Slovenská komisia Matematickej olympiády
FPEDaS ŽU, Veľký diel, 010 26 Žilina

MATEMATICKÁ OLYMPIÁDA NA STREDNÝCH ŠKOLÁCH

54. ročník, školský rok 2004/2005

Riešenia úloh 2. kola kategórie P

Upozornenie. Praktická časť celoštátneho kola MO, kat. P bude veľmi pravdepodobne celá prebiehať pod operačným systémom Linux, s kompilátormi FreePascal a gcc. Podrobnejšie informácie budú v dostatočnom predstihu zverejnené na stránkach `http://www.ksp.sk/mop/`.

P-II-1

Pri riešení tejto úlohy použijeme riešenie úlohy P-I-1 z domáceho kola. Jedinou zmenou oproti domácemu kolu je podmienka, že žiadny zákazník v dobe, keď bude prať, neuvidí dvoch rôznych zákazníkov používať tú istú práčku.

Inak povedané, práčka, ktorú používal zákazník Cyril, môže byť použitá až v momente, keď zo salónu odíde posledný zákazník, ktorý pral prádlo súčasne s Cyrilom. Povedzme, že zo všetkých zákazníkov, ktorí prali prádlo zároveň s Cyrilom, je Metod ten, ktorý odíde zo salónu najneskôr. Potom práčku, ktorú používal Cyril, môže použiť ďalší zákazník až po odchode Metoda zo salónu.

Toto pozorovanie využijeme tak, že z pôvodných zákaziek vytvoríme nové zákazky, ktoré budeme nazývať „tieňové“. Tieňová zákazka zákazníka A bude rovnaká ako pôvodná zákazka zákazníka A, ak je v dobe jeho odchodu salón prázdny. Ak tomu tak nie je, koniec tieňovej zákazky zákazníka A bude čas, kedy zo salónu odíde posledný zákazník, ktorý pral prádlo zároveň so zákazníkom A.

Riešenie úlohy s pôvodnými zákazkami a podmienkou, že žiadny zákazník v dobe, keď bude prať, neuvidí dvoch rôznych zákazníkov používať tú istú práčku, je rovnaké ako riešenie úlohy s tieňovými zákazkami bez tejto podmienky. Tieňové zákazky držia nejaké práčky „obsadené“ až do doby, kým ich môže Borivoj znovu použiť. K nájdeniu riešenia úlohy s tieňovými zákazkami sa potom dá použiť riešenie domáceho kola.

Pri riešení úlohy budeme postupovať tak, že zo zadaných zákaziek vytvoríme tieňové. Takto vytvorenú úlohu potom vyriešime ako v domácom kole. Tieňové zákazky vytvoríme nasledovným postupom. Ako v domácom kole vytvoríme udalosti príchodu a odchodu pre každého zákazníka. Tieto udalosti zotriedime (udalosti odchodu pred príchodmi, ktoré nastanú v rovnaký čas). Zotriedené udalosti raz prejdeme (podľa vzrastajúceho času) a budeme si udržovať dobu T, kedy zo salónu odíde posledný zákazník, ktorý je v salóne práve prítomný. Dobu T vieme ľahko udržovať tak, že kedykoľvek narazíme na príchod nejakého zákazníka, novú hodnotu T zistíme ako maximum jej minulej hodnoty a doby, kedy chce spracovávaný zákazník zo salónu odísť.

Prechod zotriedených udalostí bude prebiehať tak, že keď narazíme na

- *príchod* zákazníka A, upravíme dobu T.
- *odchod* zákazníka A, vytvoríme tieňovú zákazku pre zákazníka A, ktorej príchod nastavíme na príchod zákazníka A a odchod na dobu T (ktorá je v tej chvíli väčšia alebo rovná dobe, kedy chcel zo salónu odísť posledný zákazník A).

Tento prechod sa dá uskutočniť v lineárnom čase, avšak triedenie udalostí nám zaberie čas $O(N \log N)$. Na uloženie zadaných zákaziek, tieňových zákaziek a zotriedených udalostí budeme potrebovať $O(N)$ pamäti. Druhá časť riešenia, algoritmus z domáceho kola, má rovnakú časovú zložitosť ako práve popísané vytvorenie tieňových zákaziek, a tak je celková časová zložitosť nášho riešenia $O(N \log N)$ a pamäťová $O(N)$.

Listing programu:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_N 100

struct zakazka {
  long z,t;             /* začiatok a doba trvania */
};

struct udalost {
  long t;               /* čas udalosti */
  char prichod;         /* príchod alebo odchod */
  int zakaznik;         /* číslo zákazníka tejto udalosti */
};

/* porovnanie udalostí: triedi sa podľa času a v prípade rovnosti času najskôr odchod */
int udalost_cmp(const void *e1,const void *e2) {
  if (((struct udalost *)e1)->t == ((struct udalost *)e2)->t)
    return ((struct udalost *)e1)->prichod - ((struct udalost *)e2)->prichod;
  return ((struct udalost *)e1)->t - ((struct udalost *)e2)->t;
}

int N;
struct zakazka zakazky[MAX_N];
int stack[MAX_N];
int stack_top=0;        /* index prvého neobsadeného miesta v stacku */

int priradenie_praciek[MAX_N];
int pocet_praciek=0;

struct udalost udalosti[2*MAX_N];

void uprav_udalosti(void) {
  struct udalost prichody[MAX_N+1];
  struct udalost odchody[MAX_N+1];
  int prichody_len=0,odchody_len=0,i;
  long max_odchod=-1;

  /* zarážka pre zlievanie */
  prichody[prichody_len++].t=-1;
  odchody[odchody_len++].t=-1;

  /* vytvorenie príchodov a odchodov */
  for (i=0;i<2*N;i++)
    if (udalosti[i].prichod) {
      int odchod=zakazky[udalosti[i].zakaznik].z + zakazky[udalosti[i].zakaznik].t;
      if (odchod > max_odchod) max_odchod=odchod;

      prichody[prichody_len++]=udalosti[i];
    } else /* odchod */ {
      odchody[odchody_len]=udalosti[i];
```

```
      odchody[odchody_len++].t=max_odchod;
    }

  /* ešte zliať príchody a odchody do udalostí */
  prichody_len--; /* aby ukazoval nie za platný príchod, ale naňho */
  odchody_len--;  /* to isté s odchodmi */
  i=2*N-1;        /* index na koniec poľa udalostí, plníme ho zozadu */
  while (prichody[prichody_len].t!=-1 || odchody[odchody_len].t!=-1)
    if (prichody[prichody_len].t >= odchody[odchody_len].t) /* pridaj príchod */
      udalosti[i--]=prichody[prichody_len--];
    else /* pridaj odchod */
      udalosti[i--]=odchody[odchody_len--];
}

int main(void) {
  int i;

  /* Načítanie dát */
  printf("Zadajte počet zákazníkov:");
  scanf("%d",&N);
  for (i=0;i<N;i++) {
    printf("Doba príchodu a čas prania %d. zákazníka:",i+1);
    scanf("%ld %ld",&zakazky[i].z,&zakazky[i].t);
  }

  /* Vytvorenie udalostí */
  for (i=0;i<N;i++) {
    /* príchod */
    udalosti[2*i].t=zakazky[i].z;
    udalosti[2*i].prichod=1;
    udalosti[2*i].zakaznik=i;
    /* odchod */
    udalosti[2*i+1].t=zakazky[i].z+zakazky[i].t;
    udalosti[2*i+1].prichod=0;
    udalosti[2*i+1].zakaznik=i;
  }

  qsort(udalosti,2*N,sizeof(struct udalost),udalost_cmp);

  /* zmena oproti domácemu kolu */
  uprav_udalosti();

  /* simulácia provozu */
  for (i=0;i<2*N;i++)
    if (udalosti[i].prichod) {
      int pracka;
      if (!stack_top) /* nová pračka */ pracka=++pocet_praciek;
      else pracka=stack[--stack_top];
      priradenie_praciek[udalosti[i].zakaznik]=pracka;
    } else /* odchod */ {
      stack[stack_top++]=priradenie_praciek[udalosti[i].zakaznik];
    }

  /* vypísanie dát */
  printf("Je treba aspoň %d pračiek.\n",pocet_praciek);
  for (i=0;i<N;i++) printf("Zákazník %d pôjde k práčke %d.\n",i+1,priradenie_praciek[i]);

  return 0;
}
```

P-II-2

Najskôr zavedieme substitúciu $x_i = y_i + a_{i,N+1}$ pre i, $1 \leq i \leq N$, a $x_{n+1} = y_{N+1}$. Napríklad pre sústavu z prvého príkladu v zadaní úlohy po substitúcii dostaneme:

$$\begin{aligned} y_1 - y_2 &\neq 0 \\ y_1 - y_3 &\neq 0 \\ y_1 - y_4 &\neq 0 \\ y_2 - y_3 &\neq 1 \\ y_2 - y_3 &\neq 0 \\ y_3 - y_4 &\neq 0 \end{aligned}$$

Pre sústavu z druhého príkladu po substitúcii dostaneme:

$$\begin{aligned} y_1 - y_2 &\neq 2 = 0 \pmod 2 \\ y_1 - y_3 &\neq 0 \\ y_2 - y_3 &\neq 0 \end{aligned}$$

Novozískaná sústava podmienok má riešenie práve vtedy, keď ho mala tá pôvodná: Keď x_i, $1 \leq i \leq N+1$, rieši pôvodnú sústavu, potom riešenie novej sústavy podmienok je $y_i = x_i - a_{i,N+1}$ pre $i \leq N$ a $y_{N+1} = x_{N+1}$; naopak, z riešenia novej sústavy získame riešenie pôvodnej sústavy pričítaním hodnôt $a_{i,N+1}$.

Všimnime si, že všetky podmienky, v ktorých sa vyskytuje y_{N+1}, majú pravú stranu 0:

$$y_i - y_{N+1} = (x_i - a_{i,N+1}) - x_{N+1} = (x_i - x_{N+1}) - a_{i,N+1} \neq a_{i,N+1} - a_{i,N+1} = 0$$

Pozrime sa bližšie na novozískanú sústavu podmienok. Ak sú všetky pravé strany rovné nule, znamená to, že všetky neznáme $y_1, \ldots, y_{N+1}$ musia byť navzájom rôzne. Pretože ale existuje iba N čísel s rôznymi zvyškami po delení N, sústava podmienok nemá riešenie. Vzápätí ukážeme, že ak nie sú všetky pravé strany rovné nule, potom nová sústava podmienok, a teda aj tá pôvodná, majú riešenie.

Označme $a'_{i,j}$ pravú stranu podmienky, ktorej ľavá strana je $y_i - y_j$. Predpokladajme, že $a'_{i,j} \neq 0$ pre nejaké i a j. Pretože $a'_{i,N+1} = 0$, z voľby y_i musí platiť $1 \leq i < j \leq N$. Položme $y_i = y_j = 0$. Pretože $a'_{i,j} \neq 0$, podmienka $y_i - y_j \neq a'_{i,j}$ je tým zjavne splnená. Teraz zvolíme hodnoty ostatných neznámych postupne od y_1 po y_N tak, aby všetky podmienky, ktoré ich obsahujú, boli splnené. V okamihu, keď volíme hodnotu neznámej y_k, $1 \leq k \leq N$, $k \neq i, j$, tak sa neznáma y_k vyskytuje v najviac $N-1$ podmienkach s ostatnými neznámymi, ktorých hodnotu sme už zvolili. Pretože každá podmienka zakazuje priradenie práve jedného z čísel $0, \ldots, N-1$ neznámej y_k, je spolu zakázaných najviac $N-1$ hodnôt a dá sa y_k nejaká hodnota priradiť.

Ostáva zvoliť hodnotu y_{N+1}. Po prevedení našej substitúcie boli pravé strany všetkých N podmienok, v ktorých sa y_{N+1} vyskytuje, rovné nule. Teda hodnota y_{N+1} musí byť rôzna od hodnôt $y_1, \ldots y_N$. Pretože $y_i = y_j$, majú neznáme $y_1, \ldots, y_N$ najviac $N-1$ rôznych hodnôt a preto sa dá y_{N+1} priradiť (aspoň) jedna z hodnôt $0, \ldots, N-1$.

Naše doterajšie úvahy vedú priamočiaro ku kvadratickému algoritmu, ktorý rieši zadanú úlohu. Najskôr prevedieme substitúciu popísanú v prvom odseku. Ak sú všetky pravé strany

po substitúcii rovné 0, nová i pôvodná sústava podmienok nemá riešenie. V opačnom prípade nájdeme riešenie novej sústavy podmienok postupom popísaným v predchádzajúcich dvoch odsekoch. Riešenie pôvodnej sústavy ľahko získame aplikáciou substitúcie inverznej k prvej substitúcii. Časová aj pamäťová zložitosť nášho algoritmu je kvadratická, tj. $O(N^2)$.

Listing programu:

```
program rozdiely;
const MAX=1000;
var A: array[1..MAX,1..MAX] of integer;{ pravé strany nerovností }
    N: integer;     { počet neznámych }
    subst: array[1..MAX+1] of integer;     { substitučný posun }
    y: array[1..MAX+1] of integer;     { riešenie novej sústavy }

procedure nacitaj;
  var i,j:integer;
  begin
    write('N = ');
    readln(N);
    for i:=1 to N+1 do
      for j:=i+1 to N+1 do
        begin
          write('a[',i,',',j,'] = ');
          readln(A[i][j]);
        end;
  end;

procedure substituuj;
  var i,j:integer;
  begin
    for i:=1 to N do
      subst[i]:=A[i][N+1];
    subst[N+1]:=0;
    for i:=1 to N+1 do
      for j:=i+1 to N+1 do
        A[i][j]:=(A[i][j]+subst[j]-subst[i]+N) mod N;
  end;

function spocitaj: boolean;
  var i,j,k,l:integer;  { pomocné premenné i, j a k ako v popise algoritmu }
      zakazane:array[0..MAX-1] of boolean;{ pole zakázaných hodnôt pre neznámu }
  begin
    i:=1;
    while i<N do
      begin
        j:=i+1;
        while j<=N do
          begin
            if A[i][j]<>0 then break;
            inc(j)
          end;
        if j<=N then break;
        inc(i)
      end;
    if i=N then
      begin
        spocitaj:=false;
        exit;
      end;
```

```
    y[i]:=0; y[j]:=0;
    for k:=1 to N do
      begin
        if (k=i) or (k=j) then continue;
        for l:=0 to N-1 do zakazane[l]:=false;
        for l:=1 to k-1 do zakazane[(y[l]-A[l][k]+N) mod N]:=true;
        if i>k then zakazane[(y[i]+A[k][i]) mod N]:=true;
        if j>k then zakazane[(y[j]+A[k][j]) mod N]:=true;
        l:=0;
        while zakazane[l] do inc(l);
        y[k]:=l
      end;
    for l:=0 to N-1 do zakazane[l]:=false;
    for l:=1 to N do zakazane[y[l]]:=true;
    l:=0;
    while zakazane[l] do inc(l);
    y[N+1]:=l;
    spocitaj:=true
  end;

procedure vypis;
  var i:integer;
  begin
    for i:=1 to N+1 do
      writeln('x[',i,'] = ',(y[i]+subst[i]) mod N);
  end;

begin
  nacitaj;
  substituuj;
  if spocitaj then vypis
              else writeln('Zadaná sústava nerovníc nemá riešenie.');
end.
```

P-II-3

1. riešenie:

Najskôr si ukážeme jednoduché riešenie pracujúce v čase $O(kn)$, založené na priehradkovom triedení (RadixSort). Ako také triedenie funguje? Keď chceme zotriediť m reťazcov $s_1, \ldots, s_m$ dĺžky l, najskôr ich zotriedime podľa posledného písmena, potom podľa predposledného (pričom ak sa predposledné písmeno zhoduje, zachováme poradie podľa posledného písmena) atď. až podľa prvého písmena. Triedenie podľa i-teho písmena (tomu budeme hovoriť jeden *prechod*) robíme tak, že si založíme priehradky indexované písmenami, jednotlivé reťazce (resp. ich čísla) rozmiestnime do priehradok podľa toho, aké je ich i-te písmeno a nakoniec priehradky prejdeme od najmenšieho písmena k najväčšiemu a reťazce z nich vyzbierame.

Každý prechod bude trvať $O(m)$ [čas závisí i na veľkosti abecedy, pretože musíme prejsť i prázdne priehradky, ale to pre nás bude konštanta, ktorá sa „schová do O-čka“], prechodov je l, takže triedením strávime spolu čas $O(lm)$.

Aby sme vyriešili našu úlohu, nájdeme v zadanom reťazci všetky podreťazce dĺžky k (tých je $n - k + 1$), zotriedime ich RadixSortom a potom v zotriedenom zozname nájdeme najdlhší úsek tvorený rovnakými podreťazcami. Všimnime si, že pri triedení podreťazcov si nemusíme

pamätať celé podreťazce, ale stačia ich začiatky vo „veľkom“ reťazci. Takto všetko zvládneme v čase $O(kn)$ a pamäti $O(n)$.

Ešte jedna poznámka k implementácii: aby sme zbytočne neplytvali pamäťou, neukladáme jednotlivé priehradky ako oddelené polia, ale všimneme si, že všetky priehradky spolu obsahujú len n prvkov, takže ich naskladáme do jedného n-prvkového poľa, len si v druhom pamätáme, kde priehradka začína. Z vyzbieraných hodnôt z priehradok sa potom stane len skopírovanie jedného poľa do druhého.

Listing programu:

```
program Redundancia1;
const max = 100;
var T : string[max];                    { Zadaný reťazec }
    k : integer;                        { Ako dlhé podreťazce hľadáme }
    n : integer;                        { Koľko existuje podreťazcov dĺžky k }
    a, b : array [1..max] of integer;   { Pole, ktoré triedime; pole s priehradkami }
    p : array [char] of integer;        { Veľkosti / začiatky priehradok }
    i, j, l, maxcnt, maxpos : integer;  { Pomocné premenné }
    c : char;

begin
  readln (T);                           { Načítame vstup }
  readln (k);
  n := length (T) - k + 1;
  for i := 1 to n do a[i] := i;         { Všetky možné pozície podreťazcov }

  for j := k-1 downto 0 do begin        { RadixSort; podľa j-teho písmena }
    for c := #0 to # 255 do             { Spočítame veľkosti priehradok }
      p[c] := 0;
    for i := 1 to n do
      inc (p[T[a[i]+j]]);
    l := 1;                             { Spočítame ich začiatky }
    for c := #0 to #255 do begin
      l := l + p[c];
      p[c] := l - p[c];
    end;
    for i := 1 to n do begin            { Rozdelíme do priehradok }
      c := T[a[i]+j];
      b[p[c]] := a[i];
      inc (p[c]);
    end;
    for i := 1 to n do a[i] := b[i];    { A opäť z nich vyberieme }
  end;

  i := 1;                               { Hľadáme najdlhší úsek }
  maxcnt := 0;
  while i <= n do begin
    j := i;
    while (i <= n) and (copy (T, a[i], k) = copy (T, a[j], k)) do inc (i);
    if i - j > maxcnt then begin
      maxcnt := i - j;
      maxpos := a[j];
    end;
  end;

  writeln (copy (T, maxpos, k));
  writeln (maxcnt);
end.
```

2. riešenie:

Existuje algoritmus, ktorý zadanú úlohu rieši v čase $O(n)$, kde n je dĺžka textu T, je však pomerne komplikovaný – je potrebné vybudovanie a prechod tzv. sufixového stromu pre text T. Ak by sme sa uspokojili s časom $O(n \log n)$, vystačíme si aj s jednoduchšou dátovou štruktúrou, ktorou je sufixové pole. Obe tieto riešenia však rozhodne presahujú rámec tejto súťaže a neočakávali sme, že spomenuté dátové štruktúry viete použiť, tobôž že ich v priebehu súťaže vymyslíte.

Aj nasledujúce riešenie je dosť trikové a využíva netriviálne výsledky. Opäť, nepredpokladáme samozrejme, že by ste tieto techniky mali poznať, či dokonca aktívne používať – všetky tieto riešenia už uvádzame skôr pre zaujímavosť a prípadne ako inšpiráciu pre záujemcov o tento obor.

3. riešenie:

Podrobnejšie si ukážeme pomerne jednoduchý *randomizovaný* algoritmus, t.j. algoritmus používajúci pri výpočte náhodné čísla, ktorý bude pracovať v čase $O(n)$ aspoň v priemernom prípade. Tým je myslené, že ak budeme mať smolu na to, aké čísla nám padajú z generátoru náhodných čísel, môže to trvať dlhšie, ale vo väčšine prípadov nám (nezávisle na vstupných dátach) dá výsledok v čase $O(n)$.

Náš randomizovaný algoritmus bude fungovať tak, že najprv riešenie „uhádne“, overí si, že uhádnuté riešenie je naozaj správne a ak nebude, celý postup zopakuje. Takýto algoritmus, samozrejme, nemusí nikdy skončiť, ale my si dokážeme, že uhádnuté riešenie bude správne s pravdepodobnosťou aspoň $1/2$, takže v priemernom prípade budeme potrebovať najviac dva pokusy. Ako hľadanie, tak overovanie budú zaberať čas $O(n)$, takže túto časovú zložitosť bude mať v priemere i celý algoritmus.

Najprv si rozmyslime, že ak uhádneme, že sa nejaký reťazec dĺžky k opakuje v texte l-krát, môžeme si ľahko overiť, že tomu tak skutočne je. K tomu použijeme riešenie úlohy z domáceho kola – zostrojíme si vyhľadávací automat pre tento podreťazec, prejdeme s ním text a spočítame počet jeho výskytov, tj. počet prechodov akceptačným stavom. To nám zaberie čas $O(n)$ – v čase $O(k)$ zostrojíme automat, v čase $O(n)$ prejdeme text automatom a $k < n$.

Fáza hľadania funguje takto: Najskôr si zvolíme hashovaciu funkciu. To je nejaká funkcia h, zobrazujúca reťazce dĺžky k na celé čísla medzi 0 a $p-1$ (hodnotu p si zvolíme neskôr). Samozrejme sa nám môže stať, že dva reťazce zobrazíme na rovnaké číslo (tomu hovoríme kolízia), avšak ak bude hashovacia funkcia dobrá, nebude sa to stávať často. Teraz každý podreťazec textu T dĺžky k zobrazíme touto funkciou a spočítame počet reťazcov, ktoré sa zobrazia na jedno číslo. Z týchto počtov si vezmeme ten najväčší a vrátime jeden z prípadne viacerých reťazcov, ktoré sa na príslušné číslo zobrazili. Ak tento reťazec nekolidoval so žiadnym iným, vyhrali sme, pretože všetky ostatné reťazce (aj keď sme niektoré kvôli kolíziám nedokázali od seba odlíšiť) nie sú častejšie. Ak kolidoval, odhalí to kontrola.

Ostáva domyslieť detaily tak, aby sme všetko zvládli v lineárnom čase:

Nájdenie najčastejšej hodnoty hashovacej funkcie: Poslúži nám opäť RadixSort: každé číslo si zapíšeme v tvare $a_3n^3+a_2n^2+a_1n+a_0$, kde a_i sú menšie ako n (to nie je nič iné ako zápis čísla v sústave so základom n) a budeme ho triediť ako reťazec $a_3a_2a_1a_0$ nad n-prvkovou abecedou. Ako už vieme, RadixSort takéto triedenie zvládne v čase $O(n)$.

Počítanie hashovacej funkcie musíme urobiť šikovne (v skutočnosti toto je hlavný trik celého riešenia, zbytok sú len technické detaily), inak by sme tu potrebovali čas nk alebo väčší. Trik je v tom, že si zvolíme takú hashovaciu funkciu, aby sme (pre slovo w dĺžky $k-1$, a písmená s a t) dokázali $h(ws)$ spočítať so znalosťou $h(tw)$ v konštantnom čase. Potom ak budeme hashovacie funkcie počítať postupne od začiatku a posúvať sa vždy o jedno písmeno, spotrebujeme skutočne čas len $O(n)$.

Hashovaciu funkciu si zvolíme takto: p bude náhodne zvolené prvočíslo medzi $kn^3/2$ a kn^3,

$$h(s_k s_{k-1} \ldots s_1) = (\sum_{i=1}^{k} 256^{i-1} s_i) \bmod p$$

(predpokladáme, že abeceda [alebo skôr ASCIIceda] má 256 znakov).

Výraz v zátvorke si označíme $X(s)$. $X(s)$, samozrejme, môže byť väčšie ako je rozsah čísla reprezentovaného v počítači, avšak modulovanie p sa dá pri jeho výpočte vykonávať priebežne, takže nemôže pretiecť. Postupné počítanie funkcie h je potom jednoduché, lebo zrejme $h(ws) = (256 \cdot h(tw) - (256^k \bmod p)t + s) \bmod p$.

Ľahko nahliadneme, že pravdepodobnosť, že dôjde ku kolízii, je menšia ako 1/2: aby dvom rôznym reťazcom s_1 a s_2 bola priradená rovnaká hodnota, musel by rozdiel $X(s_1) - X(s_2)$ byť deliteľný p. Veľkosť tohto rozdielu je najviac 256^k, teda ho delí najviac $8 \log_2 k$ rôznych prvočísel (každé z nich má veľkosť aspoň 2 a keby ich bolo viac, ich súčin by musel byť väčší ako 256^k). Rôznych reťazcov je najviac n, teda ich dvojíc je najviac $n^2/2$ a „zlých“ prvočísel najviac $4kn^2$. Pomerne triviálny výsledok teórie čísel hovorí, že počet prvočísel medzi $kn^3/2$ a kn^3 je približne $\frac{kn^3}{2 \log kn^3}$, teda pravdepodobnosť, že sa trafíme do zlého prvočísla je menšia ako $\frac{8 \log kn^3}{n} \leq \frac{8 \log n^4}{n} = \frac{32 \log n}{n}$, čo je menej ako 1/2 pre $n \geq 381$ – v skutočnosti sú uvedené odhady pomerne hrubé, takže táto pravdepodobnosť je ešte podstatne menšia.

Ostáva jediný technický detail – ako nájsť náhodné prvočíslo. To sa dá spraviť v čase $O(\log^d n)$, kde d je nejaká konštanta, avšak nie je to úplne triviálne. Miesto toho v programe volíme iba náhodné číslo v danom intervale; do prvočísla sa trafíme s pravdepodobnosťou približne $1/\log n$, teda časová zložitosť sa tým zhorší najviac na $O(n \log n)$.

Listing programu:

```
program Redundancia2;
const maxN = 1000;
type pole = array[0..maxN] of longint;
var t : string;
    k, l, nRep1, nRep2, idx, i : integer;
    p, v256naKmodP, highP, lowP : longint;

{ Spočíta počet výskytov reťazca začínajúceho na pozícii idx a uloží túto hodnotu do nRep }
procedure Verify (idx : integer; var nRep : integer);
var back : pole;
    f, i : integer;
begin
  { spočíta spätnú funkciu pre reťazec od pozície idx }
  f := 0;
  back[1] := 0;
  for i := 2 to k do
    begin
      while true do
        begin
          if t[idx + f] = t[idx + i - 1] then
            begin
```

```
            inc (f);
            break;
          end;
        if f = 0 then
          break;
        f := back[f];
      end;
    back[i] := f;
  end;

  { spočíta počet výskytov reťazca }
  f := 0;
  nRep := 0;
  for i := 1 to l do
    begin
      while true do
        begin
          if t[idx + f] = t[i] then
            begin
              inc (f);
              break;
            end;
          if f = 0 then
            break;
          f := back[f];
        end;
      if f = k then
        begin
          inc (nRep);
          f := back[f];
        end;
    end;
end;

{ Spočíta hashovaciu funkciu pre začiatok t }
function HashBegin : longint;
var i : integer;
    h : longint;
begin
  h := 0;

  for i := 1 to k do
    h := (256 * h + ord (t[i])) mod p;
  HashBegin := h;
end;

{ Posunie hashovaciu funkciu h na pozíciu idx o 1 písmeno }
function HashShift (h : longint; idx : integer) : longint;
var minus : longint;
begin
  h := (h * 256) mod p;
  minus := (v256naKmodP * ord (t[idx])) mod p;
  HashShift := (h + p - minus + ord (t[idx+k])) mod p;
end;

{ Vyberie z a n-tú číslicu }
function digit (a : longint; n : integer) : integer;
var i : integer;
begin
```

```
  for i := 1 to n do
    a := a div l;
  digit := a mod l;
end;

{ Zotriedi pole a dĺžky m podľa n-tej číslice a výsledok uloží do sorted }
procedure SortByDigit (var a : pole; m, n : integer; var sorted : pole);
var nReps : pole;
    i, d : integer;
begin
  { spočíta počty opakovaní }
  for i := 0 to 2*l do
    nReps[i] := 0;
  for i := 1 to m do
    inc (nReps[digit (a[i], n) + 1]);

  { spočíta začiatky úsekov prvkov }
  for i := 1 to 2*l do
    nReps[i] := nReps[i - 1] + nReps[i];

  { zapíše prvky na správne pozície }
  for i := 1 to m do
    begin
      d := digit (a[i], n);
      inc (nReps[d]);
      sorted[nReps[d]] := a[i];
    end;
end;

{ Zotriedi pole a dĺžky m a výsledok uloží do sorted }
procedure RadixSort (var a : pole; m : integer; var sorted : pole);
var temp : pole;
begin
  SortByDigit (a, m, 0, temp);
  SortByDigit (temp, m, 1, sorted);
  SortByDigit (sorted, m, 2, temp);
  SortByDigit (temp, m, 1, sorted);
end;

{ Nájde v poli a dĺžky m číslo, ktoré sa v ňom najčastejšie opakuje,
  počet opakovaní dá do nRep, číslo do v }
procedure MostCommon (var a : pole; m : integer; var v : longint; var nRep : integer);
var sorted : pole;
    i, c : integer;
begin
  RadixSort (a, m, sorted);
  v := sorted[1];
  c := 1;
  nRep := 1;
  for i := 2 to m do
    begin
      if sorted[i] = sorted[i - 1] then
        inc(c)
      else
        c := 1;
      if c > nRep then
        begin
          v := sorted[i];
          nRep := c;
```

```
        end;
    end;
end;

{ "Uhádne" podreťazec, ktorý sa v T vyskytuje najčastejšie.  Počet výskytov uloží do nRep,
  začiatok prvného výskytu do idx }
procedure Guess (var idx : integer; var nRep : integer);
var hashVals : pole;
    i : integer;
    v : longint;
begin
  hashVals[1] := hashBegin;
  for i := 1 to l - k do
    hashVals[i + 1] := HashShift (hashVals[i], i);

  MostCommon (hashVals, l - k + 1, v, nRep);
  for i := 1 to l - k + 1 do
    if hashVals[i] = v then
      begin
        idx := i;
        break;
      end;
end;

{ Hlavný program }
begin
  readln (k);
  readln (t);
  l := length (t);
  highP := l * l * l * k;
  lowP := highP div 2;

  repeat
    p := random (highP - lowP) + lowP;
    v256naKmodP := 1;
    for i := 1 to k do
      v256naKmodP := (256 * v256naKmodP) mod p;
    Guess (idx, nRep1);
    Verify (idx, nRep2);
  until nRep1 = nRep2;

  writeln (nRep1);
  for i := idx to idx + k - 1 do
    write (t[i]);
  writeln;
end.
```

P-II-4

Túto úlohu by sme mohli riešiť podobne ako úlohy z minulého kola v čase $O(\log N)$ s $O(N)$-bitovými číslami – stačí si všimnúť, že otočenie čísla dĺžky N sa dá spraviť prehodením jeho polovíc (čo zvládneme na konštantný počet operácii) a následným otočením oboch polovíc (čo môžeme spraviť súčasne). Pre $N = 8$ by to vyzeralo takto:

	$x = x_7x_6x_5x_4x_3x_2x_1x_0$
$a := x \wedge \mathbf{11110000}$	$a = x_7x_6x_5x_4\,\mathbf{0\ 0\ 0\ 0}$
$b := x \wedge \mathbf{00001111}$	$b = \mathbf{0\ 0\ 0\ 0}\,x_3x_2x_1x_0$
$y := (a \gg 4) \vee (b \ll 4)$	$y = x_3x_2x_1x_0x_7x_6x_5x_4$
$a := y \wedge \mathbf{11001100}$	$a = x_3x_2\,\mathbf{0\ 0}\,x_7x_6\,\mathbf{0\ 0}$
$b := y \wedge \mathbf{00110011}$	$b = \mathbf{0\ 0}\,x_1x_0\,\mathbf{0\ 0}\,x_5x_4$
$y := (a \gg 2) \vee (b \ll 2)$	$y = x_1x_0x_3x_2x_5x_4x_7x_6$
$a := y \wedge \mathbf{10101010}$	$a = x_1\,\mathbf{0}\,x_3\,\mathbf{0}\,x_5\,\mathbf{0}\,x_7\,\mathbf{0}$
$b := y \wedge \mathbf{01010101}$	$b = \mathbf{0}\,x_0\,\mathbf{0}\,x_2\,\mathbf{0}\,x_4\,\mathbf{0}\,x_6$
$y := (a \gg 1) \vee (b \ll 1)$	$y = x_0x_1x_2x_3x_4x_5x_6x_7$

Keď však využijeme násobenie a delenie, zvládneme to na konštantný počet operácii, aj keď budeme potrebovať čísla dĺžky $O(N^2)$.

Náš algoritmus bude založený na tom, že zadané číslo $x = x_{N-1} \ldots x_0$ najprv N-krát skopírujeme (to sa dá spraviť pomocou násobenia), potom i-tu kópiu nahradíme číslom, ktoré bude na i-tom mieste obsahovať x_{N-1-i} a všade inde nuly. Následne všetky kópie sčítame (k tomu môžeme opäť použíť násobenie, ale my si ukážeme pekný trik so zvyškom po delení).

Kopírovanie bude vyzerať takto: číslo x vynásobíme číslom $(\mathbf{0}^{N-1}\mathbf{1})^{N+1}$, čím získame číslo $(x_{N-1} \ldots x_0)^{N+1}$, teda $N+1$ kópii zadaného čísla. Tento výsledok si tiež môžeme predstaviť rozdelený na N úsekov dĺžky $N+1$: najnižší úsek bude obsahovať čísla $x_0x_{N-1} \ldots x_0$, ten nad ním $x_1x_0x_{N-1} \ldots x_1$ atď. Teda i-ty úsek zdola (počítané od nuly) bude mať na najnižšom mieste x_i.

Každý úsek vyplníme hodnotou jeho najnižšieho bitu. K tomu stačí úsek upraviť do tvaru $\mathbf{10}^{N-1}x_i$ (použijeme $\wedge$ a $\vee$) a odčítať od neho jednotku. Ak x_i bolo $\mathbf{1}$, výjde $\mathbf{10}^N$, inak dôjde k prenosu a dostaneme $\mathbf{01}^N$. V každom prípade sa na najvyššom mieste objaví x_i a na ostatných $\neg\, x_i$, čo jedným *xor*om opravíme na x_i všade. Naviac nemohlo dôjsť k prenosu mimo úsek, takže sa úseky navzájom neovplyvňujú, a preto môžeme túto operáciu spraviť pre všetky úseky súčasne.

Teraz stačí vhodným *and*ovaním v každom úseku ponechať x_i na pozícii, na ktorej má vo výsledku skončiť, ostatné výskyty x_i vynulovať a všetky úseky sčítať. Sčítanie môžeme spraviť pekným trikom: ak máme v registri r uložené číslo tvaru $t_1 \ldots t_z$, kde t_i sú k-bitové čísla a sčítame $r \,\%(2^k - 1)$, vyjde $(t_1 + \ldots + t_z) \,\%(2^k - 1)$. V našom prípade naviac vieme, že $t_1 + \ldots + t_z < 2^k - 1$, pretože sčítame $(N+1)$-bitové úseky s N-bitovým výsledkom a tak sa modulo na výsledku neprejaví a dostaneme priamo hľadaný súčet.

Prečo modulo $2^k - 1$ funguje ako súčet blokov môžeme vidieť z toho, ako by sa choval „školský“ algoritmus pre delenie čísel na papieri, ale tiež to môžeme ľahko dokázať indukciou: pre $z = 1$ trik určite funguje; keď už vieme, že funguje pre $z - 1$ a chceme dokázať, že funguje i pre z, všimneme si, že:

$$t_1 \ldots t_z \,\%(2^k - 1) = \Big((t_1 \ldots t_{z-1}) \cdot 2^k + t_z\Big) \,\%(2^k - 1) =$$

$$= \Big(\underbrace{\Big(t_1 \ldots t_{z-1} \,\%(2^k - 1)\Big)}_{\text{indukčný predpoklad}} \cdot \underbrace{\Big(2^k \,\%(2^k - 1)\Big)}_{=1} + t_z \Big) \,\%(2^k - 1),$$

čo je spolu $(t_1 + \ldots + t_z) \,\%(2^k - 1)$, presne, ako sme chceli.

Na všetky výpočty sme potrebovali registre dĺžky $N \cdot (N+1) = O(N^2)$. Operácii bolo konštantne veľa. Ostáva už len program:

	$x = x_{N-1} \dots x_0$
$y := x * (\mathbf{0}^{N-1}\mathbf{1})^{N+1}$	$y = (x_{N-1} \dots x_0)^{N+1}$
	$= (x_{N-1} \dots x_0 x_{N-1}) \dots (x_1 x_0 x_{N-1} \dots x_1)(x_0 x_{N-1} \dots x_0)$
$y := y \wedge (\mathbf{0}^N\mathbf{1})^N$	$y = (\mathbf{0}^N x_{N-1}) \dots (\mathbf{0}^N x_1)(\mathbf{0}^N x_0)$
$y := y \vee (\mathbf{10}^N)^N$	$y = (\mathbf{10}^{N-1} x_{N-1}) \dots (\mathbf{10}^{N-1} x_1)(\mathbf{10}^{N-1} x_0)$
$y := y - (\mathbf{0}^N\mathbf{1})^N$	$y = (x_{N-1}(\neg\, x_{N-1})^N) \dots (x_0(\neg\, x_0)^N)$
$y := y \oplus (\mathbf{01}^N)^N$	$y = (x_{N-1}^{N+1}) \dots (x_0^{N+1})$
$y := y \wedge (\mathbf{0}^N\mathbf{1})(\mathbf{0}^{N-1}\mathbf{10}) \dots (\mathbf{010}^{N-1})$	$y = (\mathbf{0}^N x_{N-1})(\mathbf{0}^{N-1} x_{N-2}\mathbf{0}) \dots (\mathbf{00} x_1 \mathbf{0}^{N-2})(\mathbf{0} x_0 \mathbf{0}^{N-1})$
$y := y \,\%\, \mathbf{1}^{N+1}$	$y = \mathbf{0} x_0 x_1 \dots x_{N-1}$

SLOVENSKÁ KOMISIA MATEMATICKEJ OLYMPIÁDY

54. ROČNÍK MATEMATICKEJ OLYMPIÁDY

Riešenia 2. kola kategórie P

Vydala IUVENTA
pre vnútornú potrebu Ministerstva školstva SR
Sadzba programom LaTeX
Zodpovedný redaktor M. Forišek