P-II-1 Síť

Zadanou úlohu si převedeme do řeči teorie grafů. Budovy firmy představují *vrcholy* našeho grafu, *hrany* grafu odpovídají možným propojením optickým kabelem. Úlohu vyřešíme nejprve pro případ $K = 1$. V tomto případě je naším úkolem vybrat takovou množinu hran, aby všechny vrcholy byly navzájem propojeny (ne nutně přímo). Taková množina hran se nazývá *kostra grafu* a jelikož chceme, aby součet cen hran v kostře byl co nejmenší, řešíme problém hledání *minimální kostry.*

Rozmyslete si, že minimální kostra grafu neobsahuje žádný cyklus – kdyby totiž nějaký obsahovala, mohli bychom jeho libovolnou hranu odstranit. Na druhé straně když do minimální kostry přidáme libovolnou hranu, vznikne nám cyklus, neboť vrcholy, mezi nimiž vede přidaná hrana, byly už spojeny pomocí nějakých hran kostry.

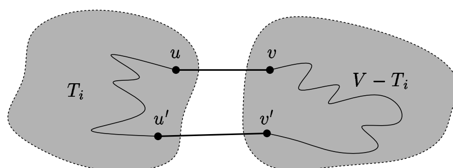
Hledání minimální kostry (Primův algoritmus). Algoritmus je založen na následující myšlence. Vrcholy grafu rozdělíme na dvě skupiny: na připojené a nepřipojené. Na začátku algoritmu zvolíme libovolný vrchol a prohlásíme ho za připojený, ostatní vrcholy jsou zatím nepřipojené. V každém kroku algoritmu připojíme jeden vrchol k dosud vytvořené síti následujícím způsobem. Najdeme nejkratší hranu spojující připojený a nepřipojený vrchol. Tuto hranu přidáme do sítě a její druhý konec se stane připojeným vrcholem. Skončíme ve chvíli, když jsou všechny vrcholy připojeny.

Aby byl algoritmus efektivní, potřebujeme umět rychle nalézt nejkratší hranu spojující připojený a nepřipojený vrchol. To zařídíme tak, že pro každý dosud nepřipojený vrchol si budeme pamatovat, ze kterého připojeného vrcholu k němu vede nejkratší hrana. Pokaždé, když přidáme k připojeným vrcholům další vrchol, musíme si informaci o nejbližších připojených vrcholech aktualizovat. Projdeme všechny nepřipojené vrcholy a pokud je nově připojovaný vrchol bližší, naši informaci změníme.

Skutečnost, že výsledná množina hran tvoří kostru, je zřejmá. Je však třeba dokázat, že je tato kostra minimální. Představme si libovolnou minimální kostru (dále ji budeme označovat MK) a porovnávejme ji s výsledkem našeho algoritmu (dále VNA).

Jestliže MK a VNA jsou shodné, VNA je minimální kostra. Předpokládejme tedy, že MK a VNA nejsou shodné. Nechť T_i je množina připojených vrcholů po i-tém kroku našeho algoritmu. Seřadíme hrany ve VNA podle toho, jak jsme je přidávali, a najdeme první hranu, která se vyskytuje ve VNA, ale není obsažena v MK. Nechť tato hrana byla přidána v kroku $i + 1$ a nechť spojuje vrchol $u \in T_i$ a vrchol $v \notin T_i$.

Přidejme hranu (u, v) do MK. Tím vznikne v MK cyklus, který začíná v T_i, přejde po hraně (u, v) ven z T_i a potom se vrátí nějakou cestou zpět do T_i (viz obr. 1). Na této cestě musí existovat aspoň jedna hrana (u', v'), která má jeden konec v T_i a druhý konec mimo T_i. Cena této hrany musí být aspoň taková, jako je cena hrany (u, v). V opačném případě by si náš algoritmus v kroku $i + 1$ musel vybrat hranu (u', v') namísto hrany (u, v). Proto pokud hranu (u', v') odebereme z MK a přidáme tam místo ní hranu (u, v), cena MK se nezvýší. Nemůže se však ani snížit, neboť MK je minimální. Upravená MK bude tedy nadále minimální kostrou v grafu. Navíc VNA a MK se nyní shodují v prvních $i + 1$ hranách. Stejným způsobem postupně přeměníme MK na VNA, přičemž nezvýšíme její cenu, takže VNA musí být také minimální kostrou.



Přidáním hrany (u, v) vznikne v MK cyklus, který začíná v T_i, přejde po hraně (u, v) ven z T_i a potom se vrátí nějakou cestou zpět do T_i.

Řešení pro obecné K. Dosud jsme předpokládali $K = 1$. Jestliže $K > 1$, nemusíme hranami pospojovat všechny vrcholy. Ke komunikaci totiž můžeme využít také Internet. Stačí, když se naše síť bude skládat z K souvislých částí, v každé z těchto souvislých částí vybereme jeden vrchol, který připojíme na Internet, a tak bude moci komunikovat každý vrchol s každým.

Takovouto síť můžeme získat například odebráním $K - 1$ nejdražších hran z minimální kostry MK. Tím se nám totiž MK rozpadne právě na K souvislých částí. Jediným problémem je ukázat, že toto řešení je skutečně nejlevnější možné.

Označme tedy symbolem P množinu $K - 1$ nejdražších hran kostry MK. Jejich odebráním z MK dostaneme množinu hran Q, která se skládá z K souvislých částí. Nechť existuje levnější množina hran T, která rovněž tvoří síť složenou z K souvislých částí.

Budeme uvažovat graf tvořený kostrou MK a hranami z množiny T. Jelikož už MK je souvislá, tento graf je jistě souvislý. Proto lze zvolit několik hran z MK, jimiž se dají jednotlivé komponenty T pospojovat. Každá přidaná hrana spojí dvě komponenty do jedné větší, takže stačí přidat $K - 1$ hran. Množinu těchto přidaných hran označíme symbolem S.

Všimněte si následujících dvou skutečností:

- Množina hran S určitě není dražší než P, neboť obě obsahují $K - 1$ hran z kostry MK, ale P jsme vybrali tak, aby obsahovala nejdražší hrany.
- Podle našeho předpokladu množina hran T je levnější než množina hran Q.

Z toho ale vyplývá, že kostra $S \cup T$ je levnější než MK $= P \cup Q$, což je spor s tím, že MK je minimální kostra. Tím jsme ukázali, že k vyřešení úlohy stačí z MK odebrat $K - 1$ nejdražších hran.

Časová složitost. Při hledání minimální kostry se v každém kroku přidá jeden vrchol do množiny připojených, vykoná se tedy celkem $N - 1$ kroků. V každém kroku nejprve v čase $O(N)$ najdeme nejkratší hranu spojující připojený a nepřipojený vrchol. Potom aktualizujeme informaci o nejbližším připojeném vrcholu pro všechny dosud nepřipojené vrcholy. Tato aktualizace představuje opět provedení $O(N)$ operací. Celková časová složitost je proto kvadratická, tj. $O(N^2)$.

Z výsledné minimální kostry potom potřebujeme odebrat $K - 1$ nejdražších hran. To můžeme udělat tak, že hrany kostry setřídíme. Třídění lze provést v čase $O(N \log N)$, v tomto případě nám ovšem stačí použít jednoduché třídění pracující v čase $O(N^2)$. Celková časová složitost je $O(N^2)$.

```
program Sit_P_II_1;

const
   maxn = 1000;
   nekonecno = 10000;

var
   N,K: integer; { počet budov, počet komponent }
   a: array[1..maxn,1..maxn] of integer; { ceny spojení }
   sit: array[1..maxn,1..2] of integer; { seznam hran výsledné sítě }

procedure nacti_vstup;
var
   i,j: integer;
begin
   write('Počet budov N:'); readln(N);
   write('Počet internetových připojení K:'); readln(K);
   for i:=1 to N do
      for j:=i+1 to N do begin
         write('Cena (',i,',',j,'):'); readln(a[i,j]);
         a[j,i]:=a[i,j];
      end;
end; {nacti_vstup}

procedure minimalni_kostra;
{najde minimální kostru a uloží její hrany do pole sit}
var
   pripojene: array[1..maxn] of boolean;
   nej: array[1..maxn] of integer;
   i,j: integer;
   min, nejlepsi: integer;

begin
   {na začátku je jenom vrchol 1 připojený}
   pripojene[1]:=true;
   for i:=2 to N do begin
      {v poli nej si budeme pro každý dosud nepřipojený vrchol udržovat nejbližší připojený vrchol}
      pripojene[i]:=false;
      nej[i]:=1;
   end;

   for i:=1 to N-1 do begin
      {najdeme nejkratší hranu, která spojuje připojený a nepřipojený vrchol}
      min:=nekonecno;
      for j:=1 to N do begin
         if not pripojene[j] then begin
            if a[j,nej[j]]<min then begin
               nejlepsi:=j; min:=a[j,nej[j]]
            end;
         end;
      end;

      {nalezený vrchol připojíme}
      pripojene[nejlepsi]:=true;
```

```
      {spojení (nejlepsi,nej[nejlepsi]) patří do sítě}
      sit[i,1]:=nejlepsi; sit[i,2]:=nej[nejlepsi];

      {přepočítáme pole nej: pro každý vrchol zjistíme, zda právě
      připojený vrchol nezkrátí jeho vzdálenost k připojeným vrcholům}
      for j:=1 to N do begin
         if not pripojene[j] then begin
            if a[j,nej[j]]>a[j,nejlepsi] then nej[j]:=nejlepsi;
         end;
      end;
   end;
end; {minimalni_kostra}

procedure utrid_hrany_site;
{setřídí hrany sítě od nejlevnější po nejdražší}
var
   i,j,k,min: integer;
begin
   for i:=1 to N-1 do begin
      min:=i;
      for j:=i+1 to N-1 do begin
         if a[sit[j,1],sit[j,2]]<a[sit[min,1],sit[min,2]] then
            min:=j;
      end;
      k:=sit[min,1]; sit[min,1]:=sit[i,1]; sit[i,1]:=k;
      k:=sit[min,2]; sit[min,2]:=sit[i,2]; sit[i,2]:=k;
   end;
end; {utrid_hrany_site}

procedure vypis_vysledek;
{vypíše výsledné hrany sítě}
var
   i: integer;
begin
   writeln('Je třeba vybudovat spojení mezi následujícími budovami:');
   for i:=1 to N-K do begin
      writeln('(',sit[i,1],',',sit[i,2],')');
   end;
end; {vypis_vysledek}

begin
   nacti_vstup;
   minimalni_kostra;
   utrid_hrany_site;
   vypis_vysledek;
end.
```

P-II-2 AttoSoft

Uvažujme, který ze studentů má pracovat u počítače v daném okamžiku t. Zřejmě to musí být jeden z těch studentů, kteří už přišli do firmy, ale ještě nedokončili svůj program. O těchto studentech řekneme, že jsou *aktivní* v čase t. Dokážeme, že v optimálním řešení vždy můžeme poslat pracovat na počítači toho z aktivních studentů, který musí odejít nejdříve (nechť je to student a). Kdyby totiž existoval rozvrh, ve kterém v čase t pracuje nějaký jiný student b, pak se student a musí dostat k počítači ještě v čase t' někdy mezi t a časem odchodu t_a. Student b však odchází v čase $t_b \geq t_a$. Proto můžeme sestrojit nový rozvrh, v němž necháme studenta a chvíli pracovat na počítači v čase t a stejně dlouhou dobu potom necháme studenta b pracovat na počítači v čase t'. Jestliže byl původní rozvrh správný, je správný i takto modifikovaný rozvrh, neboť každý pracuje stejně dlouho jako v původním rozvrhu a každý pracuje před svým odchodem.

Dokázali jsme, že v každém okamžiku může pracovat ten z aktivních studentů, který musí nejdříve odejít. Kdy tedy může dojít ke změně obsazení počítače? Buď tehdy, když přijde nový student a potřebuje odejít dříve, než student právě pracující (v tom případě se vystřídají u počítače), nebo když nějaký student dokončí svůj program a uvolní počítač.

Náš algoritmus bude sestrojovat rozvrh obsazení počítače postupně od začátku do konce. Studenty seřadíme podle času jejich příchodu a u každého si zaznamenáme, jak dlouho ještě potřebuje pracovat. Budeme si také udržovat množinu aktivních studentů. V každém kroku algoritmu najdeme nejbližší událost, jež může ovlivnit rozvrh. Touto událostí je buď příchod studenta nebo ukončení práce právě pracujícího studenta. V obou případech zaktualizujeme datové struktury a potom najdeme aktivního studenta s nejbližším odchodem a přidělíme mu počítač. Pokud tento student již nemá dost času na

dokončení svého programu, oznámíme, že všechny programy nelze dokončit. Správnost tohoto tvrzení přímo vyplývá z důkazu uvedeného výše.

Seřazení studentů je možné provést v čase $O(N \log N)$. Počet událostí je $2N$, neboť každý student jednou přijde a jednou dokončí program. Při každé události potřebujeme najít aktivního studenta s nejmenším časem odchodu. Kdybychom kvůli tomu vždy procházeli všechny aktivní studenty, dostali bychom algoritmus s časovou složitostí $O(N^2)$. Algoritmus se však dá zefektivnit, jestliže uložíme aktivní studenty do haldy uspořádané podle času jejich odchodu. V takovém případě zpracování jedné události trvá jen $O(\log N)$ – když někdo přišel, vložíme ho do haldy, když někdo dokončil práci, z haldy ho odstraníme. Poté se podíváme na minimum v haldě – studenta, který bude od této chvíle pracovat u počítače. Celková časová složitost algoritmu s použitím haldy je tedy pouze $O(N \log N)$.

```
program AttoSoft_P_II_2;

type student =
   record
      prichod, odchod, zbyva: integer;
   end;

const Nekonecno = 10000;

var A: array [1..1000] of student;      { pole studentů }
    N: integer;                         { počet studentů }
    Halda: array [1..1000] of integer;  { halda podle času odchodu }
    Halda_N: integer;                   { počet studentů v haldě }

procedure trid;
begin
   { vynechána kvůli úspoře místa }
end; { trid }

procedure vloz_do_haldy(student: integer);
var i, rodic, tmp:  integer;
begin
   { vlož studenta na konec haldy a posouvej ho nahoru }
   Halda_N := Halda_N + 1;
   Halda[Halda_N] := student;
   i := Halda_N;
   while i>1 do begin
      rodic := i div 2;
      if A[Halda[i]].odchod < A[Halda[rodic]].odchod then begin
         tmp := Halda[i];
         Halda[i] := Halda[rodic];
         Halda[rodic] := tmp;
      end;
      i := rodic;
   end;
end; { vloz_do_haldy }

procedure vyber_z_haldy;
var i, dite, tmp:  integer;
begin
   { první prvek nahraď posledním a posouvej ho dolů }
   Halda[1] := Halda[Halda_N];
   Halda_N := Halda_N - 1;
   i := 1;
   while i*2<=Halda_N do begin
      dite := i*2;
      if (i*2+1<=Halda_N)
         and (A[Halda[i*2+1]].odchod < A[Halda[dite]].odchod) then begin
            dite := i*2+1;
         end;
      if A[Halda[i]].odchod > A[Halda[dite]].odchod then begin
         tmp := Halda[i];
         Halda[i] := Halda[dite];
         Halda[dite] := tmp;
```

```
      end;
      i := dite;
   end;
end; { vyber_z_haldy }

procedure nacti;
var i: integer;
begin
   readln(N);
   for i := 1 to N do begin
      readln(A[i].prichod, A[i].odchod, A[i].zbyva);
   end;
end; { nacti }

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

var Pracuje, Skonci: integer;      { kdo právě pracuje a kdy skončí }
    Stary_cas, Novy_cas: integer;  { aktuální a předcházející událost }
    i: integer;                    { index do pole A }

begin
   nacti;                          { načti studenty do pole A }
   A[N+1].prichod := Nekonecno;    { zarážka }
   trid;                           { seřaď prvky pole A podle položky "prichod" }
   Pracuje := -1;                  { nikdo nepracuje }
   Skonci := Nekonecno;
   Halda_N := 0;                   { inicializace haldy }
   i := 1;
   Stary_cas := 0;
   while (i <= N) or (i = N+1) and (Pracuje > 0) do begin
      { Najdi novou událost }
      Novy_cas := min(A[i].prichod, Skonci);
      writeln('Čas ',Novy_cas);
      { Pokud někdo pracoval u počítače, vyhoď ho }
      if Pracuje > 0 then begin
         writeln(Novy_cas, ': student ', Pracuje, ' od počítače.');
         A[Pracuje].zbyva := A[Pracuje].zbyva - (Novy_cas - Stary_cas);
         if A[Pracuje].zbyva = 0 then vyber_z_haldy;
      end;

      { Jestliže událostí je příchod, vlož do haldy a jdi na další příchod }
      if (i <= N) and (A[i].prichod < Skonci) then begin
         vloz_do_haldy(i);
         i := i+1;
      end;

      { Najdi nového studenta k počítači }
      if Halda_N > 0 then begin
         Pracuje := Halda[1];
         Skonci := Novy_cas + A[Pracuje].zbyva;
         if Skonci > A[Pracuje].odchod then begin
            writeln('Rozvrh neexistuje!');
            exit;
         end;
         writeln(Novy_cas, ': student ', Pracuje, ' k počítači.');
      end
      else begin
         Pracuje := -1;
         Skonci := Nekonecno;
      end;
      Stary_cas := Novy_cas;
```

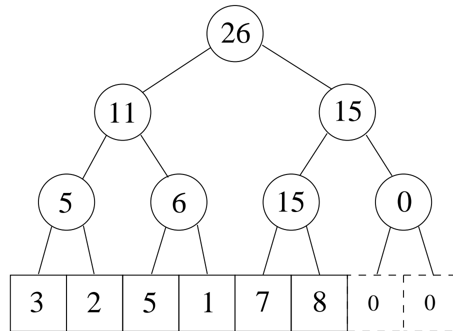
```
   end;
end.
```

P-II-3 Bageta

Na příkaz `KOLIK` c musí náš program odpovídat, v kolika dosud zadaných intervalech c leží. Na úvod uvedeme dvě triviální řešení. První je založeno na tom, že si budeme jednoduše pamatovat všechny dosud zadané intervaly a při každém příkazu `KOLIK` je všechny projdeme. Paměťová složitost tohoto řešení je $O(P)$, časová v nejhorším případě až $O(P^2)$. (Příkaz `PRIDEJ` dokážeme zpracovat v čase $O(1)$, ale na `KOLIK` potřebujeme v nejhorším případě až $O(P)$.) Trochu lepší řešení využívá pomocné pole velikosti $N+1$, v němž si pro každou celočíselnou pozici budeme pamatovat počet intervalů, které ji obsahují. Jeden interval přidáme v čase $O(N)$, na otázku odpovíme v čase $O(1)$. Výsledná časová složitost tohoto algoritmu je $O(N \cdot P)$, paměťová $O(N)$.

Uvědomte si, co vlastně potřebujeme zjistit, když nám přijde příkaz `KOLIK` c. Potřebujeme určit S – počet intervalů, které začínají na pozici $\leq c$ a končí na pozici $\geq c$. Nechť $Z(x)$ je počet intervalů, které začínají na pozici $\leq x$, a $K(x)$ je počet intervalů, které končí na pozici $\leq x$. Potom $S = Z(c) - K(c-1)$. (Intervaly, které končí před c, jsou započteny v $Z(c)$ i v $K(c-1)$, a proto je do S nezapočítáváme.) Stačilo by nám tedy umět rychle zjišťovat hodnoty $Z(x)$ a $K(x)$.

V řešení úlohy budeme využívat myšlenku z domácího kola – datovou strukturu, kterou jsme nazvali *intervalový strom**. Připomeňme si, o co šlo: Představte si, že nad polem A (jehož délku N jsme zvětšili na nejbližší mocninu dvou) vybudujeme úplný binární strom. Jeho listy budou odpovídat jednotlivým prvkům pole A, každý vyšší vrchol tohoto stromu bude odpovídat nějakému intervalu v poli A (přesněji bude odpovídat prvkům určeným listy z jeho podstromu). V každém vrcholu stromu si budeme pamatovat součet čísel v příslušném intervalu pole. Změnit hodnotu v poli A (a příslušně upravit součty ve vrcholech stromu) dokážeme v čase $O(\log N)$, zjistit součet libovolného intervalu v poli A dokážeme rovněž v čase $O(\log N)$.



$Z(c)$ je vlastně součet počtů intervalů začínajících na pozicích 0, 1, 2, ..., c. Budeme mít pole, ve kterém si tyto počty budeme pamatovat, a nad ním vybudovaný *intervalový strom*. Každé přidání intervalu změní jednu hodnotu v poli, tuto změnu dokážeme uskutečnit v čase $O(\log N)$. Analogicky budeme používat druhé pole (a druhý *intervalový strom*) pro počty intervalů, které na jednotlivých pozicích končí. Pomocí těchto datových struktur dokážeme každou hodnotu Z a K spočítat v čase $O(\log N)$.

Detailnější popis obou operací s *intervalovým stromem* a jeho implementaci v poli najdete ve vzorových řešeních domácího kola. Časová složitost našeho vzorového řešení je $O(P \log N)$ a paměťová $O(N)$. Všimněte si, že by nám stačilo udržovat jedno pole. Přidání intervalu $\langle a, b \rangle$ by znamenalo např. zvýšení hodnoty na pozici a a snížení hodnoty na pozici $b+1$.

```
program Bageta_P_II_3;

var ZZ,KK: array[0..3000047] of longint; { stromy pro Z a K }
    N,oldN,a,b,c,kde: longint;
    prikaz,pom: char;

function Soucet(var T: array of longint; delka, koren, interval: longint): longint;
{T - pole, v němž počítáme součty (všimněte si: "var T", ne "T" -- proč?)
 delka - délka intervalu, jehož součet počítáme
 koren - kořen podstromu, ve kterém ho počítáme
 interval - délka intervalu odpovídajícího kořenu (abychom ji nemuseli počítat)}
begin
   if (delka=0) then begin Soucet:=0; exit; end;
   if (interval=1) then begin Soucet:=T[koren]; exit; end;
   if (delka<=(interval div 2))
      then Soucet:=Soucet(T,delka,2*koren,interval div 2)
      else Soucet:=T[2*koren]+Soucet(T,delka-(interval div 2),2*koren+1,interval div 2);
end;
```

* Neplést si s intervaly ze zadání!

```
begin
   fillchar(ZZ,sizeof(ZZ),0);
   fillchar(KK,sizeof(KK),0);
   readln(oldN); N:=1; while (N<oldN+1) do N:=N*2; { upravíme velikost pole }

   while not eof do begin
      read(prikaz); pom:=prikaz; while (pom<>' ') do read(pom);
      if (prikaz='P') then begin
         readln(a,b);
         kde:=a+N; while (kde>=1) do begin Inc(ZZ[kde]); kde:=kde div 2; end;
         kde:=b+N; while (kde>=1) do begin Inc(KK[kde]); kde:=kde div 2; end;
      end else begin
         readln(c);
         writeln(Soucet(ZZ,c+1,1,N) - Soucet(KK,c,1,N));
      end;
   end;
end.
```

P-II-4 Registrový počítač

Představte si, že bychom kromě registrů měli k dispozici ještě jeden *zásobník**. Potom bychom již úlohu dokázali snadno vyřešit: Procházíme vstupním slovem zleva doprava a přečtená písmena vkládáme do zásobníku. Až potom budeme ze zásobníku písmena odebírat, budeme je dostávat v opačném pořadí, než v jakém byla do zásobníku vložena. Vrátíme se proto na začátek slova a budeme porovnávat, zda je slovo stejné odpředu jako odzadu. Vždy přečteme jedno písmeno ze vstupu, vyzvedneme jedno písmeno ze zásobníku a porovnáme je. Skončíme, když někdy dostaneme dvě různá písmena (slovo je špatné) nebo když dočteme celé vstupní slovo (slovo je správné).

Kdybychom tedy měli k dispozici zásobník, máme úlohu vyřešenou. Zásobník si však dokážeme simulovat v jednom registru (s pomocí druhého)! Jak na to? Písmena `a`, `b`, `c`, `d` budou odpovídat číslicím 1, 2, 3, 4. Číslo uložené v registru R_1 bude představovat obsah zásobníku – když ho zapíšeme v poziční soustavě o základu 5, jednotlivé cifry budou představovat hodnoty vložené do zásobníku (cifra na místě jednotek bude naposledy vložená hodnota). Například když do prázdného zásobníku vložíme postupně písmena `a`, `c`, `b`, `a` bude v R_1 hodnota $a\times5^3+c\times5^2+b\times5+a = 1\times5^3+3\times5^2+2\times5+1 = 125+75+10+1 = 211$.

Jak ale s takovýmto registrem-zásobníkem pracovat? Vložit novou hodnotu x je jednoduché – pomocí registru R_2 vynásobíme obsah R_1 pěti a potom ho x-krát zvětšíme o 1. Vyzvednout naposledy vloženou hodnotu také není těžké – je to přesně opačná operace. Vydělíme obsah registru R_1 pěti, zbytek po dělení je naposledy vložená hodnota, podíl (který dostaneme v R_2) je nový obsah zásobníku bez této hodnoty.

Máme tedy funkční řešení úlohy, které potřebuje dva registry. Pokusíme se však nalézt řešení ještě lepší. Jen s jedním registrem se nám už nepodaří simulovat zásobník a musíme proto vymyslet něco jiného.

Nejprve trochu terminologie: aktuální písmeno se bude v našem řešení pohybovat sem a tam po vstupním slově. Kvůli názornosti místo "aktuální je i-té písmeno vstupního slova", resp. "přesuneme aktuální písmeno doleva/doprava" budeme říkat "stojíme na pozici i", resp. "jdeme doleva/doprava". Délku vstupního slova budeme značit n.

Představte si, že stojíme na pozici i (přičemž ale i si nijak nepamatujeme, v R_1 je nula). Chtěli bychom písmeno na této pozici porovnat s jemu odpovídajícím písmenem na pozici $n+1-i$. Náš program ovšem nezná n ani i. Jak na to? Písmeno na naší pozici si zapamatujeme v proměnné. Nyní si zjistíme i. Jdeme doleva, dokud nepřijdeme na začátek vstupního slova, a zvyšujeme R_1. Odpovídající písmeno je i-té od konce. Není tedy těžké dojít k němu – přejdeme na konec slova, potom zmenšujeme R_1 a jdeme doleva, dokud v R_1 není nula. Písmena porovnáme a jsou-li různá, končíme. Jinak se potřebujeme vrátit zpět na pozici, kde jsme začínali. K tomu použijeme úplně stejný postup: Cestou doprava spočítáme v registru R_1 potřebný počet kroků, přesuneme se na začátek slova a vykonáme stejný počet kroků směrem doprava. Tím jsme se dostali do stejné situace, v níž jsme začínali, jen máme porovnané aktuální písmeno s jemu odpovídajícím písmenem. Celý tento postup budeme nazývat *porovnání*.

Chtěli bychom postupně porovnat všechny navzájem si příslušející dvojice písmen. To ale není problém. Začínáme na prvním písmenu vstupu a provedeme *porovnání*. Pokud není první písmeno stejné jako poslední, skončili jsme, jinak pokračujeme. Přesuneme se doprava (na druhé písmeno) a vykonáme další *porovnání*. Takto pokračujeme tak dlouho, dokud neporovnáme n-té písmeno s prvním (a nezjistíme, že právě porovnané písmeno bylo již posledním písmenem vstupního slova).

```
var vstup: char;
    pismeno: byte;
begin
   while (vstup<>'$') do begin
         { v R1 máme nulu, začínáme porovnání }
      if (vstup='a') then pismeno:=1;
      if (vstup='b') then pismeno:=2;
      if (vstup='c') then pismeno:=3;
```

* Zásobník je datová struktura, která podporuje operace „vlož prvek“ a „odeber naposledy vložený prvek“.

```
      if (vstup='d') then pismeno:=4;
         { spočítáme, kde jsme }
      while (vstup<>'$') do begin Inc(R1); Left; end;
         { přejdeme na pravý konec }
      Right;
      while (vstup<>'$') do Right;
         { přejdeme na odpovídající pozici }
      while not Zero(R1) do begin Dec(R1); Left; end;
         { kontrola }
      if (pismeno=1) and (vstup<>'a') then Reject;
      if (pismeno=2) and (vstup<>'b') then Reject;
      if (pismeno=3) and (vstup<>'c') then Reject;
      if (pismeno=4) and (vstup<>'d') then Reject;
         { návrat zpět }
      while (vstup<>'$') do begin Inc(R1); Right; end;
      Left;
      while (vstup<>'$') do Left;
      while not Zero(R1) do begin Dec(R1); Right; end;
         { posun na další písmeno, které je třeba zkontrolovat }
      Right;
   end;
   Accept; { všechno správně }
end.
```