INFORMATIKA

Kapitola z kombinatoriky

STANISLAV TRÁVNÍČEK

Přírodovědecká fakulta UP, Olomouc

V článku [1] byl uveden program *Permu*, který vytvářel a vypisoval všechny permutace n zadaných řetězců, a to v „abecedním uspořádání“, v němž každý řetězec byl reprezentován pořadím svého zadání. Z hlediska programového šlo o klasické použití rekurze a článek se pak zmínil o souvislosti permutací s jazykovými jevy.

S použitím rekurze v počítačovém programu se čtenář MFI zde setkal i později, zejména v článcích [2] (4 příklady na použití rekurzivních funkcí a procedur), [3] (úloha MO na hledání kritické cesty v grafu), [4] (zde jsou použity tři do sebe vložené rekurze), [5] (procedura Pruchod řídí v úloze MO průchod robota soustavou potrubí), [6] (v úloze MO je použita rekurzivní procedura QuickSort). V článcích [7] a [8] jsme se podívali i na rekurzi v tabulkových kalkulátorech.

Vraťme se znovu k permutacím. Jistě nás napadne otázka, zda lze užitím rekurze nalézt vhodné algoritmy i pro výpis všech permutací s opakováním, variací bez opakování i s opakováním a také kombinací bez opakování a s opakováním. Algoritmy „s opakováním“ jistě musí fungovat i pro případ „bez opakování“, čímž by se počet případů redukoval ze šesti na tři, ovšem dá se očekávat jejich větší složitost. Takže i algoritmy „bez opakování“ mají určité právo na život.

Pro větší názornost práce algoritmů budeme uvažovat permutace (variace, kombinace) *znaků* zadaných ve vstupním řetězci. Pořadí znaků v zadaném řetězci pro nás bude mít význam „abecedního uspořádání“ znaků. Např. v zadaném řetězci *stopa* je prvním znakem abecedy znak „s“, druhým je „t“, ... , pátým je „a“. Pak v abecedním uspořádání pětic je první pětice *stopa*, druhou *stoap*, třetí *stpoa*, další *stpao*, atd. V tomto smyslu

budeme chtít mít naše výsledky abecedně uspořádány. Dialog s počítačem vybavíme jen minimem komfortu a výpis doplníme počítadlem řádků se zastavením po každých dvaceti řádcích.

Permutace s opakováním

Jak už jsme připomněli, klasicky pojatým programem na permutace jsme se zabývali již v [1] (tuto metodu použijeme při výpočtu variací). Vezměme si tedy problém permutací s opakováním.

Úvod programu s deklaracemi je zřejmý; omezíme se na řetězce např. do délky 10 znaků:

```
program PermuOpa;
uses Crt;
const
  MAX = 10;
type
  TypStr = string[MAX];
var
  C: TypStr;
  N, Radek: Integer;
```

Procedura *Tiskni* vypisuje na obrazovku výsledné řetězce, kde pro lepší čitelnost jsou mezi znaky doplněny mezery:

```
procedure Tiskni(A: TypStr);
var I: Integer;
begin {Tiskni}
  for I := 1 to N do Write(A[I], ' ');
  Inc(Radek);
  WriteLn('  ', Radek);
  if (Radek mod 20 = 0) then ReadLn
end; {Tiskni}
```

Procedura *Vytvarej* je klíčová. Dochází při ní k „přesunu“ znaků z řetězce ve druhém parametru do řetězce v prvním parametru a každý takový dokonaný přesun dává novou permutaci. Přitom příkaz `if Pos(B[I], B) = I then` se stará o to, aby se nevytvořila permutace, která tu již byla:

```
procedure Vytvarej(A, B: TypStr);
var
  I: Integer;
begin {Vytvarej}
  if B = '' then Tiskni(A)
  else
    for I := 1 to Length(B) do
      if Pos(B[I], B) = I then
        Vytvarej(A+B[I], Copy(B,1,I-1)+ Copy(B,I+1,N))
end; {Vytvarej}
```

Vlastní program už je pak zcela jednoduchý:

```
begin  {program}
  ClrScr;
  repeat
    Write('Permutace s~opakovanim, vstupni retezec: ');
    ReadLn(C);
    N := Length(C)
  until N in [1..MAX];
  Radek := 0;
  Vytvarej('', C);
  WriteLn('O.K.');
  ReadLn
end.  {program PermuOpa}
```

Variace

Program je postaven na algoritmu podobném jako byl použit v programu *Permu* v [1]. Sestavují se tu abecedně uspořádané permutace z n prvků, v nichž se uvažuje jen prvních m znaků. Vzhledem k abecednímu uspořádání permutací jsou i příslušné variace m-té třídy abecedně uspořádané. Stačí – a to provádí procedura *TiskniNovou* – pustit do výstupu vždy jen tu variaci, která je odlišná od variace předchozí. Např. tvoříme-li variace 3. třídy z pěti prvků *abcde*, vytvářejí se permutace **abc***de*, *abced*, **abd***ce*, *abdec*, **abe***cd*, *abedc*, ... a do tisku jsou přefiltrovány jen „nové“ trojice, tj. ty, co jsou zde vytištěny tučně.

```
program Varia;
uses Crt;
const
  MAX = 10;
type
  TypStr = string[MAX];
var
  M, N, Radek: Integer;
  A, A0: TypStr;

procedure Vymen(var S1, S2: Char);
var
  S: Char;
begin {Vymen}
  S := S1; S1 := S2; S2 := S
end; {Vymen}

procedure TiskniNovou(A: TypStr);
var
  I: Integer;
  Nova: Boolean;
begin {TiskniNovou}
  Nova := false;
  I:=1;
  repeat
    Nova := (A[I]<>A0[I]);
    Inc(I)
  until Nova or (I > M);
  if Nova then
  begin
    for I := 1 to M do
    begin
      Write(A[I],' ');
      A0[I] := A[I]
    end;
    Inc(Radek);
    WriteLn('  ',Radek);
```

```
    if Radek mod 20 = 0 then ReadLn
  end
end; {TiskniNovou}
```

Procedura *Variace* je stejná jako procedura pro vytváření permutací v [1]. Vlastní program jen zařizuje potřebné vstupy a předává práci rekurzivní proceduře Variace.

```
procedure Variace(A: TypStr; V: Integer);
var
  I: Integer;
begin {Variace}
  if V = N then TiskniNovou(A)
  else
    for I := V to N do
    begin
      Vymen(A[V], A[I]);
      Variace(A, V+1)
    end
end; {Variace}

begin {program}
  Radek := 0;
  repeat
    ClrScr;
    Write('Variace, vstupni retezec: ');
    ReadLn(A);
    N := Length(A)
  until N in [1..MAX];
  repeat
    Write('trida variace: ');
    ReadLn(M)
  until M in [1..N];
  Variace(A, 1);
  WriteLn('O.K.');
  ReadLn
end. {program Varia}
```

Variace s opakováním

V tomto programu se vychází z programu *PermuOpa* a provádí se tu stejné filtrování, jak bylo popsáno v programu *Varia*. Úvodní deklarace se liší jen nepatrně

```
program VariaOpa;
uses Crt;
const
  MAX = 10;
type
  TypStr = string[MAX];
var
  C, A0: TypStr;
  M, N, Radek: Integer;
```

Pak následuje procedura *TiskniNovou* stejná jako v programu *Varia*, pak procedura *Vytvarej* jako v programu *PermuOpa* jen s tím rozdílem, že v ní je na příslušném místě použit příkaz

```
 if B = '' then TiskniNovou(A);
```

Vlastní program je pak:

```
begin  {program}
  ClrScr;
  J := 0;
  repeat
    Write('Variace s opakovanim, vstupni retezec: ');
    ReadLn(C);
    N := Length(C)
  until N in [1..MAX];
  repeat
    Write('trida variace: ');
    ReadLn(M)
  until M in [1..N];
  Radek := 0;
  Vytvarej('', C);
  WriteLn('O.K.');
  ReadLn
end.  {program VariaOpa}
```

Kombinace

Zde je situace dosti odlišná. Mohlo by nás sice napadnout sdružit odpovídající si variace do jedné kombinace, např. z variací 134, 143, 314, 341, 413, 431 vytvořit jedinou kombinace, např. 134, avšak vzhledem k abecednímu uspořádání nejdou tyto variace ihned po sobě, takže tudy cesta nevede. Jinou možností je opět vytváření všech variací, přičemž bychom do tisku pustili jen ty, v nichž mají znaky abecední uspořádání. My zde zvolíme ještě jiný postup. Pro zjištění různých kombinací z n prvků m-té třídy budeme vytvářet všechny takové n-tice nul a jedniček, kde je právě m jedniček. Do výstupu pak pošleme m-tice právě těch znaků zadaného řetězce, které svou polohou odpovídají jedničkám v právě zjištěné n-tici nul a jedniček. Např. je-li zadaný řetězec *abcde* a výpočtem dostaneme 01101, vystoupí kombinace *bce* (viz proceduru *DosadTiskni*).

```
program Kombi;
uses Crt;
const
  MAX = 10;
type
  TypStr = string[MAX];
  PoleN = array[1..MAX] of Integer;
var
  M, N, Radek: Integer;
  A: TypStr;
  C: PoleN;

procedure DosadTiskni(A: TypStr);
var
  I: Integer;
begin {DosadTiskni}
  for I := 1 to N do
    if C[I] = 1 then Write(A[I],' ');
  Inc(Radek);
  WriteLn('  ',Radek);
  if Radek mod 20 = 0 then ReadLn
end; {DosadTiskni}
```

Rekurzivní procedura *Vytvor01* generuje běžným způsobem n-tice jedniček a nul. Při vytváření n-tice však po každém vstupu do procedury

(pokud není ještě n-tice dobudována) zjišťuje, kolik jedniček už je v rozpracované n-tici obsaženo. Pokud je jich už m, doplní se na konec n-tice nuly; pokud chybí tolik jedniček, kolik je ještě volných míst, doplní se na konec vytvářené n-tice jedničky. Pro každou vytvářenou n-tici nastane jedna z těchto dvou možností, takže je tím dosaženo toho, že jedniček v každé n-tici je právě m a samozřejmě se tím také zablokují zbytečné další průchody rekurzí.

```
function ZjistiPocet1(QQ: Integer): Integer;
var I, ZP: Integer;
begin {ZjistiPocet}
  ZP := 0;
  for I := 1 to QQ-1 do
    ZP := ZP + C[I];
  ZjistiPocet1 := ZP
end; {ZjistiPocet}

procedure Vytvor01(Q: Integer);var
  IV, JV, Poc1 : Integer;
begin {Vytvor01}
  if Q < N then
  begin
    for IV := 1 downto 0 do
    begin
      Poc1 := ZjistiPocet1(Q);
      C[Q] := IV;
      Inc(Poc1,IV);
      if Poc1 = M then
      begin
        for JV := Q+1 to N do
          C[JV] := 0;
        DosadTiskni(A)
      end
      else
      begin
        if M-Poc1 = N-Q then
        begin
          for JV := Q+1 to N do
```

```
            C[JV] := 1;
          DosadTiskni(A)
        end
        else
          if M-Poc1 < N-Q then Vytvor01(Q+1)
      end
    end
  end
end; {Vytvor01}


begin {program}
  ClrScr;
  Radek := 0;
  repeat
    Write('Kombinace, vstupni retezec: ');
    ReadLn(A);
    N := Length(A)
  until N in [1..MAX];
  repeat
    Write('trida kombinace: ');
    ReadLn(M)
  until M in [1..N];
  Vytvor01(1);
  WriteLn('O.K.');
  ReadLn
end. {program Kombi}
```

Kombinace s opakováním

`program KombiOpa;`

(deklarace jsou stejné jako v programu *Kombi*)

Jak už napovídá jméno následující procedury, postupuje se stejně jako v programu *Kombi*, ale navíc má tato procedura ještě jeden úkol – vyloučit tisk stejných kombinací. To se zajišťuje tak, že pro každý znak odpovídající jedničce se zjišťuje, jestli se takový znak už v zadaném řetězci vyskytl a to s hodnotou 0; v kladném případě se tisk nepovolí a procedura svou činnost končí, aby zkoumala další n-tici, kterou jí dodá procedura *Vytvor01*. Jestliže není nalezen „nevyužitý“ stejný znak, pak proběhne tisk a započítání řádku do výsledku.

```
procedure DosadRozhodniTiskni(A: TypStr);
var
  I, J: Integer;
  Tisk: Boolean;
begin {DosadRozhodniTiskni}
  Tisk := true;
  I := 1;
  repeat
    if C[I] = 1 then
    begin
      for J := 1 to I do
        if (A[J] = A[I]) and (C[J] = 0) then
          Tisk := false
    end;
    Inc(I)
  until (I > N) or not Tisk;
    if Tisk then
    begin
      for I := 1 to N do
        if C[I] = 1 then
          Write(A[I],' ');
      Inc(Radek);
      WriteLn('  ',Radek);
      if Radek mod 20 = 0 then ReadLn
    end
end; {DosadRozhodniTiskni}
```

Další dvě procedury jsou stejné jako v programu *Kombi*, jen v proceduře *Vytvor01* je místo volání *DosadTiskni* volána procedura *DosadRozhodniTiskni*. I vlastní program se shoduje s programem *Kombi*, samozřejmě jen s pozměněným textem při představování.

Ukázali jsme si některé algoritmy pro kombinatoriku s využitím rekurze, které mohou začínající programátory zaujmout. Pracovali jsme jen se znaky, ale vaši studenti se mohou pokusit aplikovat užité algoritmy na případ, kdy by bylo zadáno n řetězců.

Literatura

[1] *Trávníček, S. – Trávníček, P.:* Permutace. MFI, 4 (1994/95), č. 7, s. 321 – 324.

[2] *Trávníček, S. – Trávníček, P.:* O rekurzi. MFI, 7 (1997/98), č. 4, s. 231 – 239.
[3] *Töpfer, P.:* Úlohy z MO – kategorie P (část 5 – využití rekurze). MFI, 10 (2000/01), č. 9, s. 562 – 567.
[4] *Trávníček, S.:* Matematické podklady k rytmickým etudám. MFI, 11 (2001/02), č. 3, s. 168 – 177.
[5] *Töpfer, P.:* Prohledávání grafu. MFI, 11 (2001/02), č. 8, s. 497 – 502.
[6] *Töpfer, P.:* Pořadí programátorů. MFI, 13 (2003/04), č. 8, s. 497 – 502.
[7] *Hvorecký, J. – Trenčanský, I.:* Využitie adaptability tabuľkových kalkulátorov na rekurzívne výpočty. MFI, 8 (1998/99), č. 1, s. 39 – 45.
[8] *Hvorecký, J. – Trenčanský, I.:* Ohraničenia rekurzívnych výpočtov v tabuľkových kalkulátoroch. MFI, 8 (1998/99), č. 7, s. 430 – 435.

Vyčíslitelnost a složitost

HASHIM HABIBALLA – TIBOR KMEŤ

Přírodovědecká fakulta OU, Ostrava – Fakulta prírodných vied UKF, Nitra

(*Dokončení*)

2. Formalizace pojmu algoritmus

Turingův stroj je nejen jednoduchou a přitom zcela exaktní formalizací pojmu algoritmu, ale na druhé straně je i lehce pochopitelný „selským rozumem". Je možné si jej opět jako konečný automat představit i jako fyzický stroj vykonávající instrukce (program). Lze pomocí něj i nahlédnout na zajímavé obecné vlastnosti programů. Jedním z nich je totiž existence tzv. **UNIVERZÁLNÍHO TURINGOVA STROJE (UTS)**. Tento UTS dokáže simulovat libovolný jiný Turingův stroj (pokud se omezíme na jednoduchou abecedu, což ale nesnižuje obecnost). Pokud si opět místo TS představíme například program v Pascalu, pak nám to dává tvrzení, že existuje univerzální pascalovský program, který dokáže simulovat všechny napsané programy v Pascalu. Simulací se zde myslí, že takový univerzální program dostane na vstup kód simulovaného programu, provede ho přesně jako by byl program proveden sám a vrátí výstupy totožné očekávaným výstupům programu. Sestrojit takový univerzální pascalovský program je samozřejmě poměrně složité (i když je to jen otázka času a úsilí), ale právě jednoduchost formalizace TS umožňuje sestrojit takový UTS poměrně rychle a snadno (dokonce to bývá úloha, kterou vysokoškolští studenti řeší jako samostatný domácí úkol!).

Pojem algoritmu je samozřejmě možno formalizovat i jinými prostředky. Jedním z nich je i více matematičtěji orientovaný formalismus, nazvaný jako **PRIMITIVNĚ (OBECNĚ) REKURZIVNÍ FUNKCE (PRF/ORF)**. Tato formalizace bude mít pravděpodobně půvab pro ty čtenáře, kteří jsou více orientovaní na algebraické pojetí vyčíslitelnosti funkcí na množině přirozených čísel. Jejich idea se opírá o dvě základní definice (pro začátek budeme mluvit pouze o primitivně rekurzivních funkcích a později přidáme pojem obecně rekurzivní funkce):

1. Za základní považujeme tyto funkce:

- $o : N \to N, \forall x : o(x) = 0$ *(funkce, která vrací pro jakýkoliv argument 0 – identická nula)*
- $s : N \to N, \forall x : s(x) = x+1$ *(funkce, která vrací pro jakýkoliv argument jeho následující hodnotu – následník)*
- $I_i^{(n)} : N^n \to N, \forall x_1, x_2, ..., x_n : I_i^{(n)}(x_1, x_2, ..., x_n) = x_i$ *(funkce, která vrací i-tý argument – výběr – je nutná pro tvorbu funkcí více proměnných)*

2. Další funkce můžeme skládat pomocí operátorů:

- Operátory substituce S_n^m: $f = S_n^m(g, h_1, ..., h_m)$, kde platí $f(x_1, x_2, ..., x_n) = g(h_1(x_1, ..., x_n), ..., h_m(x_1, ..., x_n))$ *(operátor, který umožňuje skládat funkce)*
- Operátory primitivní rekurze R^n: $f = R^n(g, h)$, kde platí $f(0, x_2, ..., x_n) = g(x_2, ..., x_n)$ a $f(k+1, x_2, ..., x_n) = h(k, f(k, x_2, ..., x_n), x_2, ..., x_n)$ *(operátor, který definuje rekurzivní funkci na základě funkce g – zarážka rekurze pro $k = 0$, h – následující krok rekurze pro $k + 1$ definovaný pomocí k)*

Z těchto základních funkcí můžeme pomocí postupné aplikace operátorů vytvářet složitější funkce.

Příklad 2:

Zkusme se podívat na příklad funkce $f(x_1, x_2) = x_1 + x_2$. Podobně jako Turingův stroj je tento formalismus poměrně primitivní, takže i takto jednoduchou funkci zde musíme sestavit. Myšlenka spočívá v tom, že použijeme rekurzi (která nám nahrazuje cyklus) a pomocí rekurze vždy vytvoříme následníka hodnoty až k nule, kdy dosadíme hodnotu druhého argumentu.

Sekvence vypadá následovně:

$f_1 = s, f_2 = I_2^3, f_3 = S_3^1(s, I_2^3), f_4 = I_1^1, f = R(f_4, f_3)$ přičemž jednotlivé funkce vyjadřují:

- f – je funkce, kterou jsme chtěli vytvořit (sčítání dvou argumentů), přičemž jsme museli aplikovat rekurzi následujícím způsobem; $f(0, x_2) = x_2, f(k+1, x_2) = f(k, x_2) + 1$; což znamená, že x_1 – krát zvýšíme hodnotu x_2 o jedna a použijeme k tomu upravenou funkci následníka
- f_4 – je funkcí výběru prvního argumentu z jednoho, kterou potřebujeme, abychom správně nadefinovali zarážku rekurze funkce f
- f_3 – je upravená funkce následníka pro druhý argument ze tří (použije se v rekurzi dle formální definice operátoru); $f_3(x_1, x_2, x_3) = x_2 + 1$
- f_2 – je potřebná pro vytvoření následníka druhého argument v f_3; jde o výběr druhého argumentu ze tří $f_2(x_1, x_2, x_3) = x_2$
- f_1 – je základní funkcí následník pro jednu proměnnou ($f_1(x_1) = x_1 + 1$)

Na tomto jednoduchém příkladě jste viděli, že notace primitivně rekurzivních funkcí umožňuje možná poněkud složitě, ale zejména exaktně symbolicky odvodit jakoukoliv funkci. I když je tato konstrukce značně odlišná od formalizace algoritmické vyčíslitelnosti pomocí TS, v konečném důsledku můžeme realizovat přesně Turingovsky vyčíslitelné funkce pouze pokud k definici primitivně rekurzivních funkcí přidáme jeden operátor – tzv. operátor minimalizace, čímž dostaneme obecně rekurzivní funkce. Jelikož je přesná formální definice poněkud složitější, omezíme se na jeho vysvětlení laické. Jde o operátor, který umožňuje vytvořit funkci, která vrací nejmenší hodnotu prvního argumentu, kdy je funkční hodnota 0. Například, kdybychom potřebovali při výstavbě určité funkce znát nejmenší hodnotu funkce $f(x_1) = 5 - x_1$ aplikovali bychom operátor minimalizace, který by vytvořil funkci vracející vždy 5. Samozřejmě by to asi v takto jednoduchém příkladě nedávalo smysl, ale tento operátor lze definovat pro libovolný počet proměnných a libovolnou složitost formule.

Další velice zajímavou formalizací pojmu algoritmu jsou takzvané PL-programy. Zmiňujeme se o nich ihned po rekurzivních funkcích nikoliv náhodou. Jak uvidíte, i když jde opět o formalizaci z naprosto jiného pohledu, lze jednoduše ukázat, že jejich výpočetní síla je totožná. PL-programy (jak již název napovídá) jsou formalizací, kterou zřejmě ocení spíše programátorsky orientovaní čtenáři. PL-program je sekvence příkazů, které mohou obsahovat identifikátory (proměnné). Jazyk je opět velmi jednoduchý, v zásadě máme pouze tyto příkazy a elementy:

- Přiřazovací příkaz $X := 0$ (lze přiřadit nulu libovolnému identifikátoru)

- Příkaz inkrementace INC X ($X := X + 1$)
- Příkaz cyklu LOOP X [seznam příkazů] ENDLOOP (provede seznam příkazů X-krát)
- Návěští L, které určuje bod programu
- Příkaz skoku GOTO L (provede v programu skok do bodu L)
- Specifikace vstupů a výstupu INPUT X_1, X_2, OUTPUT Y

Příklad 3:

Například funkci $f(x_1, x_2) = x_1 + x_2$ lze zapsat následujícím PL-programem:

```
{INPUT X1, X2}
Y := 0;
LOOP X1 INC Y; ENDLOOP
LOOP X2 INC Y; ENDLOOP
{OUTPUT Y}
```

Pravděpodobně se Vám zdá, že tento způsob zápisu má podobné prvky jako rekurzivní funkce. Zamyslíte-li se nad jednotlivými elementy, zjistíte například že:

- Přiřazovací příkaz je v podstatě identická nula
- Příkaz inkrementace je vlastně funkcí následníka
- Příkaz cyklu může beze zbytku nahradit rekurzi

Také lze ukázat, že bez příkazu GOTO budou PL-programy mít stejnou výpočetní sílu jako PRF, jinak jsou ekvivalentní ORF. Dále lze jednoduše simulovat libovolný PL-program pomocí Turingova stroje. Vlastně bychom jen na pásce TS vyhradili místo pro hodnoty proměnných a postupně naprogramovali v TS všechny příkazy (viz ukázka TS – následník).

Všechny tyto vlastnosti nejsou jen matematickou teorií, která zastřešuje pojmy a metody, které využíváme v informatice. Studium těchto formalizací vám umožní pochopit, že způsobů zápisu algoritmu je mnoho a i přes jejich různorodost mohou mít stejnou vyjadřovací/výpočetní sílu. Je to podobné jako, když jeden programátor rád píše své programy v jazyce Pascal a jiný v jazyce C. I když techniky v jednotlivých jazycích se mohou lišit, oba programátoři mohou vytvořit plnohodnotné programy, které se budou chovat identicky. Také díky objevování těchto teoretických otázek můžeme hlouběji pochopit, proč rekurze a cyklus jsou dva navzájem zaměnitelné nástroje algoritmického řešení problémů. Důležité je, že tyto notace jsou

poměrně pochopitelné a lze si k nim vytvářet mnoho jednoduchých i složitějších příkladů, provádět převody mezi jednotlivými způsoby formalizace a tím vytvářet lepší úroveň „informatického myšlení“. Pod tímto pojmem si představujeme schopnost navrhovat efektivní algoritmické řešení problémů, což je uvažování, které by mělo být specifickým rysem každého informatika.

3. Složitost řešení problému

Když už mluvíme o efektivním řešení problémů, musíme se alespoň krátce zmínit o druhé stránce oboru, kterým se zabývá tento článek. Je jí takzvaná složitost, čímž můžeme chápat především dvě odlišné vlastnosti algoritmů – buď časovou složitost nebo prostorovou složitost. Představte si, že několik studentů dostane za úkol samostatně vyřešit jistý algoritmický úkol. Budeme-li předpokládat, že všichni jej vyřeší správně a sestaví funkční algoritmy (programy) je pravděpodobné, že každý navrhne trochu jiný způsob řešení. U těchto programů pak lze položit otázku, který je vlastně nejlepší. A právě na tuto otázku lze (mimo jiné) nahlížet ze dvou hledisek:

1. Který program dá v průměru nejrychleji výsledek?
2. Který program spotřebuje ke svému správnému chodu nejméně paměti (prostoru)?

Jistě je odpověď na tyto otázky důležitá. Vždyť aby nám vůbec výpočetní technika v našem životě byla užitečná, musí pracovat rozumně dlouho a na strojích s kapacitou, které máme v dané době k dispozici. Aby bylo možné zadefinovat exaktně tyto pojmy používá se opět exaktní pojem algoritmu. Pro naše účely zvolme Turingův stroj.

U něj můžeme rozlišovat tyto základní charakteristiky:

1. **Složitost (časová) výpočtu TS** na určitý vstup – jde o počet vykonaných instrukcí TS na tento vstup (je to tedy prosté přirozené číslo); např. v příkladu 1 v tomto textu byl uveden výpočet Turingova stroje na slovo 1101, počet vykonaných instrukcí než se stroj zastavil je 8.
2. **Složitost (časová) TS (obecně)** – zde jde již o funkci, kde argument je velikost slova a funkční hodnota je počet instrukcí na slovo o této velikosti v nejhorším možném případě (je jasné, že různých slov o určité velikosti je mnoho – viz příklad 1); pro náš příklad 1 bychom opět mohli přemýšlet, jaká funkce to je. Stroj funguje tak, že vždy dojede na konec slova (což je n instrukcí), dále v nejhorším případě projde celé binární číslo (slovo) pozpátku až na začátek před první symbol a skončí v koncovém stavu($n + 2$ kroků). Složitost toho TS je tedy

$f(n) = n + n + 2 = 2n + 2$

3. **Odhad (časové) složitosti TS** je zjednodušením funkce složitosti TS. Většinou nás totiž nezajímá přesná funkce jako ve výše uvedeném případě, ale stačí nám jen ta nejdůležitější část funkce (kterou označíme $O(f)$), která roste nejrychleji, vzhledem k velikosti vstupního slova. Většinou se za tyto funkce berou $f(n) = n$, $f(n) = n^2$, ... atd.
   Tedy u našeho příkladu by odhad byl $O(n)$.
4. **Třída (časové) složitosti** je množina těch problémů, které jsou vyčíslovány nějakým TS, se složitostí $O(f)$. Většinou se obecně vymezují třídy lineární složitosti ($T(n)$), logaritmicko-lineární ($T(n \cdot log(n))$), kvadratické ($T(n^2)$), kubické ($T(n^3)$), ..., polynomiální ($T(n^k)$) (PTIME), exponenciální ($T(2^n)$) (EXPTIME). Například do třídy ($T(n)$) patří náš problém inkrementace binárního čísla.

Druhé hledisko, tedy prostorová složitost, může být definována v podstatě totožně, pokud složitost výpočtu TS deklarujeme jako počet symbolů pásky, které během výpočtu projde TS než se zastaví. Třídy prostorové složitosti se pak označují podobně s příponou SPACE- PSPACE, EXPSPACE. Třídy časové složitosti a prostorové složitosti pak můžeme uspořádat a intuitivně dokázat některé vlastnosti. Například je jasné, že když nějaký problém patří do určité třídy časové složitosti, pak patří do příslušné třídy prostorové složitosti (např. PTIME $\Rightarrow$ PSPACE). Proč tomu tak je? Vysvětlení je jednoduché. Turingův stroj nemůže za „polynomiálně mnoho“ času projít více než „polynomiálně mnoho“ symbolů, neboť se nemůže při jedné instrukci s hlavou posunout o více než jeden symbol.

V informatice je na časové složitosti založen jeden významný problém, který je jen zdánlivě pouze teoretický. Ve skutečnosti je však více „ze života“ než si uvědomujeme. Jde o takzvaný **P-NP PROBLÉM**. TS má podobně jako konečný automat také svou nedeterministickou verzi. Nedeterminismus spočívá ve více možnostech při přechodu z určité kombinace – stav, symbol, což je situace běžnému algoritmickému myšlení cizí, nicméně z hlediska složitosti umožňuje příslušnost do efektivnější třídy nedeterministické složitosti. Vezměme si příklad problému obchodního cestujícího. Spočívá v hledání nejefektivnější (nejkratší) cesty mezi několika městy, přičemž chceme navštívit všechna města právě jednou a vrátit se do výchozího. Jediné „klasické deterministické“ řešení, které je možné, spočívá v zásadě vždy ve zkoušení všech možných posloupností (existují i vylepšení, která částečně zlepšují složitost, nikoliv ale v principu). Tedy takové algoritmy patří do třídy (EXPTIME). Nicméně s pomocí nedeterministic-

kého TS můžeme tento problém řešit ve třídě NPTIME (nedeterministická polynomiální složitost). Teoretický trik (byť v praxi prozatím nic nepřinášející) je v tom, že takový nedeterministický stroj by prostě mohl z každého města zvolit jakoukoliv možnost a jelikož nás nezajímá, jak dokáže TS tuto nejlepší možnost najít, je jeho složitost lineární (tu správnou možnost TS prostě uhodne).

Proč je to pro praxi důležité? Kdyby se podařilo dokázat, že třída PTIME = NPTIME (jinak řečeno by pro každý problém, který lze řešit nedeterministicky, existoval i deterministický polynomiální algoritmus), pak bychom našli polynomiální deterministický algoritmus pro řešení optimalizačních problémů. Tyto problémy se v každodenním životě řeší často (kupříkladu sestavení rozvrhu hodinu na škole). Jelikož je však umíme řešit pouze v exponenciální čase, jsou při velkém rozsahu nezvládnutelné. Sestavíme jinak řečeno algoritmus, který pro malý počet vstupních prvků (třeba měst u problému obchodního cestujícího) funguje, ale existuje jistá mez, kdy algoritmus bude trvat neúnosně dlouho a nepomůže nám ani zvýšení výkonu stroje, který algoritmus realizuje. Na vině je totiž ona exponenciální funkce složitosti, kdy růst této funkce je vždy od určité hodnoty příliš strmý.

4 Závěr

Pokusili jsme se na poměrně malém prostoru ukázat na základní problémy teorie vyčíslitelnosti a složitosti, a to s ohledem na jejich význam pro vzdělávání informatiků a to nejen na úrovni vysokého školství. Problémy, které zkoumá tato teorie, sice vypadají odtažitě, složitě a značně formalizované, ale jejich význam pro schopnost algoritmicky myslet, uvědomovat si řešitelnost problémů a efektivitu řešení je velký.

Literatura

[1] *Kučera, M.:* Kombinatorické algoritmy, Matematický seminář SNTL 1989, Praha.

[2] *Hopcroft, J. E. – Ullman, J. D.:* Introduction to Automata theory, Languages and Computation. Addison-Wesley, Reading (Mass.), 1979.

[3] *Pavliska, V.:* Vyčíslitelnost a složitost I. Ostravská univerzita, Ostrava, 2003

[4] *Chytil, M.:* Automaty a gramatiky. Matematický seminář SNTL Praha, 1984.

[5] *Jančar, P.:* Vyčíslitelnost a složitost. VŠB TU Ostrava, http://www.cs.vsb.cz/jancar/ (2004).

[6] *Koubková, A. – Pavelka, J.:* Úvod do teoretické informatiky. Matfyz press, Praha 1998.