Vybrané úlohy MO kategorie P ročníků 41-50

In: Leo Boček (editor); Karel Horák (editor): [dokumenty-11] Padesát let matematické olympiády. 1951-2001. (Czech). Praha: Matfyzpress, 2001. pp. 69–86.

Persistent URL: http://dml.cz/dmlcz/405392

Vybrané úlohy MO kategorie P ročníků 41–50

42 – P – II – 1

Nákladní auta. Z města A vede silnice do města B. Na této silnici leží města $A_1, \ldots, A_n$, $B_1, \ldots, B_n$ — ne nutně v tomto pořadí, ale pro každé i $(1 \leqq i \leqq n)$ je vždy město A_i blíže k A než město B_i. Nákladní auto jezdí mezi městy A a B a má za úkol vyřídit n objednávek; i-tá $(1 \leqq i \leqq n)$ objednávka spočívá v převezení nákladu z města A_i do města B_i. Auto přitom může vézt vždy nejvýše jeden náklad a po naložení ve městě A_i ho může vyložit jedině v B_i.

Jízdou nákladního auta rozumíme jednu cestu auta z města A do města B a zpět do města A s případným splněním některých objednávek během této jízdy. Směr jízdy auta lze změnit jen ve městě B.

Napište a zdůvodněte program, který pro zadané vzdálenosti měst A_i a B_i od města A sestaví plán nákladního auta (tj. určí počet jízd a splněné objednávky pro každou jízdu) tak, aby auto vyřídilo všechny zadané objednávky minimálním počtem jízd.

Řešení. Označme si $a_i = |AA_i|$ a $b_i = |AB_i|$. Dle zadání $0 \leqq a_i < b_i$ pro každé i. V následujícím textu budeme jako objednávku j označovat tu objednávku, která patří městům A_j a B_j. Myšlenka algoritmu je jednoduchá — dokud existují nějaké nesplněné objednávky, přidáme vždy další jízdu a simulujeme cestu auta z A do B. Pokud je auto v nějakém městě volné, vybereme si jako další objednávku z dosud nesplněných tu, která začíná co nejdříve.

Přímá implementace této myšlenky vede k příliš složitému programu. Proto nebudeme přiřazovat objednávky jízdám, ale naopak. Čísla $a_1, \ldots, a_n$ a $b_1, \ldots, b_n$ uspořádáme vzestupně a poté procházíme úsek od A do B. Přitom si udržujeme zatím potřebný počet jízd v proměnné p a seznam „volných jízd“ (na začátku prázdný). Jestliže narazíme na město A_i a seznam je prázdný, zvýšíme počet jízd na $p + 1$ a i-tou objednávku splníme v této jízdě. Jestliže narazíme na město Bi, jízdu, ve které jsme splnili i-tou objednávku, „uvolníme“, tj. přidáme do seznamu volných jízd.

Poznámky k implementaci:

- Při třídění si ke každému prvku pamatujeme jeho původní index;
- seznam volných jízd implementujeme např. jako zásobník;
- je-li $b_i = a_k$, nejprve zpracováváme b_i.

Důkaz správnosti algoritmu:

Algoritmus zřejmě nalezne nějaký korektní plán, neboť každé objednávce přiřadí právě jednu jízdu a jízda, která je objednávce přiřazena, je v dané chvíli volná. Nyní dokážeme optimalitu získaného plánu. Označme $k_0 = \max |\{i \colon x \in (a_i, b_i\rangle\}|$ pro $x \in \langle 0, |AB|\rangle$ (maximální počet intervalů s neprázdným průnikem). Zřejmě potřebujeme alespoň k_0 jízd. Ukážeme, že algoritmus nalezne plán s přesně k_0 jízdami. Uvažujme takové i, že počet jízd (proměnná p) se zvyšuje z p na $p + 1$ při přidání bodu a_i. Tehdy není k dispozici žádná volná jízda, tedy bod a_i leží

v nějakých intervalech $\langle a_{kj}, b_{kj})$ pro $j = 1, \ldots, p$. Bod $a_i + e$, kde e je dostatečně malé číslo, leží uvnitř těchto p intervalů a navíc v intervalu $(a_i, b_i\rangle$, tedy $p+1 \leqq k_0$, takže hodnota p nikdy nevzroste nad k_0.

Odhad časové složitosti:

Odhlédneme-li od třídění, pro každý prvek a_i nebo b_i vykonáme konstantní množství operací, tedy celkem $O(n)$. Složitost třídění je $O(n \log n)$, dohromady tedy máme časovou složitost algoritmu $O(n \log n)$.

Paměťová složitost je lineární.

42 – P – III – 2

Nepostradatelné silnice. V zemi je N měst označených čísly od 1 do N. Mezi městy je vybudována silniční síť. Každá silnice spojuje vždy dvojici měst. Všechny silnice jsou obousměrné. Mezi některými dvojicemi měst přímá silnice nevede, ale z každého města je možné dojet po silnicích do libovolného jiného města (třeba i více různými způsoby). Všechna případná křížení silnic mimo města jsou mimoúrovňová a neumožňují vozidlům přejet z jedné silnice na druhou.

Silnici nazveme nepostradatelnou, pokud by se jejím zničením úplně přerušilo silniční spojení mezi některou dvojicí měst.

Napište program, který vyhledá a vypíše všechny nepostradatelné silnice. Vstupem programu je počet měst N a dále seznam všech silnic vedoucích mezi městy. Každá silnice je zadána dvojicí čísel měst, mezi nimiž vede.

Řešení. Úloha je jednou z klasických úloh teorie grafů. Silniční síť představuje souvislý neorientovaný graf, v němž vrcholy grafu odpovídají městům a hrany grafu silnicím. Nepostradatelné silnice, tak jak jsou definovány v zadání úlohy, odpovídají v teorii grafů zvláštním hranám zvaným mosty. Úkolem tedy je nalézt v daném grafu všechny mosty.

Algoritmus řešení je založen na procházení zadaným grafem do hloubky. Při procházení bude každý vrchol grafu navštíven právě jednou. Způsob procházení lze znázornit stromem. Kořenem stromu procházení je vrchol, z něhož bylo procházení zahájeno. Za kořen můžeme zvolit libovolný vrchol grafu. Bezprostředními následníky některého vrcholu V jsou všechny ty vrcholy, do nichž prohledávání z vrcholu V bezprostředně pokračovalo. Protože zadaný graf je souvislý, budou ve stromu procházení obsaženy všechny vrcholy původního grafu (města). Ze všech hran (silnic) budou ve stromě procházení obsaženy jen ty, které nás v průběhu procházení dovedly do nového, doposud nenavštíveného vrcholu.

Představme si, že do stromu procházení dokreslíme zelenou barvou všechny zbývající hrany grafu. To jsou tedy takové, kterými průchod do hloubky nepokračoval. Jinak řečeno, v průběhu procházení tyto silnice vedly z právě procházeného města do jiného, již dříve navštíveného města. Doplněný strom tedy bude izomorfní s původním grafem.

Nyní vyslovíme jedno pomocné tvrzení: Oba koncové vrcholy každé zelené hrany leží na téže větvi stromu procházení. Tvrzení snadno dokážeme sporem. Předpokládejme, že by některá zelená silnice spojovala dvě města A a B, která neleží na jedné větvi stromu procházení. Označme je tak, že během průchodu bylo A poprvé navštíveno dříve než B. Město B jistě neleží v podstromu procházení s kořenem A (jinak by A a B ležela na téže větvi). Pro postup procházení to znamená, že nejprve bylo (případně několikrát) navštíveno město A a teprve potom (případně několikrát) navštíveno město B. Všimněme si okamžiku během procházení, kdy jsme město A navštívili naposledy. To bylo v situaci, kdy jsme se z něho vraceli zpět (kdybychom šli vpřed, museli bychom do A přijít ještě na zpáteční cestě). V tomto okamžiku jsme se ale nezachovali správně podle algoritmu procházení grafem do hloubky: vraceli jsme se zpět, a přitom jsme ještě měli projít silnicí AB, protože ta vedla do tehdy ještě nenavštíveného města B.

K tomu, aby hrana byla mostem, je nutné a stačí, aby se jejím odstraněním oddělil podstrom ve stromu procházení, který nebude dostupný ani po některé zelené hraně. Podle předchozího tvrzení by takové spojení zelenou hranou muselo vést do vyšší vrstvy ve stromě procházení.

Na základě provedených úvah již lze zformulovat algoritmus. Zadaný graf budeme procházet do hloubky, začít můžeme libovolným vrcholem (např. vrcholem číslo 1). Během procházení si budeme u každého vrcholu M pamatovat jeho hloubku ve stromě procházení H_M. Kořen stromu procházení bude mít hloubku 0. Postupně během průchodu budeme pro každý procházený vrchol M určovat číslo Z_M definované takto: Z_M je minimum z H_M a z hloubek koncových měst všech zelených silnic, které vycházejí z vrcholů v podstromu s kořenem M. Z_M je tedy číslo nejvyšší hladiny ve stromě procházení, do které vede přímé spojení zelenou silnicí z nějakého města v podstromu s kořenem M. Přitom si všímáme jen hladin nad vrcholem M. Nastane-li pro některý vrchol M nerovnost $Z_M < H_M$, existuje zelená silnice, která spojuje podstrom s kořenem ve vrcholu M se zbytkem grafu. Je-li $Z_M = H_M$, je silnice, po níž jsme do M během procházení přišli, mostem.

Zbývá ukázat, jak budeme počítat hodnoty H_M a Z_M pro vrchol M. Hodnotu H_M určíme snadno při prvním vstupu do vrcholu M během procházení grafem — je o 1 větší než odpovídající hodnota H_X vrcholu X, z něhož do M přicházíme. Stanovení hodnoty Z_M je o něco obtížnější. Hodnota Z_M je rovna minimu z hodnoty H_M a z hodnot Z_i všech vrcholů ležících v podstromu s kořenem ve vrcholu M. Při prvním vstupu do vrcholu M můžeme tudíž inicializovat hodnotu Z_M již známou hodnotou H_M a pak ji během procházení podstromu vrcholu M budeme případně zmenšovat, bude-li to možné. Při každém dalším příchodu do vrcholu M (tj. při návratu z nějakého následníka vrcholu M) lze hodnotu Z_M snížit na Z-hodnotu tohoto následníka. K dalšímu snížení Z_M mohou přispět hrany, které vedou z vrcholu M do již navštívených uzlů a po nichž se tudíž při průchodu nepostupuje. Hodnotu Z_M můžeme snížit na H-hodnotu koncových vrcholů těchto zelených hran. Definitivní hodnotu Z_M získáme až při posledním opuštění vrcholu M.

Složitost celého algoritmu je dána složitostí průchodu grafem do hloubky. Ostatní výpočty spojené s určováním hodnot H_M a Z_M mají konstantní časové nároky. Časová složitost programu pro průchod grafem do hloubky závisí na vhodné volbě vnitřní reprezentace grafu. Při vhodně zvolené reprezentaci (viz uvedená programová ukázka) je přímo úměrná počtu hran v grafu, tj. je řádu n^2, kde n je počet vrcholů grafu.

```
program Nepostradatelne_silnice;

{Formát očekávaných vstupních dat - zadání grafu:
 - sousedi každého vrcholu vždy na jednom řádku ve tvaru
     <číslo-vrcholu> <číslo-souseda1> <číslo-souseda2> ...
 - vrcholy musí být očíslovány od jedné po jedné a v tomto
   pořadí musí být také řádky na vstupu zadány
 - každá hrana se tedy uvádí dvakrát (na řádcích pro jeden
   a pro druhý její koncový vrchol)
 - program pro jednoduchost netestuje správnost zadaných
   vstupních dat (nebylo by těžké testy doplnit) }

const
  MaxPocetMest = 40;
  MaxPocetSilnic = 200;

type
  Mesto   = 1..MaxPocetMest+1;   {fiktivní město na konci}
  Silnice = 1..MaxPocetSilnic;

var
  GMesto : array [Mesto] of record
                              Spoje   : Silnice;
                              Hloubka : integer;
                              Projito : Boolean;
                            end;
  GSilnice : array [Silnice] of Mesto;
   {pole GMesto a GSilnice představují vnitřní uložení grafu
   - ke každému vrcholu je v poli GSilnice uložen seznam
     jeho sousedů, položka Spoje v GMesto určuje, kde přesně
     jsou uloženi sousedi každého konkrétního vrcholu
   - viz úvodní komentář o tvaru vstupních dat a procedura
     NactiGraf}

  PocetMest   : 0..MaxPocetMest;
  PocetSilnic : 0..MaxPocetSilnic;
  F:integer;    {pomocná proměnná - je potřebná pro správné
                 volání procedury Pruchod v hlavním programu}

procedure NactiGraf;
{načtení vstupních dat - zadání zkoumaného grafu}
```

```
var dummy:integer;
begin
  PocetMest:=0;
  PocetSilnic:=1;
  repeat
    PocetMest:=PocetMest+1;
    read(dummy);                {číslo města - nevyužívá se}
    GMesto[PocetMest].Spoje:=PocetSilnic;
    while not eoln do
      begin
        read(GSilnice[PocetSilnic]);
        PocetSilnic:=PocetSilnic+1;
      end;
    readln;
  until eof;
  GMesto[PocetMest+1].Spoje:=PocetSilnic;
end;

function min(X,Y:integer):integer;
{pomocná procedura - minimum ze dvou celých čísel}
begin
  if X<Y then min:=X else min:=Y;
end;

procedure PisMost(odkud,kam : Mesto);
{vypíše, že z města "odkud" do města "kam" vede most}
begin
  writeln('Most z ',odkud,' do ',kam,'.');
end;

procedure PredPruchodem;
{označí všechna města za dosud neprojitá - nutné!}
var i:Mesto;
begin
  for i:=1 to PocetMest do GMesto[i].Projito := false;
end;

procedure Pruchod(Start: Mesto; hl:integer; var Z: integer);
{Projde odpovídající část grafu do hloubky.
 Začíná ve městě Start, jeho hloubka je hl, spočítá pro něj
 hodnotu Z (viz text).}
var Nasl : Mesto;
    S : Silnice;
    pomZ : integer;
begin
  GMesto[Start].Projito := true; {vrchol navštíven}
  GMesto[Start].Hloubka := hl; {má hloubku hl}
  Z := hl; {prozatímní hodnota Z}
```

```
  for S:=GMesto[Start].Spoje to GMesto[Start+1].Spoje-1 do
    begin
      Nasl := GSilnice[S]; {město dostupné po silnici S}
      if GMesto[Nasl].Projito then
        begin {nepočítat silnici, po níž jsme přišli!}
          if GMesto[Nasl].Hloubka+1 <> hl then
                  {zelená silnice, buď nahoru, nebo dolů}
            Z := min(Z,GMesto[Nasl].Hloubka)
                  {pokud vede zelená silnice nahoru,
                   snížíme hodnotu Z v našem uzlu}
    end
  else
    begin
      Pruchod(Nasl,hl+1,pomZ);
      if hl+1 = pomZ then
        PisMost(Start,Nasl);          {nalezen most v grafu}
      Z := min(Z,pomZ);          {případné snížení hodnoty Z}
    end;
  end;
end;

begin
  NactiGraf;
  PredPruchodem;
  Pruchod(1,0,F);
  {vždy je F=0}
end.
```

47 – P – III – 2

Kódy. Při přenosu dat (posloupností bitů) po linkách mezi počítači může někdy dojít k chybě: čas od času se stane, že se místo některého vyslaného bitu přijme na konci linky bit jiný. Nedochází ovšem k tomu, že by se při přenosu některý bit zcela ztratil nebo že by naopak bit přibyl.

Pro přenos dat po nespolehlivých vedeních se proto často používají různé zabezpečovací kódy. K přenášeným bitům $b_1 \ldots b_n$ se přidají navíc kontrolní bity $c_1 \ldots c_m$ tak, aby podle nich bylo možné na druhé straně linky zjistit, zda se zpráva přenesla v pořádku.

Nejjednodušším možným zabezpečovacím kódem je paritní kód doplňující jediný bit c_1 takový, aby celkový počet jedničkových bitů v celé zprávě byl sudý. Tento kód umožňuje spolehlivě odhalit nejvýše jeden chybný bit v celé zprávě (ať již to byl jeden z datových bitů b_i nebo přidaný paritní bit). Neumožňuje však přesně zjistit, který bit je chybný (což by se rovnalo opravení tohoto bitu, jelikož jsou pouze dvě možnosti, jakou hodnotu může mít).

a) Navrhněte zabezpečovací kód, který k n-bitové zprávě přidá m zabezpečovacích bitů tak, aby případnou chybu v jednom bitu zprávy bylo možné nejen zjistit, ale také opravit.

b) Navrhněte zabezpečovací kód, který k n-bitové zprávě přidá m zabezpečovacích bitů tak, aby bylo možné ověřit správnost došlé zprávy za předpokladu, že při jejím přenosu dojde k chybě v nejvýše dvou bitech (včetně bitů zabezpečovacích).

V obou případech se snažte o co nejmenší prodloužení zprávy (tzn. minimalizujte počet přidaných bitů m) a dokažte, že navržený kód skutečně požadované vlastnosti má.

Řešení. Následující kód řeší obě podúlohy (nikoliv však současně — není schopen rozhodnout, jestli došlo k opravitelné jednochybě nebo neopravitelné dvojchybě).

Nechť zpráva má $n = 2^k - 1$ bitů označených $b_1 \ldots b_n$ (v případě, že je kratší, považujeme zbylé bity za nuly). My ji doplníme o paritní bit p (stejně jako v ukázkovém příkladu v zadání) a zabezpečovacími bity c_0 až c_{k-1}, jež budou paritními bity jednotlivých bloků zvolených podle následujícího pravidla: v i-tém bloku budou právě ty bity, jejichž pořadové číslo (pro bit b_z to je z) má v i-tém binárním řádu jedničku (každý bit původní zprávy tak patří do alespoň jednoho bloku a je jednoznačně identifikován tím, do kterých bloků patří a do kterých nikoliv).

a) Po příjmu zprávy zkusíme podle přijatých datových bitů znovu spočíst všechny bity kontrolní (tedy p a všechna c_i). Pokud souhlasí s těmi, které byly přijaty, zpráva byla přijata bezchybně nebo došlo k více jak jedné chybě. Pokud došlo k právě jedné chybě, mohlo k ní dojít těmito způsoby:

- V jednom datovém bitu b_i: v tomto případě zaručeně nesouhlasí paritní bit p a rovněž některé z blokových paritních bitů c_j — konkrétně ty, jež odpovídají blokům, do nichž bit b_i patří. Jenže tím, ve kterých blocích se b_i nachází, je již jeho pořadové číslo i jednoznačně určeno.
- V jednom kontrolním bitu c_i: v tomto případě souhlasí centrální paritní bit p, čímž tento případ snadno odlišíme.
- V centrálním paritním bitu p: v tomto případě nesouhlasí tento bit, ale souhlasí všechny ostatní kontrolní bity, což znamená, že chyba se nenachází v žádném bloku, ale to u chyb v datových bitech nastat nemůže.

b) Nedošlo-li k žádné chybě, souhlasí všechny zabezpečovací bity s jejich vypočtenými hodnotami (viz řešení úlohy a). Došlo-li k právě jedné chybě, nesouhlasí centrální paritní bit nebo jeden z blokových paritních bitů (viz diskuse výše). Došlo-li k právě dvěma chybám, mohou nastat následující případy:

- Chyba ve dvou různých datových bitech: centrální paritní bit jistě souhlasí, ale jelikož každý datový bit je jednoznačně popsán kombinací bloků, do nichž patří, zaručeně existuje alespoň jeden blok, do kterého patří právě jeden z chybně přijatých bitů, a proto jeho blokový paritní bit nesouhlasí.

- Chyba ve dvou různých blokových paritních bitech: centrální paritní bit souhlasí, ale oba chybně přijaté blokové paritní bity nikoliv.
- Chyba v jednom datovém bitu a jednom blokovém paritním bitu: nesouhlasí centrální paritní bit.
- Chyba v jednom datovém bitu a centrálním paritním bitu: centrální paritní bit souhlasí, ale alespoň jeden z blokových nikoliv (každý datový bit je v alespoň jednom bloku).
- Chyba v centrálním paritním bitu a jednom z blokových: nesouhlasí centrální paritní bit.

Ve všech případech tedy poznáme, že k chybě došlo.

48 – P – I – 4

Normální algoritmy Markova. Konečnou množinu symbolů $T = \{a_0, a_1, \ldots, a_n\}$ nazveme **abecedou**. Prvky množiny T budeme nazývat **znaky** abecedy. Slovem P v abecedě T nazveme každou konečnou posloupnost znaků abecedy T. **Prázdné slovo** budeme označovat symbolem Λ. **Algoritmem** v abecedě T budeme rozumět funkci, jejíž definičním oborem je podmnožina slov v abecedě T a oborem hodnot je opět podmnožina slov v T. Nechť P je slovo v abecedě T. Řekneme, že algoritmus $\mathcal{A}$ je **přípustný** pro slovo P, právě když P je prvkem jeho definičního oboru. Většinu algoritmů je možno rozdělit na nějaké jednodušší kroky. Jeden ze způsobů navrhl v roce 1954 A. A. Markov. Základní operací je substituce jednoho slova za jiné. Výrazy $P \to Q$ a $P \to \cdot Q$, kde P a Q jsou slova v abecedě T, nazýváme **formulemi substituce** v abecedě T. Přitom předpokládáme, že šipka ($\to$) a tečka ($\cdot$) nejsou prvky T. Některé ze slov P a Q může být i prázdné. Formuli $P \to Q$ nazýváme **obyčejnou** a formuli $P \to \cdot Q$ nazýváme **koncovou**. Zápisem $P \to (\cdot)Q$ rozumíme buď formuli $P \to Q$, nebo $P \to \cdot Q$. Konečný seznam formulí substituce v abecedě T

$$\begin{cases} P_1 & \to (\cdot)Q_1 \\ P_2 & \to (\cdot)Q_2 \\ \vdots & \\ P_r & \to (\cdot)Q_r \end{cases}$$

nazýváme **schéma algoritmu** $\mathcal{A}$.

Řekneme, že **slovo** W **je obsaženo ve slově** Q, právě když existují taková slova U, V (mohou být i prázdná), že $Q = UWV$. Algoritmus $\mathcal{A}$ pracuje následujícím způsobem. Nechť je dáno slovo P v abecedě T. Ve schématu algoritmu $\mathcal{A}$ najdeme první formuli $P_m \to (\cdot)Q_m$ $(1 \leqq m \leqq r)$ takovou, že P_m je obsaženo v P. Provedeme substituci nejlevějšího výskytu slova P_m na Q_m. Označme R_1 slovo, které je výsledkem této substituce. Můžeme napsat $\mathcal{A}\colon P \vdash R_1$. Je-li $P_m \to \cdot Q_m$ koncová formule substituce, činnost algoritmu $\mathcal{A}$ končí, jeho **hodnotou** je slovo R_1 a píšeme $\mathcal{A}(P) = R_1$. Je-li $P_m \to Q_m$ obyčejná formule substituce, aplikujeme na

R_1 stejný postup, který jsme použili na slovo P. Tímto způsobem pokračujeme dále. Dostaneme posloupnost slov $P, R_1, \ldots, R_i$ $(i \geqq 1)$. Můžeme psát

$$\mathcal{A}\colon P \vdash R_1, \ldots, \vdash R_i \quad \text{nebo též zkráceně} \quad \mathcal{A}\colon P \vdash^* R_i.$$

Jestliže v určitém okamžiku nastane situace, že ani jedno ze slov $P_1, \ldots, P_r$ není obsaženo v R_i, potom činnost algoritmu rovněž končí a R_i je hodnotou algoritmu $\mathcal{A}$. Může se ovšem stát, že popsaný postup nikdy nekončí. V takovém případě řekneme, že algoritmus **není přípustný** pro slovo P.

Příklad 1: Nechť $T = \{b, c\}$. Uvažujme schéma

$$\begin{cases} b \to \cdot\Lambda \\ c \to c \end{cases}$$

normálního algoritmu $\mathcal{A}$ pro slovo P v abecedě T.

Výsledek činnosti algoritmu $\mathcal{A}$ je pro P následující:

- Je-li P prázdné slovo, je hodnota algoritmu rovněž prázdné slovo ($\mathcal{A}(\Lambda) = \Lambda$).
- Obsahuje-li P aspoň jeden znak b, potom hodnotou algoritmu je slovo, které vznikne z P vynecháním nejlevějšího výskytu znaku b v P (např. $\mathcal{A}(cbcb) = ccb$).
- Algoritmus není přípustný pro slova neobsahující znaky b.

Příklad 2: Nechť $T = \{a_0, a_1, \ldots, a_n\}$. Uvažujme schéma

$$\begin{cases} a_0 \to \Lambda \\ a_1 \to \Lambda \\ \vdots \\ a_n \to \Lambda \end{cases}$$

Takové schéma zapisujeme zkráceně ve tvaru

$$\{ \quad \xi \to \Lambda \quad (\xi \in T).$$

Při tomto zkráceném zápisu schématu předpokládáme, že odpovídající prvky seznamu mohou být zapsány v libovolném pořadí. Výsledkem algoritmu $\mathcal{A}$ je vždy prázdné slovo. Říkáme též, že $\mathcal{A}$ přepisuje libovolné slovo (v abecedě T) na prázdné slovo. Činnost tohoto algoritmu můžeme zapsat například ve tvaru $\mathcal{A}$: $a_1a_3a_0 \vdash a_1a_3 \vdash a_3 \vdash \Lambda$ nebo $\mathcal{A}\colon a_1a_3a_0 \vdash^* \Lambda$, takže $\mathcal{A}(a_1a_3a_0) = \Lambda$.

Příklad 3: Nechť $T = \{1\}$. Definujme induktivně $\bar{0} = 1$ a $\overline{n+1} = \bar{n}1$, tj. $\bar{1} = 11$, $\bar{2} = 111$ atd. Slova $\bar{n}$ nazveme čísla.

Uvažujme schéma

$$\Lambda \to \cdot 1.$$

Pro libovolné slovo P v T platí zřejmě $\mathcal{A}(P) = 1P$, což můžeme zapsat ve tvaru $\mathcal{A}(\overline{n}) = \overline{n+1}$ pro libovolné přirozené číslo n. (Uvědomte si, že každé slovo P začíná prázdným slovem Λ, tj. $P = \Lambda P$).

Abecedu B nazveme **rozšířením** abecedy T, platí-li $T \subseteq B$. V řadě případů je nutno při konstrukci schématu algoritmu v abecedě T použít mimo znaků abecedy T ještě další pomocné znaky. Pak řekneme, že jsme vytvořili schéma algoritmů **nad abecedou** T (tj. v nějaké abecedě B, která je rozšířením T).

Příklad 4: Nechť $T = \{a_0, a_1, \ldots, a_n\}$ je abeceda. Pro každé slovo $P = a_{j_0}a_{j_1}\cdots a_{j_k}$ v T $(k \geqq 0,\ a_{j_i} \in T,\ i = 0,1,\ldots,n)$ je slovo $\breve{P} = a_{j_k}\cdots a_{j_1}a_{j_0}$ **obrácením** slova P. Sestrojte normální algoritmus $\mathcal{A}$, který pro libovolné slovo P vytvoří jeho obrácení, tj. $\mathcal{A}(P) = \breve{P}$.

Uvažujme zkrácené schéma algoritmu v abecedě $B = T \cup \{\alpha, \beta\}$:

$$\left\{\begin{array}{rll} \alpha\alpha & \to \beta & \text{(a)} \\ \beta\xi & \to \xi\beta \quad (\xi \in T) & \text{(b)} \\ \beta\alpha & \to \beta & \text{(c)} \\ \beta & \to \cdot\Lambda & \text{(d)} \\ \alpha\nu\xi & \to \xi\alpha\nu \quad (\xi, \nu \in T) & \text{(e)} \\ \Lambda & \to \alpha & \text{(f)} \end{array}\right.$$

- Na základě formule (f) dostaneme $\mathcal{A}$: $P \vdash \alpha P$.
- Potom je aplikována v potřebném počtu opakování formule (e): $\mathcal{A}$: $P \vdash$ $\vdash a_{j_1}\alpha a_{j_0}a_{j_2}\cdots a_{j_k} \vdash a_{j_1}a_{j_2}\alpha a_{j_0}a_{j_3}\cdots a_{j_k} \vdash^* a_{j_1}a_{j_2}\cdots a_{j_k}\alpha a_{j_0}$.
- S opakováním předchozích dvou kroků postupně dostaneme

$$\mathcal{A}\colon P \vdash^* a_{j_2}a_{j_3}\cdots a_{j_k}\alpha a_{j_1}\alpha a_{j_0} \vdash^* \alpha a_{j_k}\alpha a_{j_{k-1}}\cdots\alpha a_{j_1}\alpha a_{j_0} \vdash$$
$$\vdash \alpha\alpha a_{j_k}\alpha a_{j_{k-1}}\cdots\alpha a_{j_1}\alpha a_{j_0}.$$

- S pomocí (a) dále dostaneme $\mathcal{A}$: $P \vdash^* \beta a_{j_k}\alpha a_{j_{k-1}}\cdots\alpha a_{j_1}\alpha a_{j_0}$, pomocí (b) a (c) a s posledním použitím (d) dostaneme $\mathcal{A}(P) = \breve{P}$.

Tím jsme popsali činnost normálního algoritmu $\mathcal{A}$ nad abecedou T obracející slovo v abecedě T.

Soutěžní úlohy:

1. Nechť $H = \{1\}$, $M = \{1, *\}$. Každé přirozené číslo n může být zapsáno jako $\bar{n}$, což je slovo v abecedě H (viz př. 3 ve studijním textu). Napište schéma algoritmu $\mathcal{A}$ v M, který je přípustný pouze pro slova, jež jsou zápisem přirozeného čísla. Hodnotou algoritmu $\mathcal{A}$ pro libovolné $\bar{n}$ bude $\mathcal{A}(\bar{n}) = \bar{0}$. Zdůvodněte správnost navrženého algoritmu.
2. Je dána abeceda T, která neobsahuje znaky α, β, γ, $B = T \cup \{\alpha, \beta, \gamma\}$. Sestrojte schéma algoritmu $\mathcal{A}$ v abecedě B, který každé slovo v abecedě T zdvojí (tj. $\mathcal{A}(P) = PP$). Zdůvodněte jeho správnost.

Řešení. a) Výskyt znaku $*$ ve vstupním slově musí způsobit, že Markovův algoritmus nebude pro toto slovo přípustný. Toho dosáhneme tak, že prvním pravidlem schématu budeme hvězdičku stále přepisovat samu na sebe. Jinak se má jakýkoliv počet jedniček ve vstupním slově zredukovat na jediný znak 1 představující číslo nula. To zajistí formule (2). Poslední jednička ve slově zůstane a výpočet se ukončí podle formule (3). Zbývá ošetřit případ, že vstupní slovo je prázdné. V takovém případě slovo neobsahuje zápis čísla, proto pomocí pravidla (4) vložíme do slova hvězdičku, což v následujícím kroku vede opět k nekonečnému výpočtu podle první formule.

$$\begin{cases} * & \to * & (1) \\ 11 & \to 1 & (2) \\ 1 & \to \cdot 1 & (3) \\ \Lambda & \to * & (4) \end{cases}$$

Rychlost výpočtu podle tohoto schématu Markovova algoritmu je lineární, počet kroků výpočtu je roven nejvýše délce vstupního slova.

b) Úlohu lze řešit s použitím pouze dvou pomocných symbolů mimo abecedu T. Pomocí formule (5) se nejprve umístí na začátek vstupního slova symbol α. Pomocí formule (1) pak tímto symbolem procházíme slovo zleva doprava a za sebou zdvojujeme jednotlivé znaky slova. V každém okamžiku je umístěno před znakem, který má být v příštím kroku zdvojen, zatímco každý nově přidaný „dvojník" je označen symbolem β (umístěn bezprostředně před ním). Formule (1) se použije přesně tolikrát, kolik znaků má vstupní slovo. Po jejím posledním použití se dostane symbol α až na konec slova. Dále se používá opakovaně formule (2). Ta zajišťuje přesunutí přidaných dvojníků i se svými symboly β na pravý konec slova, nepřipouští však změnu vzájemného pořadí dvojníků. V okamžiku, kdy již nelze pravidlo (2) použít, jsme v podstatě hotovi — původní vstupní slovo je zdvojené, zbývá už jen odstranit pomocné symboly. Před každým dvojníkem v pravé části slova se nachází jeden symbol β a na samém konci slova je jeden symbol α. Všechny pomocné symboly snadno odstraníme pravidly (3) a (4).

$$\begin{cases} \alpha\xi & \to \xi\beta\xi\alpha & (\xi \in T) & (1) \\ \beta\xi\eta & \to \eta\beta\xi & (\xi, \eta \in T) & (2) \\ \beta & \to \Lambda & & (3) \\ \alpha & \to \cdot\Lambda & & (4) \\ \Lambda & \to \alpha & & (5) \end{cases}$$

Zbývá stanovit rychlost výpočtu. Formule (4) a (5) se použijí každá jednou, formule (1) a (3) každá n-krát, kde n je počet znaků vstupního slova. Nejpracnější je přerovnání pořadí znaků ve slově pomocí formule (2). První dvojník se musí vyměnit postupně se všemi $n-1$ zbývajícími znaky, druhý již jen s $n-2$ znaky, atd. Celkový počet použití pravidla (2) při výpočtu je proto $(n-1)+(n-2)+(n-3)+\ldots+1 = \frac{1}{2}n(n-1)$. Algoritmus má tedy kvadratickou časovou složitost.

49 – P – II – 2

Posloupnost. Nechť $A = (a_1, a_2, \ldots, a_M)$ je posloupnost celých čísel. Posloupnost A' nazveme vybranou podposloupností této posloupnosti, jestliže vznikne z posloupnosti A vynecháním některých jejích členů, přičemž pořadí ostatních prvků zachováme. Společná vybraná podposloupnost posloupností A, B je každá taková posloupnost C, která je vybranou podposloupností obou posloupností A i B.

Příklad. Nechť $A = (1, 11, 2, 1, 4, 99)$, $B = (9, 4, 1, 2, 7, 1, 99)$. Posloupnost $(1, 2, 1, 99)$ je společnou vybranou podposloupností posloupností A, B. Posloupnost $(1, 2, 4)$ je vybranou podposloupností posloupnosti A, ale není vybranou podposloupností posloupnosti B, takže není ani společnou vybranou podposloupností A a B.

Soutěžní úloha. Máme dány dvě posloupnosti celých čísel $A = (a_1, a_2, \ldots, a_M)$ a $B = (b_1, b_2, \ldots, b_N)$ s délkami M, resp. N. Tyto posloupnosti jsou uloženy v polích $a[1 \ldots M]$, resp. $b[1 \ldots N]$, jejichž obsah není dovoleno měnit. Uvažujme takovou společnou vybranou podposloupnost posloupností A a B, ve které součet všech jejích členů je největší možný. Napište program, který vypíše součet členů takovéto posloupnosti.

Poznámka. Existuje algoritmus, který tuto úlohu řeší v čase úměrném $M \times N$ a paměti, jejíž velikost je úměrná menšímu z čísel M, N.

Příklad:

```
Vstup:                Výstup:
M=6, N=7              103
A=(1,11,2,1,4,99)
B=(9,4,1,2,7,1,99)
```

(Vybrané podposloupnosti se součtem 103 existují dvě: $4, 99$ a $1, 2, 1, 99$.)

Řešení. V řešení úlohy použijeme metodu dynamického programování. Označme $A_i = a[1], a[2], \ldots, a[i]$ posloupnost tvořenou prvními i členy posloupnosti a, analogicky $B_j = b[1], \ldots, b[j]$. Budeme nejprve řešit obecnější úlohu: Pro každé i, j ($0 \leqq i \leqq M$, $0 \leqq j \leqq N$) spočítáme, jaký je maximální součet členů společné vybrané podposloupnosti posloupností A_i a B_j. Tyto maximální součty si budeme zapisovat do tabulky $p[0 \ldots M, 0 \ldots N]$, kde $p[i, j]$ je součet maximální společné vybrané podposloupnosti posloupností A_i a B_j. Hledaným maximálním součtem bude tedy hodnota $p[M, N]$.

Tabulku p budeme vyplňovat po řádcích s využitím předpočítané informace uložené v předešlém řádku. Řádek $p[0]$ obsahuje samé nuly, protože neexistuje vybraná podposloupnost prázdné posloupnosti. Řádek $p[i]$ (pro $i > 0$) vyplníme podle řádku $p[i-1]$ takto: Políčko $p[i, 0]$ má zřejmě hodnotu nula. Políčko $p[i, j]$ (pro $j > 0$) umíme určit pomocí hodnot $p[i-1, j]$, $p[i, j-1]$ a $p[i-1, j-1]$. Jestliže se čísla $a[i]$ a $b[j]$ neshodují, každá vybraná podposloupnost posloupností A_i a B_j je zároveň vybranou podposloupností posloupností A_{i-1} a B_j nebo A_i

a B_{j-1}. Tedy v tomto případě je $p[i,j]$ rovno maximu z čísel $p[i-1,j]$ a $p[i,j-1]$. Jestliže $a[i] = b[j]$, každá vybraná podposloupnost posloupností A_i a B_j je vybranou podposloupností posloupností A_{i-1} a B_j nebo A_i a B_{j-1}, nebo je vybranou podposloupností posloupností A_{i-1} a B_{j-1} s přidaným členem $a[i] = b[j]$. Proto $p[i,j]$ je rovno maximu z čísel $p[i-1,j]$, $p[i,j-1]$ a $p[i-1,j-1]+a[i]$.

Navržený algoritmus má časovou složitost $O(MN)$. Paměťová složitost je také $O(MN)$. Jelikož každý řádek tabulky p závisí pouze na předcházejícím řádku, stačí si pamatovat jen poslední dva řádky (při počítaní řádku $p[i]$ si pamatujeme předcházející řádek $p[i-1]$; v programu p_1 označuje řádek $p[i-1]$ a p_2 řádek $p[i]$), paměťová složitost algoritmu je při této realizaci proto pouze $O(M+N)$.

```
program Posloupnost;
const MAX = 1000;
var a,b: array[1..MAX] of integer;
var p1,p2: array[0..MAX] of integer;
    M,N,i,j: integer;

Begin
  for i:=0 to N do p1[i]:=0;
  p2[0] := 0;
  for j:=1 to M do begin
    for i:=1 to N do begin
      if p2[i-1] > p1[i] then p2[i] := p2[i-1]
      else p2[i] := p1[i];
      if (b[i] = a[j]) and (p1[i-1]+a[j] > p2[i]) then
        p2[i] := p1[i-1]+a[j];
    end;
    for i:=1 to N do p1[i] := p2[i];
  end;
  Writeln('Největší součet podposloupnosti je: ',p1[N]);
End.
```

50 – P – I – 2

Posel. Na království krále Mírumila III. zaútočila nepřátelská vojska a podařilo se jim obsadit několik měst. Král nyní potřebuje dát svému generálovi příkaz k protiútoku (bez příkazu přeci generál nemůže bojovat). Generál však momentálně provádí inspekci vojsk v jiném městě. Je proto třeba vyslat posla, který příkaz co nejrychleji doručí. Příkaz ovšem v žádném případě nesmí padnout do rukou nepřítele! Proto se posel musí neustále držet co nejdále od nepřítelem obsazených měst. Vaším úkolem je navrhnout pro posla co nejlepší trasu.

Soutěžní úloha: Program dostane na vstupu zadaný počet měst N ($1 \leqq N \leqq 100$). Jednotlivá města budeme označovat čísly $1 \ldots N$. Dále je na vstupu uveden počet cest M ($1 \leqq M \leqq 10\,000$) a seznam těchto cest vedoucích mezi městy. Každá cesta

je určena dvojicí čísel měst, která spojuje. Cesty se kříží pouze ve městech a je možno se po nich dostat z libovolného města do libovolného (případně přes města jiná). Další údaj K zadaný na vstupu určuje počet měst obsazených nepřítelem, následuje seznam obsazených měst. Nakonec program dostane číslo města, odkud vyráží posel, a číslo města, kde se zdržuje generál. Váš program má nalézt trasu, jejíž vzdálenost od měst obsazených nepřítelem je maximální. Pokud existuje takových tras více, program určí libovolnou nejkratší z nich. Vzdálenost měst $\mathcal{A}$ a $\mathcal{B}$ počítáme jako minimální počet cest, po kterých musíme projít, abychom se dostali z města $\mathcal{A}$ do města $\mathcal{B}$. Vzdálenost trasy od města $\mathcal{A}$ je pak nejmenší ze vzdáleností města $\mathcal{A}$ od jednotlivých měst ležících na uvažované trase. Vzdáleností trasy od obsazených měst rozumíme nejmenší ze vzdáleností mezi trasou a některým z obsazených měst nebo nulu pokud některé město na trase samé je obsazeno.

Formát vstupu: První řádek vstupního souboru `posel.in` obsahuje čísla N (počet měst) a M (počet cest). Po něm následuje M řádků, z nichž každý obsahuje popis jedné cesty. Cesta je popsána dvojicí čísel koncových měst. Následuje řádek s číslem K (počet obsazených měst) a za ním K řádků s čísly obsazených měst. Poslední řádek vstupního souboru obsahuje číslo města, odkud vyjíždí posel, a číslo města, kde dlí generál.

Formát výstupu: Výstupem programu v souboru `posel.out` jsou čísla měst na nejlepší nalezené trase uvedená v pořadí, v jakém jimi má posel projíždět. Všechna čísla měst jsou zapsána na jediném řádku výstupního souboru a jsou oddělena mezerami.

Příklad:

`posel.in`

```
10 12
1 2
2 3
3 4
4 5
2 5
1 6
6 7
7 8
8 5
1 9
9 10
10 5
1
3
1 5
```

`posel.out`

```
1 9 10 5
```

Řešení. Algoritmus řešící tuto úlohu se dá rozdělit do tří fází. V první fázi se pro každé město spočítá, jaká je jeho vzdálenost od nepřítelem obsazených měst

(ve smyslu definice uvedené v zadání). Ve druhé fázi se zjistí, jakou maximální vzdálenost od nepřátelských měst dokážeme udržet při cestě z počátečního do cílového města. Ve třetí fázi pak nalezneme nejkratší z tras vedoucích z počátečního do cílového města, které udržují spočtenou vzdálenost.

První fáze: Vzdálenost od obsazených měst budeme hledat pomocí prohledávání do šířky. U každého města si budeme udržovat informaci, zda jsme v něm již byli (na počátku bude nastaveno právě u všech obsazených měst) a jeho vzdálenost od nepřítele. Pro města obsazená nepřítelem bude tato vzdálenost rovna 0. Dále si budeme udržovat frontu měst ke zpracování, do které na začátku uložíme všechna nepřátelská města. V každém kroku výpočtu vždy vezmeme jedno město z fronty a u všech jeho sousedů, ve kterých jsme dosud nebyli, nastavíme vzdálenost o jedna větší, než je vzdálenost vybraného města. U všech těchto sousedů také označíme, že jsme v nich už byli, a přidáme je na konec fronty. První fáze výpočtu končí, když se vyprázdní fronta. Tehdy jsme prošli všechna města a určili jsme vzdálenost každého z nich od nepřítele.

Druhá fáze: V této fázi si budeme udržovat front hned několik, pro každou vzdálenost od nepřátelských měst jednu. Dále si pro každé město budeme zaznamenávat, zda jsme v něm už byli. Také si budeme pamatovat dosud největší nalezenou vzdálenost, kterou dokážeme udržet od nepřítele. Na začátku nastavíme udržitelnou vzdálenost od nepřítele na hodnotu vzdálenosti královského města od nepřítele a toto město vložíme do fronty pro příslušnou vzdálenost. U tohoto města také nastavíme, že jsme v něm už byli. Výpočet probíhá tak, že postupně vyzvedáváme města z fronty pro aktuální udržitelnou vzdálenost, dokud se tato fronta nevyprázdní. Když se fronta vyprázdní, snížíme udržitelnou vzdálenost o jedna a opět začneme vybírat města z příslušné fronty. Vždy, když vezmeme nějaké město z fronty, projdeme všechny jeho sousedy, u dosud nenavštívených z nich nastavíme příznak, že už jsme je nenavštívili, a přidáme je do fronty — jestliže je vzdálenost takového města od nepřátelských měst větší, než je aktuální udržitelná vzdálenost, přidáme vrchol do fronty odpovídající aktuální udržitelné vzdálenosti, jinak město přidáme do fronty odpovídající jeho vzdálenosti od nepřátelských měst. Druhá fáze končí, jakmile vybereme z fronty cílové město. Aktuální udržitelná vzdálenost je pak výslednou udržitelnou vzdáleností.

Třetí fáze: Tato fáze představuje opět prosté prohledávání do šířky. Pro každé město si pamatujeme, zda jsme v něm již byli, a pokud ano, zaznamenáme si také město, ze kterého jsme do něj přišli. Opět používáme frontu na dosud nezpracovaná města. Na začátku vložíme do fronty cílové město. U něj nastavíme, že jsme v něm již byli, a jako jeho předchůdce nastavíme je samé. V každém kroku výpočtu pak vezmeme jedno město z fronty a projdeme všechny jeho sousedy. Každého souseda, kterého jsme dosud nenavštívili a jehož vzdálenost od nepřátelských měst je větší nebo rovna výsledné udržitelné vzdálenosti, označíme jako navštíveného a přidáme ho na konec fronty. Také u něj jako město, ze kterého jsme přišli, nastavíme právě vybrané město. Prohledávání končí ve chvíli, když je z fronty vyzvednuto počáteční

(královské) město. Poté už jenom projdeme cestu z počátečního do cílového města (to je velmi snadné díky odkazům na města, odkud jsme do nich při prohledávání přišli) a cestu vypíšeme.

Algoritmus má časovou složitost $O(M + N)$, kde M je počet cest a N je počet měst.

Správnost algoritmu budeme ukazovat opět po fázích. To, že algoritmus spočte správně vzdálenosti od nepřátelských měst v první fázi, plyne z následujícího: Na počátku mají všechny vrcholy se vzdáleností nula tuto vzdálenost přiřazenu. V okamžiku, kdy jsou zpracovány všechny vrcholy vzdálenosti nula, prošli jsme všechny jejich sousedy, přiřadili jsme jim vzdálenost jedna a zařadili je do fronty. Protože jiné vrcholy vzdálenost jedna mít nemohou, je vzdálenost jedna přiřazena právě všem správným vrcholům. Tuto úvahu lze snadno zobecnit pro libovolnou vzdálenost D. Prohledávání tedy skutečně určí vzdálenosti od nepřátelských měst správně.

Ve druhé fázi se správně spočítá maximální udržitelná vzdálenost od obsazených měst. Sledujeme v ní totiž souběžně všechny možné trasy vedoucí z počátečního města tak dlouho, dokud dokážeme udržet vzdálenost počátečního města (výsledná vzdálenost od nepřítele zřejmě nemůže být větší než vzdálenost počátečního města). Když už neexistuje město s dostatečně velkou vzdáleností, do kterého bychom mohli jít, snížíme udržitelnou vzdálenost o jedna. Všechny vrcholy se vzdáleností o jedna nižší, do kterých se dokážeme dostat přes vrcholy s dosavadní udržitelnou vzdáleností, máme již připraveny v příslušné frontě a začneme tedy prohledávat z nich. Protože udržitelnou vzdálenost snižujeme až když jsme se již dostali všude, kam to bylo možné, její výsledná hodnota bude zřejmě nejvyšší možná.

To, že ve třetí fázi nalezneme nejkratší trasu s danou vzdáleností, je zřejmé. Provádíme totiž jednoduché prohledávání do šířky s tím, že ignorujeme města s příliš malou vzdáleností od nepřítele. Nalezneme tedy určitě trasu s dostatečnou vzdáleností od nepřítele. Skutečnost, že to bude trasa nejkratší možná, plyne z vlastností prohledávání do šířky uvedených v první části důkazu.

Program je přímou implementací uvedeného algoritmu.