008176640000$	$67! = 0$
$21! = -4249290049419214848$	$68! = 0$
$22! = -1250660718674968576$	$69! = 0$
$23! = 8128291617894825984$	$70! = 0$

Výsledky až do 20! jsou správné, výsledky 21! až 65! jsou chybné (malý počet platných číslic a výskyt záporných hodnot). Od 66! je výsledek už jen nula. I když byl použit „největší" možný celočíselný datový typ, příliš daleko jsme se nedostali. Žáci brzy přijdou na myšlenku „prodloužit číslo" využitím reálných datových typů. Jazyk Java nabízí dva reálné typy:

float	$-1.40129846432481707 \cdot 10^{-45}$ až $3.40282346638528860 \cdot 10^{38}$	$2-8$ platných číslic

double	$-4.940656455841246544 \cdot 10^{-324}$ až $1.79769313486231570 \cdot 10^{308}$	2 – 17 platných číslic

Můžeme provádět další pokusy s jednotlivými reálnými typy. Nabízí se také příležitost seznámit žáky např. s problematikou softwarové emulace matematického koprocesoru (pokud není k dispozici). Při použití typu *double* obdržíme následující výsledky:

15! = 1.307674368E12	21! = 5.109094217170944E19
16! = 2.0922789888E13	22! = 1.1240007277776077E21
17! = 3.55687428096E14	23! = 2.585201673888498E22
18! = 6.402373705728E15	24! = 6.204484017332394E23
19! = 1.21645100408832E17	25! = 1.5511210043330986E25
20! = 2.43290200817664E18	26! = 4.0329146112660565E26

Výsledky se zdají přesné. Pozorný žák si však brzy povšimne, že od 22! dochází ke ztrátě přesnosti vinou toho, že se nezobrazí číslice na nejnižších řádech (jsou „odřezávány“). Příliš daleko jsme se tedy opět nedostali. Pokud tedy použijeme „delší“ reálné typy, situace se příliš nezmění. Neomylný počítač v očích žáků selhal, nedokáže vyřešit jednoduchou úlohu, jakou je výpočet faktoriálu. V tomto momentu je třeba studentům vysvětlit, že chyba není v počítači, ale ve strategii řešení. Výsledky součinu $1 \cdot 2 \cdot \ldots n$ jsme v předchozích případech uchovávali jako jednu hodnotu.

Nyní budeme výsledek uchovávat jako posloupnost hodnot (délka posloupnosti bude určena pouze délkou pole, které pro uložení použijeme, výsledek se ukládá do pole *Cislice*):

```
public class  Faktorial2{
  public static final int Max = 5000;
  public static int[] Cislice = new int[Max];
  public static void main(String[]args){
    Scanner sc = new Scanner(System.in);
    System.out.println("Zadej přirozené číslo: ");
    int n = sc.nextInt();
    for (int i = 0; i < Max - 1; i++){
      Cislice[i] = 0;
    }//nulování číslic
    Cislice[4999] = 1;
```

```
    int Nej = 1;
    int soucin = 0;
    int des = 0;
    int zb = 0;
    for (int i = 2; i < n + 1; i++){
      for (int j = 0; j < Nej; j++){
        soucin = Cislice[4999 - j] * i + zb;
        des = soucin / 10;
        zb = soucin % 10;
        Cislice[4999 - j] = zb;
        if(soucin > 9){
          zb = des;
        if(j == Nej - 1)
          Nej++;
        }
        else
           zb = 0;
      }
    }
    for (int i = 4999 - Nej + 1; i < 5000; i++){
      System.out.print(Cislice[i]);
    }
    System.out.println("\n Pocet cislic: "+ Nej );
  }
}
```

Že je problém vyřešen, dokazuje následující výsledek:
50! = 30414093201713378043612608166064768844377641568960512000000000, počet číslic je 65.

Literatura

[1] *Töpfer, P.*: Algoritmy a programovací techniky. 1. vydání. Prometheus Praha, 1995.
[2] *Vrbík, V.*: Programování 1. Západočeská univerzita v Plzni, 2000.