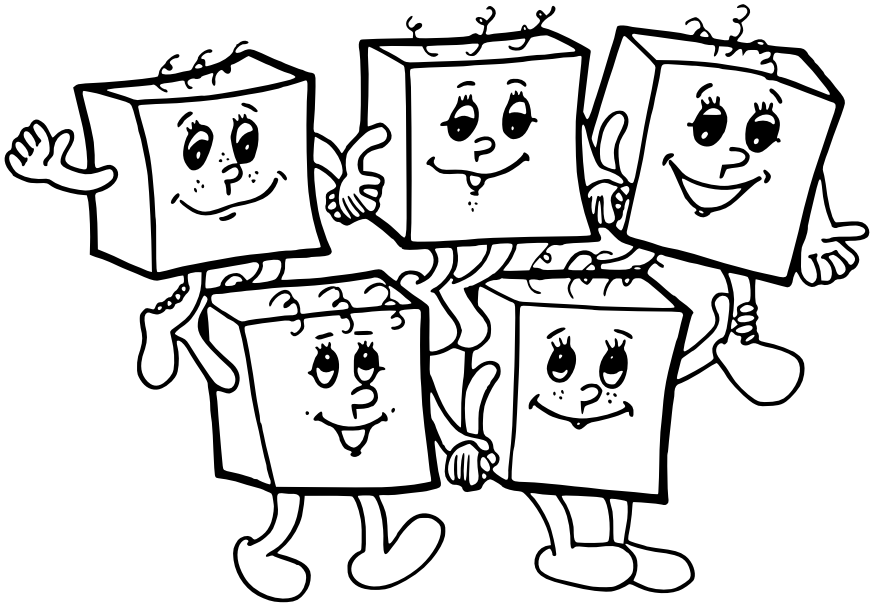
OLYMPIÁDA V INFORMATIKE
NA STREDNÝCH ŠKOLÁCH



dvadsiaty štvrtý ročník
školský rok 2008/09

riešenia krajského kola
kategória B

- **Olympiáda v informatike** je od školského roku 2006/07 samostatnou súťažou. Predchádzajúcich 21 ročníkov tejto súťaže prebiehalo pod názvom **Matematická olympiáda, kategória P** (programovanie).
- Oficiálnu **webstránku** súťaže nájdete na `http://oi.sk/`.

Riešenia kategórie B

B-II-1 Binárny súčet

Ukážeme dve rôzne, rovnako dobré riešenia zadanej úlohy.

Začneme od konca

Postupne budeme hľadať riešenie hlavolamu začínajúc najmenej významnou cifrou.

Niekedy rovno zistíme, že riešenie neexistuje (napr. pre posledné cifry `0+1=0`). Niekedy bude jediná možnosť, ako posledné cifry doplniť (napr. `0+*=0` alebo `1+*=0`). Občas budú tie možnosti dve, ale navzájom ekvivalentné (napr. `*+*=1` môžeme doplniť ako `1+0=1` aj ako `0+1=1` a je jedno, ktorú možnosť si vyberieme).

Lenže môže nastať aj situácia, v ktorej sa nebudeme priamo vedieť rozhodnúť, ako chýbajúce cifry doplniť: `*+*=0`. Tu totiž máme dve možnosti (`0+0=0` a `1+1=0`), ktoré ale nie sú ekvivalentné – pri druhej z nich nastáva prenos do vyššieho rádu. V tomto okamihu ešte nevieme povedať, ktorú z týchto možností si máme vybrať. Napr. ak by bolo celé zadanie `1*+0*=10`, treba si vybrať prvú možnosť, ale ak by bolo zadanie `01*+00*=100`, tak druhú.

(A niekedy sú dokonca oba výbery dobré, ako napr. pre zadanie `01*+0**=100`. Ale tým sa príliš trápiť nemusíme, keďže nám stačí nájsť jedno riešenie.)

Ako si s tým poradiť? Jeden možný prístup by bol skúsiť postupne obe možnosti doplnenia. Lenže takýto postup vedie k veľmi pomalému riešeniu. Napríklad pre `0*0*0*0*0* + 0********* = 1110101010` by sme takto vyskúšali 2^4 možností doplnenia, z ktorých by ani jedna nefungovala, lebo najvyššie cifry doplniť nevieme. Určite ľahko upravíte tento vstup na taký, pre ktoré by riešenie skúšajúce všetky možnosti potrebovalo prezrieť tých možností 2^{100}.

Omnoho lepšie riešenie dostaneme tak, že si jednoducho zapamätáme, že vieme najpravejšiu cifru doplniť dvoma spôsobmi – aj tak, aby prenos vznikol, aj tak, aby nevznikol.

A to už je vlastne celé riešenie. Budeme spracúvať zadané reťazce sprava doľava a pre každé k zistíme odpoveď na otázky: „Vieme pravých k stĺpcov korektne vyplniť?“ a „Vieme pravých k stĺpcov korektne vyplniť s tým, že vznikne prenos do vyššieho rádu?“. Keď spracúvam stĺpec $k+1$, stačí mi vedieť, ktoré možnosti pre k stĺpcov viem dosiahnuť, a vyskúšať všetky možnosti pre každú hviezdičku v ňom.

V nasledujúcom listingu programu počítame hodnoty `viem[k][x]` – „Viem vyplniť posledných `k` cifier tak, aby všetko sedelo a prenos bol `x`?“. Vždy, keď sa nám podarí nájsť jednu možnosť, ako to dosiahnuť, zapamätáme si v poli `ako` na pozíciách `ako[k][x][0..2]` jednu možnosť, aká musí byť cifra v reťazci A, cifra v B a prenos z rádu $k-1$.

Ak po spočítaní všetkých hodnôt v poli `viem` zistíme, že sa úloha dá riešiť, pomocou hodnôt v poli `ako` jedno riešenie spätným prechodom (zľava doprava) zostrojíme.

Pre každú cifru vyskúšame nanajvýš 8 možností, preto je časová aj pamäťová zložitosť lineárna od počtu cifier v zadaných číslach.

Listing programu:

```
var
   A, B, C : ansistring;
   N, k, lop, hip, loa, hia, lob, hib, prenos, ha, hb, hc, d1, d2 : longint;
   viem : array[0..1000047,0..1] of boolean;
   ako : array[0..1000047,0..1,0..2] of longint;

begin
   readln(A); readln(B); readln(C); N := length(A);
   viem[0][0]:=true; viem[0][1]:=false;
   for k := 1 to N do begin
      lop:=0; hip:=1; loa:=0; hia:=1; lob:=0; hib:=1;
      if not viem[k-1][0] then lop:=1;
      if not viem[k-1][1] then hip:=0;
      if A[N+1-k]<>'*' then begin loa:=ord(A[N+1-k])-48; hia:=loa; end;
      if B[N+1-k]<>'*' then begin lob:=ord(B[N+1-k])-48; hib:=lob; end;
      hc := ord(C[N+1-k])-48;
```

```
      viem[k][0]:=false; viem[k][1]:=false;
      for prenos := lop to hip do
         for ha := loa to hia do
            for hb := lob to hib do begin
               d1 := (prenos + ha + hb) mod 2;
               d2 := (prenos + ha + hb) div 2;
               if d1=hc then begin
                  viem[k][d2]:=true;
                  ako[k][d2][0]:=prenos;
                  ako[k][d2][1]:=ha;
                  ako[k][d2][2]:=hb;
               end;
            end;
   end;
   if viem[N][0] then begin
      prenos:=0;
      for k:=N downto 1 do begin
         A[N+1-k]:=chr(48+ako[k][prenos][1]);
         B[N+1-k]:=chr(48+ako[k][prenos][2]);
         prenos := ako[k][prenos][0];
      end;
      writeln(A); writeln(B);
   end else writeln('Nema riesenie.');
end.
```

Začneme od núl

Predstavme si, že by sme namiesto všetkých `*` napísali navzájom rôzne premenné. Napr. pre príklad zo zadania dostávame rovnosť `010ab0 + 01cd1e = 101001`. Preložené do matematickej reči, má platiť:

$$(2^4 + a2^2 + b2^1) + (2^4 + c2^3 + d2^2 + 2^1 + e2^0) = (2^5 + 2^3 + 2^0)$$

„Zozbierajme“ si na jednej strane členy obsahujúce neznáme, na druhej zvyšok:

$$c2^3 + a2^2 + d2^2 + b2^1 + e2^0 = (2^5 + 2^3 + 2^0) - (2^4) - (2^4 + 2^1)$$

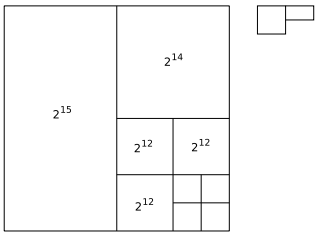
Všimnime si, čo sme na pravej strane dostali – hodnotu $C - A' - B'$, kde A' a B' vzniknú z A a B tak, že za všetky `*` dosadíme nuly.

Nech by zadanie vyzeralo akokoľvek, ľavá strana tejto rovnosti vždy bude nezáporná. Ak nám pravá strana vyjde záporná, môžeme prehlásiť, že úloha nemá riešenie – nech by sme chýbajúce cifry doplnili ako len chceme, vždy dostaneme priveľký výsledok.

Teraz ukážeme, že v opačnom prípade stačí premenné dopĺňať „od najväčšej po najmenšiu“ s tým, že vždy, keď môžeme, priradíme premennej hodnotu 1.

Začneme dôkazom jedného intuitívne zjavného tvrdenia.

Lema. Nech $a_1, \ldots, a_n$ sú nezáporné celé čísla menšie alebo rovné k a nech $2^{a_1} + \cdots + 2^{a_n} \geq 2^k$. Potom sa dá vybrať podmnožina a_i, pre ktoré dostaneme súčet presne 2^k.



Dôkaz. Bez ujmy na všeobecnosti nech $a_1 \geq a_2 \geq \cdots$. Nech q je najmenší index taký, že $2^{a_1} + \cdots + 2^{a_q} \geq 2^k$. Potom $2^{a_1} + \cdots + 2^{a_{q-1}} < 2^k$. Všetky sčítance sú deliteľné 2^{a_q}, preto je aj ich súčet deliteľný 2^{a_q}, a teda $2^{a_1} + \cdots + 2^{a_{q-1}} \leq 2^k - 2^{a_q}$. Lenže potom nutne $2^{a_1} + \cdots + 2^{a_q} \leq 2^k$, a teda $2^{a_1} + \cdots + 2^{a_q} = 2^k$.

Na obrázku vpravo je graficky znázornená myšlienka tvrdenia a dôkazu pre $k = 16$ a $a_1, a_2, \ldots = 15, 14, 12, 12, 12, 10, 10, 10, 10, 10, 9$.

Čo nám práve dokázaná lema hovorí o našej úlohe? Nás zaujíma, či existuje riešenie rovnice $x_1 2^{a_1} + \cdots + x_n 2^{a_n} = Q$, kde $x_i \in \{0, 1\}$.

Nech a_1 je najväčšie zo všetkých a_i. Sú dve možnosti: Ak $2^{a_1} > Q$, zjavne musí byť $x_1 = 0$, lebo by sme dostali priveľkú ľavú stranu. A práve dokázaná lema nám hovorí, že v opačnom prípade nič nepokazíme, ak vezmeme $x_1 = 1$. Prečo? Predstavme si, že existuje riešenie zadanej rovnice, v ktorom $x_1 = 0$. Do tohto riešenia sme vybrali niekoľko iných mocnín dvojky, ktorých celkový súčet je $Q \geq 2^{a_1}$. No a podľa lemy vieme vybrať niekoľko z nich tak, aby dali súčet presne 2^{a_1}. Lenže potom vieme zostrojiť nové riešenie, v ktorom všetky tieto mocniny dvoch vyhodíme (nastavíme ich x_i na 0) a namiesto nich použijeme 2^{a_1} (nastavíme x_1 na 1).

Inými slovami, práve sme dokázali: AK ($Q \geq 2^{a_1}$ a existuje nejaké riešenie), TAK (existuje riešenie, kde $x_1 = 1$). A tým sme dokázali, že naozaj funguje „pažravý“ postup, pri ktorom budeme neznáme dopĺňať zľava doprava použitím pravidla „ak môžeš, daj 1, ak nie, daj 0“.

Zostáva upresniť, ako to celé šikovne implementovať. Číslo na pravej strane rovnosti budeme celý čas reprezentovať ako reťazec bitov (nebudeme ho teda vyčísľovať). Rozdiel $Q = C - A' - B'$ vieme spočítať klasickým „druhá-trieda-ZŠ“ postupom v čase lineárnom od dĺžky reťazcov. Takisto si v lineárnom čase vieme zozbierať všetky mocniny dvoch, na ktorých máme v zadaní hviezdičky. No a teraz budeme postupne prechádzať zozbierané mocniny a pre každú rozhodneme, či tam musí byť 0 (a nič sa nedeje), alebo môže byť 1 (a potrebujeme zmenšiť Q).

Dalo by sa ukázať, že už tento algoritmus bude dokopy lineárny od dĺžky zadaných reťazcov, môžeme však spraviť jedno vylepšenie, po ktorom to už bude skutočne očividné.

Keďže na každej pozícii máme najviac dve hviezdičky, čokoľvek, čo dostaneme doplnením za všetky `*` na pozíciách 2^0 až 2^{k-1}, bude menšie ako 2^{k+1}. Ak teda po spracovaní pozície k máme $Q \geq 2^{k+1}$, vieme, že riešenie neexistuje a môžeme prestať. Preto pri každom zmenšení Q budeme meniť najviac dve binárne cifry Q, a teda bude celé toto riešenie lineárne.

(Na záver upozornenie: Toto riešenie naozaj musíme robiť postupne, najskôr spočítať $C - A' - B'$ a až potom dopĺňať hviezdičky. Ak by sme sa „pažravým“ spôsobom pokúšali doplniť hviezdičky priamo, zlyhalo by to napr. pre vstup `0**** + 00**1 = 01000`.)

Listing programu:

```
var
   A, B, C : ansistring;
   Q : array[1..1000047] of longint;
   N, i, j, qtop : longint;

begin
   readln(A); readln(B); readln(C); N := length(A);

   for i := 1 to N do Q[i] := ord(C[N+1-i])-48;
   for i := 1 to N do begin
      if Q[i]<0 then begin dec(Q[i+1]); inc(Q[i],2); end;
      if B[N+1-i]='1' then j:=1 else j:=0;
      if j <= Q[i] then dec(Q[i],j) else begin dec(Q[i+1]); inc(Q[i],2); dec(Q[i],j); end;
   end;
   if Q[N+1]<0 then begin writeln('Nema riesenie.'); halt; end;
   for i := 1 to N do begin
      if Q[i]<0 then begin dec(Q[i+1]); inc(Q[i],2); end;
      if A[N+1-i]='1' then j:=1 else j:=0;
      if j <= Q[i] then dec(Q[i],j) else begin dec(Q[i+1]); inc(Q[i],2); dec(Q[i],j); end;
   end;
   if Q[N+1]<0 then begin writeln('Nema riesenie.'); halt; end;
   qtop:=N; while (qtop>0) and (Q[qtop]=0) do dec(qtop);

   for i := N downto 1 do begin
      if qtop > i+1 then begin writeln('Nema riesenie.'); halt; end;
      if A[N+1-i]='*' then begin
         if qtop < i then A[N+1-i]:='0' else begin
            A[N+1-i]:='1';
            dec(Q[i]); j:=i;
            while Q[j]<0 do begin inc(Q[j],2); dec(Q[j+1]); inc(j); end;
            while (qtop>0) and (Q[qtop]=0) do dec(qtop);
         end;
      end;
      if B[N+1-i]='*' then begin
         if qtop < i then B[N+1-i]:='0' else begin
            B[N+1-i]:='1';
            dec(Q[i]); j:=i;
            while Q[j]<0 do begin inc(Q[j],2); dec(Q[j+1]); inc(j); end;
            while (qtop>0) and (Q[qtop]=0) do dec(qtop);
         end;
      end;
   end;
   if qtop>0 then writeln('Nema riesenie.') else begin writeln(A); writeln(B); end;
end.
```

B-II-2 Výber dovolenky

Najprv sa zamyslime nad menej efektívnymi riešeniami. Môžeme skúšať všetky možné podpostupnosti a pre ne zrátať počet jednotlivých písmen. Ak je z každého písmena aspoň po jednom, tak postupnosť je dobrá. Možných začiatkov a koncov je zhruba N^2. Vyrátanie počtu písmen vyžaduje toľko operácií, aká je dlhá podpostupnosť, teda najviac N operácií. Dokopy dostávame časovú zložitosť $O(N^3)$.

Tento postup vieme jednoducho zrýchliť. Predstavme si, že sme vyrátali početnosť písmen pre podpostupnosť od i po j. Aby sme vyrátali početnosť pre podpostupnosť od i po $j+1$, nemusíme ju zbytočne prechádzať celú. Stačí zobrať početnosti pre podpostupnosť od i po j a zvýšiť početnosť písmena na pozícii $j+1$ o jedna. Teda hodnoty pre každú podpostupnosť vieme vyrátať v konštantnom čase – stačí nám jediná operácia. Podpostupností je spolu zhruba N^2 a takáto je aj výsledná časová zložitosť tohto algoritmu – $O(N^2)$.

Všimnime si ešte jednu vec. Ak už podpostupnosť od i po j obsahuje všetky tri písmená `a, b, c`, tak aj všetky dlhšie podpostupnosti začínajúce na pozícii i budú obsahovať všetky 3 písmená. Nasleduje kód, ktorý robí to, čo sme si práve popísali:

```
vysl := N + 1;
for i := 1 to N do
begin
    for j := 0 to 2 do p[j] := 0;
    for j := i to N do
    begin
        inc(p[ord(a[j]) - ord('a')]);
        if (p[0] > 0) and (p[1] > 0) and (p[2] > 0) then
        begin
            if (j - i + 1 < vysl) then vysl := j - i + 1;
            break;
        end;
    end;
end;
if (vysl <= N) then writeln(vysl)
else writeln('Neda sa.');
```

V programe kontrolujeme, či už postupnosť od i do j obsahuje všetky 3 písmená. Ak áno, tak vyskočíme z cyklu pomocou príkazu `break`. Týmto sme zložitosť algoritmu nezlepšili, ale táto myšlienka vedie k rýchlejšiemu riešeniu. Všimnime si, kedy pre dané i vyskakujeme z cyklu. Platí, že ak pre i vyskočíme z cyklu v momente, keď $j = t$, tak pre $i+1$ vyskočíme z cyklu najskôr v čase t. Inými slovami, najkratšia dobrá postupnosť začínajúca $(i+1)$, znakom nemôže končiť skôr ako najkratšia dobrá postupnosť začínajúca i-tym znakom.

Označme $f(i)$ také j, pre ktoré vyskočíme z cyklu. Ak z cyklu nikdy nevyskočíme, tak $f(i) = N+1$. Náš vzorový program bude fungovať tak, že bude postupne rátať hodnoty $f(i)$. Ako sme si už ukázali vyššie, bude platiť nerovnosť $1 \leq f(1) \leq f(2) \leq \ldots \leq f(N) \leq N+1$. Vďaka tejto nerovnosti vieme všetky tieto hodnoty vyrátať v lineárnom čase. Hodnotu $f(i+1)$ rátame tak, že začneme s hodnotou $f(i)$ a posúvame sa doprava, kým nevyhovuje. Doprava sa môžeme posunúť najviac o N, takže spolu vykonáme $O(N)$ operácií.

Ešte raz poriadnejšie popíšeme, čo sa stane na konci jednej iterácie cyklu pre `i`. Práve sme zistili hodnotu $f(i)$, vieme teda, že najkratší dobrý úsek, ktorý začína na pozícii i, končí na pozícii $f(i)$. A nielen to, my aj vieme presné počty písmen a, b a c v tomto úseku. Keď sa teraz pozrieme na znak `a[i]`, vieme z neho určiť presné počty a, b a c v úseku od $i+1$ po $f(i)$. Ak sú všetky tieto počty kladné, je $f(i+1) = f(i)$ a ide ďalšia iterácia vonkajšieho cyklu. Ak nie, ďalej zvyšujeme index konca postupnosti, až kým nebudú všetky tri počty znova kladné (alebo sa neminie vstup).

Listing programu:

```
var p : array[0..2] of integer;
    vysl, N, i, f : integer;
    x : string;
begin
    readln(N);
    readln(x);
    vysl := N + 1;
    f := 1;
    for i := 0 to 2 do p[i] := 0;
```

```
    for i := 1 to N do begin
        while f <= N do begin
            inc(p[ord(x[f]) - ord('a')]);
            if (p[0] > 0) and (p[1] > 0) and (p[2] > 0) then begin
                if (f - i + 1 < vysl) then vysl := f - i + 1;
                break;
            end;
            inc(f);
        end;
        dec(p[ord(x[i]) - ord('a')]);
    end;
    if (vysl <= N) then writeln(vysl)
    else writeln('Neda sa.');
end.
```

Alternatívne riešenie

O niečo pomalšie, ale taktiež efektívne riešenie je pre každé $x \in \{a, b, c\}$ a každé $i \in \{0, \ldots, N\}$ spočítať $p_{x,i}$: počet výskytov znaku x v prvých i znakoch vstupu. Tieto hodnoty vieme jednoducho spočítať v čase $O(N)$. A pomocou týchto hodnôt vieme k ľubovoľnému začiatku úseku i efektívne nájsť najbližší koniec j, pre ktorý už bude úsek od i po j dobrý. Hľadáme totiž najmenšie j také, že pre všetky tri x je $p_{x,j} > p_{x,i-1}$. (V úseku $1 \ldots j$ je viac výskytov znaku x ako v úseku $1 \ldots i-1$ práve vtedy, keď je v úseku $i \ldots j$ aspoň jedno x.) A najmenšie takéto j vieme nájsť v čase $O(\log N)$ binárnym vyhľadávaním v intervale od i po N. Celková časová zložitosť takéhoto riešenia je teda $O(N \log N)$.

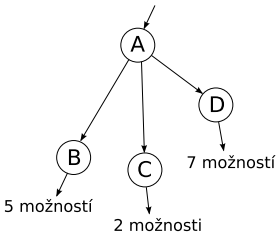
B-II-3 Sánkovanie

Riešenie „na papieri“

Na úvod sa zamyslime, ako by sa túto úlohu dalo šikovne riešiť, keby sme ju mali riešiť ručne. Niekto nám dá na papieri obrázok toho, ako to na kopci vyzerá, a my máme spočítať všetky cesty zhora dole.

Ide to samozrejme aj skúšaním všetkých možností, ľahko však prídeme na lepšiu myšlienku:

Všimnime si situáciu z obrázka vpravo. Keď sme práve na mieste A, máme na výber tri možnosti: ísť do B, do C, alebo do D. Ak by sme už vedeli, koľkými spôsobmi sa dá dostať do cieľa z každého z týchto miest, tak už vieme všetko, čo potrebujeme. Počet ciest z A do cieľa je jednoducho rovný súčtu počtov ciest z B, C a D do cieľa. V príklade na obrázku vpravo by sa teda z A dalo do cieľa dostať $5 + 2 + 7 = 14$ spôsobmi.



(Uvedomte si, že na ničom inom ako možnostiach, ktoré máme v prvom kroku, nám pri počítaní ciest z A nezáleží. Je nám napríklad úplne jedno, či bola možnosť ísť z D do C – ak aj bola, je už zarátaná v počte všetkých ciest z D do cieľa.)

Ručne teda odpoveď spočítame ľahko – stačí ísť po obrázku zdola hore a postupne pre každé miesto spočítať počet ciest, ktoré z neho vedú do cieľa.

Počty ciest, ktoré by sme dostali pre kopec z príkladu v zadaní, sú spolu s postupom ich počítania znázornené na druhom obrázku.

Prepísanie myšlienky do algoritmu

Jeden spôsob, ako našu úlohu vyriešiť, je rozdeliť si problém na dve časti. Najskôr si usporiadame miesta zdola hore, a následne spočítame počty ciest.

V tomto okamihu si treba uvedomiť, že na základe vstupných údajov sa nemusí dať zoradiť miesta podľa výšky jednoznačne. Napríklad pre príklad zo zadania vyhovuje aj poradie $7, 2, 6, 3, 4, 5, 1$, aj poradie $7, 4, 2, 5, 6, 3, 1$, aj veľa iných.

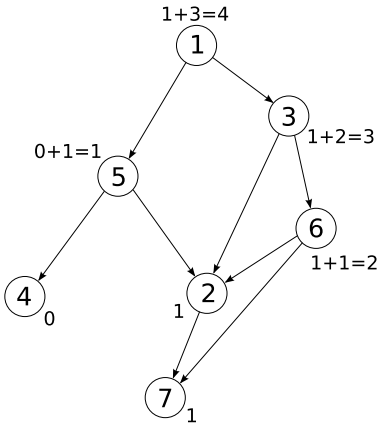
Dobrá správa však je, že nám je úplne jedno, ktoré poradie si vyberieme, všetky sú pre nás dobré. Jediné, čo potrebujeme, je, aby každé miesto bolo v nájdenom poradí skôr ako všetky, z ktorých sa doň vieme dostať.

Terminologická vsuvka. Matematickému modelu kopca zo zadania sa hovorí orientovaný acyklický graf. *Zaujímavé miesta sa volajú* vrcholy *a čiary medzi nimi sú* hrany. *Slovo* orientovaný *znamená, že každá hrana*

má smer a slovo acyklický*, že tam nie sú cykly – okruhy, po ktorých by sa dalo sánkovať dookola. To, čo chceme nájsť, sa odborne volá* topologické usporiadanie *vrcholov.*

Jedna možnosť, ako zostrojiť topologické usporiadanie vrcholov, je nasledovná. Pre každý vrchol si budeme pamätať dve veci: Počet hrán, ktoré doň vchádzajú, a čísla vrcholov, kam z neho vedú hrany.

Teraz si stačí uvedomiť, že akonáhle do niektorého vrcholu nevchádza žiadna hrana, môžeme ho prehlásiť za najvyšší, nič tým nepokazíme. Teraz zoberieme vrchol, ktorý sme práve spracovali, odstránime ho, a odstránime aj všetky hrany, ktoré z neho vedú. Čo sa stane? Môžu nám vzniknúť iné vrcholy, do ktorých nič nepovedie. Napríklad v grafe z príkladu v zadaní po tom, ako prehlásime 1 za najvyšší vrchol, už budú takéto vrcholy dva: 3 a 5.



A takto pokračujeme dokola ďalej, až kým sa nám všetky vrcholy neminú. V príklade by sme teraz napríklad mohli za druhý najvyšší prehlásiť vrchol 3, čím by nám pribudol nový kandidát: vrchol 6.

(Ak by mohol byť v grafe cyklus, časom by sme sa zasekli na tom, že do každého vrcholu už vchádza nejaká hrana. Ale keďže máme zaručené, že náš graf cykly neobsahuje, toto nepotrebujeme nijako ošetrovať.)

Dobrý spôsob, ako tento algoritmus implementovať, je pamätať si v ľubovoľnej rozumnej dátovej štruktúre (napríklad fronta alebo zásobník) množinu všetkých kandidátov. Aby bol náš algoritmus efektívny, stačí nám vedieť robiť v konštantnom čase dve operácie – pridať do množiny nového kandidáta a vybrať z množiny jedného (ľubovoľného) kandidáta.

Reprezentácia grafu v počítači

Najjednoduchší spôsob, ako reprezentovať graf, je pole rozmerov $N \times N$, kde je na súradniciach i, j zapísané, či existuje hrana z i do j. Takýto prístup je dobrý vtedy, ak je graf na vstupe „hustý“.

Lenže všimnime si, čo by sa stalo pre $N = 100\,000$. Aj ak by sme použili len jeden bit pamäte na hranu, ešte stále by sme potrebovali $100\,000^2/8$ bajtov $\sim 1.2TB$.

Navyše, ak by sme aj mali maximálny počet hrán $M = 1\,000\,000$, tak by v priemere z jedného vrcholu išlo 10 hrán. Ale my by sme museli prejsť 100 000 políčok v tabuľke na to, aby sme tých 10 hrán našli.

Šikovnejšia reprezentácia grafu je pamätať si presne to, čo využívame pri topologickom triedení – *výstupné stupne vrcholov* (t. j. počty hrán, ktoré z nich vedú) a pre každý vrchol zoznam vrcholov, kam z neho vedú hrany.

Implementácia riešenia používajúceho topologické triedenie

Ukážeme si teraz, ako všetko, čo sme doteraz vymysleli, implementovať v Pascale. V našom programe najskôr všetky hrany načítame do poľa, spočítame si stupne vrcholov, potom príslušne pozväčšujeme a vyplníme potrebné polia. Následne vyrobíme vyššie popísaným postupom jedno topologické usporiadanie vrcholov, a na záver ich v tomto poradí spracujeme a vypočítame hľadaný počet ciest.

Listing programu:

```
const MAXN = 100047; MAXM = 1000047;

var D : array[1..MAXN] of longint; { stupne vrcholov }
    G : array[1..MAXN] of array of longint; { G[i] je zoznam potomkov vrcholu i }
    T : array[1..MAXN] of longint; { topologicke usporiadanie vrcholov }
    N,M : longint;

    E : array[1..MAXM,1..2] of longint; { pomocne pole na hrany }
    Dpom : array[1..MAXN] of longint; { pomocne pole na aktualne stupne vrcholov }
    K : array[1..MAXN] of longint; { kandidati na dalsi najvyssi vrchol }

    P : array[1..MAXN] of longint; { P[i] je pocet ciest z i do N }
```

```
procedure nacitaj;
var i : longint;
begin
   { nacitame vstup }
   readln(N); readln(M);
   for i:=1 to M do readln(E[i][1],E[i][2]);
   { spocitame stupne vrcholov }
   for i:=1 to N do begin D[i]:=0; Dpom[i]:=0; end;
   for i:=1 to M do inc( D[ E[i][1] ] );
   { zvacsime riadky pola G na spravnu velkost }
   for i:=1 to N do setlength(G[i],D[i]);
   { vyplnime pole G }
   for i:=1 to M do begin
      G[ E[i][1] ][ Dpom[ E[i][1] ] ] := E[i][2];
      inc( Dpom[ E[i][1] ] );
   end;
end;

procedure utried;
var i, j, kde : longint;
    PK : longint; { pocet kandidatov }

begin
   { inicializacia: vyplnime Dpom, pridame ako kandidatov vsetkych co uz mozu }
   for i:=1 to N do Dpom[i]:=0;
   for i:=1 to N do for j:=0 to D[i]-1 do inc( Dpom[G[i][j]] );
   PK:=0;
   for i:=1 to N do if Dpom[i]=0 then begin inc(PK); K[PK]:=i; end;
   { dokola vyberieme a spracujeme kandidata, az kym sa vsetci neminu }
   for j:=1 to N do begin
      kde:=K[PK]; dec(PK); T[N+1-j]:=kde;
      { "kde" je prave spracuvany kandidat, ideme odstranit hrany z neho }
      for i:=0 to D[kde]-1 do begin
         dec(Dpom[ G[kde][i] ]);
         { ak tento pocet klesol na nulu, mame noveho kandidata }
         if Dpom[ G[kde][i] ]=0 then begin inc(PK); K[PK]:=G[kde][i]; end;
      end;
   end;
end;

procedure spocitaj;
var i, j, kde : longint;
begin
   for i:=1 to N do begin
      kde := T[i];
      if kde=N then P[kde]:=1 else P[kde]:=0;
      for j:=0 to D[kde]-1 do inc(P[kde],P[ G[kde][j] ]);
   end;
end;

begin
   nacitaj; utried; spocitaj; writeln(P[1]);
end.
```

Jednoduchšie implementovateľné riešenie

Na ten istý problém sa však dá pozrieť aj z opačnej strany. Na základe pozorovania z prvého obrázku vieme napísať jednoduchú rekurzívnu funkciu, ktorá bude počítať počet ciest. Vyzerala by nejak takto:

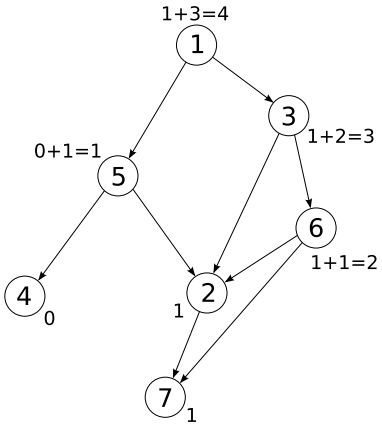
```
function pocet_ciest(odkial : longint) : longint;
begin
   if odkial=N then result:=1 { som dole }
   else begin
      result:=0;
      for i := 1 to stupen[odkial] do inc(result, pocet_ciest( potomok[odkial][i] ) );
   end;
end;
```

Keby sme takúto funkciu napísali poriadne a spustili, zistili by sme, že síce počet ciest zráta správne, ale zato veľmi pomaly – totiž postupne vyskúša všetky možné cesty, a tých môže byť vééééľmi veľa.

No ale my máme v rukáve jeden jednoduchý, ale o to účinnejší trik, odborne zvaný *memoizácia*. Stačí si uvedomiť, že akonáhle raz zistíme počet ciest z nejakého vrcholu, môžeme si ho zapísať do poľa s výsledkami.

A akonáhle budeme túto hodnotu niekedy v budúcnosti potrebovať, namiesto opätovného skúšania všetkých možností sa pozrieme do poľa, zistíme, že už vieme odpoveď, a rovno ju vrátime.

Takto by celý beh algoritmu vyzeral na príklade zo zadania.



- zavoláme `pocet_ciest(1)`
- zavoláme `pocet_ciest(5)`
- zavoláme `pocet_ciest(4)`
- `pocet_ciest(4)` spočíta, zapamätá si a vracia 0
- zavoláme `pocet_ciest(2)`
- zavoláme `pocet_ciest(7)`
- `pocet_ciest(7)` spočíta, zapamätá si a vracia 1
- `pocet_ciest(2)` spočíta, zapamätá si a vracia 1
- `pocet_ciest(5)` spočíta, zapamätá si a vracia 1
- zavoláme `pocet_ciest(3)`
- zavoláme `pocet_ciest(2)`
- `pocet_ciest(2)` **rovno vracia zapamätanú hodnotu 1**
- zavoláme `pocet_ciest(6)`
- zavoláme `pocet_ciest(2)`
- `pocet_ciest(2)` **rovno vracia zapamätanú hodnotu 1**
- zavoláme `pocet_ciest(7)`
- `pocet_ciest(7)` **rovno vracia zapamätanú hodnotu 1**
- `pocet_ciest(6)` spočíta, zapamätá si a vracia 2
- `pocet_ciest(3)` spočíta, zapamätá si a vracia 3
- `pocet_ciest(1)` spočíta, zapamätá si a vracia 4

A akoby zázrakom dostávame z pôvodne mizerného algoritmu efektívny. Všimnite si totiž, že každý vrchol spracujeme a vypočítame práve raz. Tiež si všimnite poradie, v akom náš algoritmus zistil a uložil vypočítané hodnoty: ako prvý sme sa dozvedeli vrchol 4, potom 7, potom 2, 5, 6, 3, a nakoniec 1. Toto samozrejme nie je nič iné ako jedno možné topologické usporiadanie, ktoré sme dostali úplne zadarmo ako bonus k riešeniu zadanej úlohy.

Listing programu:

```
const MAXN = 100047; MAXM = 1000047;

var D : array[1..MAXN] of longint; { stupne vrcholov }
    G : array[1..MAXN] of array of longint; { G[i] je zoznam potomkov vrcholu i }
    N,M : longint;
    E : array[1..MAXM,1..2] of longint; { pomocne pole na hrany }
    Dpom : array[1..MAXN] of longint; { pomocne pole na aktualne stupne vrcholov }
    P : array[1..MAXN] of longint; { P[i] je pocet ciest z i do N }

procedure nacitaj;
var i : longint;
begin
   readln(N); readln(M);
   for i:=1 to M do readln(E[i][1],E[i][2]);
   for i:=1 to N do begin D[i]:=0; Dpom[i]:=0; end;
   for i:=1 to M do inc( D[ E[i][1] ] );
   for i:=1 to N do setlength(G[i],D[i]);
   for i:=1 to M do begin
      G[ E[i][1] ][ Dpom[ E[i][1] ] ] := E[i][2];
      inc( Dpom[ E[i][1] ] );
   end;
   for i:=1 to N do P[i]:=-1; { DOLEZITE! inicializujeme pole P na hodnotu "neviem" }
end;

function pocet_ciest(kde : longint) : longint;
var vysledok, i, j : longint;
begin
   if P[kde] >= 0 then vysledok := P[kde] else
```

```
   if kde = N then vysledok := 1 else begin
      vysledok := 0;
      for i:=0 to D[kde]-1 do inc( vysledok, pocet_ciest(G[kde][i]) );
   end;
   P[kde] := vysledok; pocet_ciest := vysledok;
end;

begin
   nacitaj;
   writeln(pocet_ciest(1));
end.
```

Časová zložitosť riešení a poznámky k implementácii

Obe prezentované implementácie majú časovú aj pamäťovú zložitosť $O(M + N)$.

Pamäťová zložitosť je zjavná, naozaj si vystačíme s konečným počtom M- a N-prvkových polí, a ešte používame jedno dvojrozmerné pole `G`, ktoré dokopy obsahuje práve M záznamov.

Čo sa týka časovej zložitosti, tam si stačí uvedomiť, že pre každý vrchol len konečne veľa krát (dvakrát v prvom riešení, raz v druhom) spracujeme hrany, ktoré z neho vychádzajú.

Implementácia riešenia, ktoré sme prezentovali ako druhé v poradí, je výrazne stručnejšia ako u prvého riešenia. Pridáme ale pár varovných slov: Existuje veľa úloh, ktoré sa, podobne ako táto, dajú riešiť z oboch koncov – aj „zdola hore“ (takýto postup sa často volá dynamické programovanie), aj „zhora dole“ (to je práve predvedená memoizácia). Niekedy je lepší jeden prístup, niekedy druhý. Memoizácia sa väčšinou ľahšie implementuje. Takéto riešenie ale môže byť aj niekoľkokrát pomalšie od riešenia opačným smerom – máme totiž navyše réžiu s veľanásobným volaním funkcie.

A s tým súvisí aj druhé riziko – na vnorené volania funkcie treba mať dosť veľký zásobník (stack). Pri súčasných obmedzeniach v praxi (štandardná veľkost stacku je 8 MB) je 100 000 vnorení rekurzívnej funkcie približne na hranici toho, čo sa nám ešte na stack zmestí. Keby teda bola táto úloha zadaná ako praktická, nášmu druhému riešeniu by hrozilo, že na zákerných veľkých vstupoch môže namiesto správneho výsledku skončiť s chybou Stack overflow (Pretečenie zásobníka).

B-II-4 Banka

Podúloha a)

Stačí vziať čísla účtov 00, 11, 22, ..., 99. Ak sa v jednej cifre pomýlime, dostaneme číslo s dvoma rôznymi ciframi – teda nikdy takto nedostaneme iné číslo účtu.

Podúloha b)

Všetkých možných N-ciferných čísel účtu je 10^N.

Rozdeľme ich do 10^{N-1} skupín podľa prvých $N - 1$ cifier. V každej skupine teda dostaneme 10 čísel, ktoré sa navzájom líšia len v poslednej cifre.

Teraz si už len stačí uvedomiť, že z každej skupiny môžeme použiť nanajvýš jedno číslo – ak by sme z niektorej použili dve, zjavne by sa mohlo stať, že sa pri písaní jedného z nich pomýlime v poslednej cifre a napíšeme to druhé.

Podúloha c)

Najskôr ukážeme ľahšie vymysliteľný postup, ktorým dostaneme $\lceil 10^N/11 \rceil$ vyhovujúcich čísel účtu. Tento postup bude jednoduchým zovšeobecnením riešenia, ktoré sme našli v prvej podúlohe: Stačí ako čísla účtu brať násobky čísla 11.

Prečo to funguje? Z podobného dôvodu ako funguje kritérium deliteľnosti 11, s ktorým ste sa už možno stretli: číslo je deliteľné 11 práve vtedy, ak je 11 deliteľný jeho ciferný súčet so striedavými znamienkami. Obe veci si teraz dokážeme.

Zapíšme naše číslo účtu ako $U = \overline{a_{n-1}a_{n-2}\ldots a_1a_0}$. Hodnota tohto čísla je $a_0+10a_1+10^2a_2+\cdots+10^{n-1}a_{n-1}$. Teraz si stačí uvedomiť, že 10 dáva po delení 11 rovnaký zvyšok ako -1. A teda pre ľubovoľné k dáva 10^k rovnaký zvyšok ako $(-1)^k$. A samozrejme vieme, že pre nepárne k to je -1 a pre párne to je 1.

Tým sme práve dokázali, že naše číslo účtu dáva rovnaký zvyšok po delení 11 ako $U' = a_0 - a_1 + a_2 - \cdots + (-1)^{n-1}a_{n-1}$. Z tohto priamo vyplýva vyššie spomínané kritérium deliteľnosti 11.

A ako je to teda s číslami účtov? Všimnime si, že ak zmeníme v čísle U ľubovoľnú jednu cifru, hodnota U' sa určite zmení. A keďže cifru vieme zmeniť najviac o 9 (z 0 na 9 alebo naopak), zmena jednej cifry určite zmení zvyšok, aký dáva číslo po delení 11. To ale znamená, že ak sme pred zmenou mali číslo, ktoré bolo deliteľné 11, po zmene cifry už 11 deliteľné nebude. A to je presne to, čo potrebujeme.

Práve popísaná konštrukcia je síce dobrá, ale nie je optimálna. Optimálne riešenie, ktoré si popíšeme, bude založené na myšlienke *kontrolného súčtu*.

Čísla účtov budeme voliť takto: Prvých $N-1$ cifier zvolíme ľubovoľne, a následne poslednú zvolíme tak, aby bol ciferný súčet nášho čísla násobkom 10.

Poslednú cifru vždy vieme určiť jednoznačne – stačí sčítať prvých $N-1$ cifier a pozrieť sa na poslednú cifru výsledku, označme ju x. Ak $x = 0$, je posledná cifra čísla účtu 0, inak je to $10 - x$.

Týmto spôsobom teda dostaneme 10^{N-1} čísel účtov. A zjavne platí, že ak v čísle účtu zmeníme jednu cifru, zmeníme tým jeho ciferný súčet o 1 až 9. A teda ak pôvodné číslo malo ciferný súčet deliteľný 10, nové číslo určite nebude mať ciferný súčet deliteľný 10.

Podúloha d)

Obe banky majú presne rovnaký maximálny počet účtov, a to nie len pre 10-ciferné čísla účtov, ale dokonca pre ľubovoľné N. Požiadavky zo zadania sú totiž ekvivalentné. Ukážeme, prečo.

Požiadavka z Popletenej Viesky je zjavne splnená práve vtedy, keď sa každé dve čísla účtov líšia aspoň v piatich cifrách. (Ak by existovala dvojica čísel, ktorá sa líší na menej miestach, vieme max. 4 chybami z jedného vyrobiť druhé. V opačnom prípade sa to zjavne stať nemôže.)

Lenže aj požiadavka z Bezpečnej Triesky je splnená práve vtedy, keď sa každé dve čísla účtov líšia aspoň v piatich cifrách.

Totiž množina čísel účtov je pre túto banku zlá, ak existujú nejaké dva účty U_1 a U_2 také, že nejaké číslo U sa dá na nejaké ≤ 2 zmeny vyrobiť z U_1 a na nejaké iné ≤ 2 zmeny vyrobiť z U_2.

Ak také U existuje, tak vieme na ≤ 2 zmeny prerobiť U_1 na U, a na ďalšie ≤ 2 zmeny prerobiť U na U_2, preto sa U_1 a U_2 líšia na ≤ 4 miestach. A naopak, ak sa U_1 a U_2 líšia na $x \leq 4$ miestach, také U existuje – stačí zobrať U_1 a ľubovoľných $\lceil x/2 \rceil$ cifier kde sa líši od U_2 zmeniť na príslušné cifry U_2.

Tým sme ukázali, že množina čísel účtov je pre túto banku zlá práve vtedy, keď sa niektoré dve čísla účtov líšia na ≤ 4 miestach. Inými slovami, množina čísel účtov je dobrá práve vtedy, keď sa každé dve čísla účtov líšia aspoň v piatich cifrách.

(*Táto podúloha ukazuje dôležité pozorovanie z teórie samoopravných kódov: Ľubovoľný kód, ktorý dokáže rozpoznať $2k$ chýb, je zároveň kódom, ktorý dokáže opraviť k chýb, a naopak.*)

SLOVENSKÁ KOMISIA OLYMPIÁDY V INFORMATIKE

DVADSIATY ŠTVRTÝ ROČNÍK OLYMPIÁDY V INFORMATIKE

Zodpovedný redaktor: Michal Forišek
Sadzba programom LaTeX