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

MATEMATICKÁ OLYMPIÁDA NA STREDNÝCH ŠKOLÁCH

54. ročník, školský rok 2004/2005

Riešenia úloh 1. kola kategórie P

Tento pracovný materiál nie je určený priamo študentom — účastníkom olympiády. Má pomôcť učiteľom na školách pri príprave konzultácií a pracovných seminárov pre riešiteľov súťaže, členom krajských výborov MO slúži ako podklad pre opravovanie úloh domáceho kola MO kategórie P. **Študentom možno tieto komentáre poskytnúť až po termíne stanovenom pre odovzdanie riešení úloh domáceho kola MO kategórie P** ako informáciu, ako bolo treba úlohy správne riešiť a pre ich odbornú prípravu na účasť v krajskom kole súťaže.

P-I-1

Najskôr určíme potrebný počet práčok. Predstavme si, že celý deň prebehne a každý zákazník bude v práčovni práve počas doby, na ktorú chcel mať prenajatú práčku. Nech K je najväčší počet zákazníkov, ktorí boli naraz v práčovni. Zjavne potrebujeme aspoň K práčok, lebo každý z nich jednu potrebuje. Neskôr ukážeme, že K práčok aj stačí. Teraz sa ale zamyslime nad tým, ako určiť K.

Budeme simulovať príchody a odchody zákazníkov a zároveň si pamätať, koľko ich je v aktuálnom okamihu v práčovni. *Udalosťou* nazveme príchod alebo odchod zákazníka. Každý zákazník nám teda spôsobí dve udalosti. Takto získané udalosti utriedime podľa času, kedy nastanú. Následne ich v tomto poradí budeme spracúvať, pričom ak spracovaná udalosť je príchod, zvýšime si pamätaný počet zákazníkov v práčovni, ak je to odchod, počet zákazníkov znížime.

Ak v rovnakom čase niekto príde a zároveň niekto odíde, môžu použiť tú istú práčku. Preto udalosti, ktoré nastanú v tom istom čase, zoradíme tak, aby sme najskôr spracovali všetky odchody (uvoľnia sa práčky, ubudnú zákazníci v práčovni) a až potom príchody, ktoré v danom okamihu nastali.

Takto spočítame číslo K. Teraz ukážeme, že K práčok vieme naozaj priradiť zákazníkom, a teda že K je naozaj riešením našej úlohy. Znova spustíme simuláciu príchodov a odchodov zákazníkov, pričom si o každej práčke pamätáme, či je momentálne voľná alebo nie. Spracovanie odchodov je triviálne – uvoľníme príslušnú práčku. Všimnime si teraz ľubovoľný príchod zákazníka. V danom okamihu je v práčovni najviac $K-1$ iných zákazníkov (lebo on je najviac K-ty), a teda aspoň jedna práčka je voľná. Túto mu priradíme a zapamätáme si, že je už obsadená.

Zjavne takto každému zákazníkovi priradíme práčku, a teda K práčok stačí.

Ukážeme si ešte zopár trikov, ako vyššie uvedený algoritmus zrýchliť a zjednodušiť.

Budeme počas jednej simulácie udalostí počítať K aj priraďovať práčky zákazníkom. Začneme s práčovňou bez práčok (t.j. $K = 0$). Pri príchode zákazníka ho skúsime umiestniť k neobsadenej práčke. Ak sú všetky práčky obsadené, zvýšime K a umiestnime ho k novej (K-tej) práčke. Zjavne aktuálne K je najväčší počet zákazníkov, ktorí boli naraz v práčovni do daného okamihu. Preto na konci dostaneme správnu hodnotu K a zároveň správne priradenie práčok zákazníkom.

Aby sme vedeli rýchlo uvoľniť práčku, budeme si pre každého zákazníka v práčovni pamätať, ktorú práčku používa. Aby sme vedeli rýchlo priradiť voľnú práčku novému zákazníkovi, budeme si čísla voľných práčok pamätať napríklad v zásobníku. (Teda pamätáme si počet voľných práčok V a v prvých V políčkach nejakého poľa si pamätáme ich čísla.)

Zostáva určiť časovú a pamäťovú zložitosť výsledného algoritmu. Udalostí je $O(N)$, práčok tiež, všetko potrebné si teda vieme pamätať v lineárne veľkej pamäti. Utriediť udalosti vieme použitím nejakého klasického triediaceho algoritmu v čase $O(N \log N)$. Na spracovanie jednej udalosti nám stačí konštantný čas, preto prechod všetkými udalosťami zvládneme v čase $O(N)$. Výsledná časová zložitosť je teda $O(N \log N)$.

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_N 31000

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

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

// porovnanie udalostí: triedi sa podľa času a v prípade rovnosti najskôr odchody
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_pracok[MAX_N];
int pocet_pracok=0;

struct udalost udalosti[2*MAX_N];
FILE *fr,*fw;

int main(void)
{
  int i;

  fr=fopen("pracky.in","r");// načítanie údajov
  fscanf(fr,"%d",&N);
  for (i=0;i<N;i++) fscanf(fr,"%ld %ld",&zakazky[i].z,&zakazky[i].t);
```

```
  fclose(fr);

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

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

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

  fw=fopen("pracky.out","w");// výpis výstupu
  fprintf(fw,"%d\n",pocet_pracok);
  for (i=0;i<N;i++) fprintf(fw,"%d\n",priradenie_pracok[i]);
  fclose(fw);

  return 0;
}
```

P-I-2

Našou úlohou je zistiť pre danú permutáciu $a_1, \ldots, a_N$ počet takých dvojíc (a_i, a_j), kde $i < j$ a $a_i > a_j$ (teda väčšie číslo je uvedené pred menším). Takúto dvojicu voláme *inverzia*.

Ak by nám stačil čas kvadratický od N, jednoducho vyskúšame všetky dvojice prvkov a pre každú sa pozrieme, či sú dotyčné prvky v správnom poradí alebo nie. Ukážeme si myšlienky dvoch riešení, ktoré pracujú v čase $O(N \log N)$.

Prvé riešenie bude prvky permutácie spracúvať priebežne. Vždy, keď prečíta ďalší prvok permutácie, pripočíta k počtu inverzií tie, ktoré nám práve pribudli. Potrebujeme teda vedieť rýchlo povedať, koľko väčších čísel ako to práve prečítané sme už videli.

Jednou možnosťou je pamätať si počty prečítaných čísel ležiacich vo vhodne zvolených intervaloch. Predpokladajme pre jednoduchosť, že N je mocnina dvoch. (V opačnom prípade ho zväčšíme na najbližšiu väčšiu mocninu dvoch, časovú zložitosť nám to nepokazí.) Budeme si pamätať, koľko spomedzi doteraz prečítaných čísel ležalo v intervale $[N/2, \ldots, N-1]$, koľko v $[N/4, \ldots, N/2-1]$ a $[N/2, \ldots, 3N/4-1]$, a tak ďalej.

Na začiatku sú všetky pamätané počty samozrejme nulové. Keď prečítame nejaké číslo, pozrieme sa, či je v intervale $[0, \ldots, N/2 - 1]$. Ak áno, pripočítame k počtu inverzii počet dovtedy prečítaných čísel z intervalu $[N/2, \ldots, N - 1]$ a pokračujeme s testovaním na intervale $[0, \ldots, N/2 - 1]$. Ak nie, tak zvýšime počet čísel v intervale $[N/2, \ldots, N - 1]$ a pokračujeme s testovaním na ňom. Vo všeobecnosti sa na intervale pozrieme, či je práve prečítané číslo z jeho prvej polovice alebo nie. Ak áno, zväčšíme počet inverzii o počet prečítaných čísel z druhej polovice, ak nie, zväčšíme počet čísel v druhej polovici. Takto pokračujeme, kým sa nedostaneme k intervalu dĺžky 1.

Takto pre každé číslo spočítame počet nových inverzií a zároveň správne upravíme pamätané počty prečítaných čísel. Keďže pri každom kroku sa dĺžka uvažovaného intervalu zmenší na polovicu, krokov bude $\log_2 N$, a teda na spracovanie jedného čísla potrebujeme čas $O(\log N)$.

V **druhom riešení** upravíme algoritmus MergeSort tak, aby okrem utriedenia danej permutácie spočítal aj jej počet inverzií. Predstavme si teda, že vstupnú permutáciu rozdelíme na dve približne rovnaké časti. Každú z nich rekurzívnym volaním utriedime a zároveň v nej spočítame počet inverzii. Ostáva nám zarátať tie inverzie, pri ktorých je prvé číslo v prvej polovici a druhé v druhej. Tie spočítame pri spájaní oboch utriedených postupností do jednej.

Máme teda dve utriedené postupnosti a chceme ich spojiť do jednej. Samotné spájanie vyzerá tak, že zakaždým porovnáme ich prvé prvky a ten menší presunieme na koniec práve vytváranej výslednej postupnosti. Ako spočítať počet hľadaných inverzii? Vždy, keď sme vybrali číslo z druhej postupnosti, vieme, že je menšie od všetkých čísel prvej postupnosti, ktoré sme ešte nevybrali. S každým z nich tvorí inverziu.[1] Preto k počtu inverzii prirátame aktuálnu veľkosť prvej postupnosti.

Obe uvedené riešenia majú zjavne časovú zložitosť $O(N \log N)$ a pamäťovú zložitosť $O(N)$. Program je implementáciou druhého riešenia.

```
//
// pocet inverzii v postupnosti
//

#include <stdio.h>
#define MAX 1000000
int A[MAX],B[MAX];  //pole so vstupnymi udajmi

long long merge(int zac, int kon, int medzi){
 int a=zac,b=medzi+1,pis=a,i;
 long long pocl=medzi-zac+1,poc=0;

 while (a<=medzi || b<=kon){
   if ((A[a]<=A[b] && a<=medzi) || b>kon) {
     pocl--;              //na lavej strane pribudol novy prvok
     B[pis++]=A[a++];     //pridame k utriedenej postupnosti
   } else {
     poc+=pocl;
     B[pis++]=A[b++];
   }
  }
  //zapiseme merdznute udaje do povodneho pola
```

[1] Tu je vhodné si uvedomiť, že bez ohľadu na to, kde presne tie čísla pôvodne boli, určite tvorili inverziu aj v pôvodnej, neutriedenej permutácií. A naopak, žiadna takáto inverzia sa nám nemohla stratiť a všetky nájdeme.

```
  for (i=zac; i<=kon; i++)
    A[i]=B[i];

  return poc;
}

long long work(int zac, int kon){  //merge sort spojeny s ratanim inverzii
  long long poc=0;
  int medzi;

  if (zac>=kon) return 0;
  medzi=(zac+kon)/2;

  //split
  poc+=work(zac,medzi);     //pocet inverzii pravej casti
  poc+=work(medzi+1,kon);   //pocet inverzii lavej casti
  //merge
  poc+=merge(zac,kon,medzi);

  return poc;
}

int i,n;
int main(void){
  scanf("%d ",&n);
  for (i=0;i<N;i++) scanf("%d ",&A[i]);
  printf("%lld\n",work(0,n-1));
  return 0;
}
```

P-I-3

Najprv si zadefinujme niektoré pojmy a značenia. Pojmy *slovo* a *reťazec* budú v nasledujúcom texte znamenať to isté – konečnú postupnosť znakov z nejakej konečnej abecedy (v našom prípade '`A`', '`C`', '`G`' a '`T`'). Napríklad `AAA` a `GGAGCTG` sú reťazce. Špeciálny prípad reťazca je prázdny reťazec, t.j. taký, ktorý neobsahuje žiadne znaky. Budeme ho označovať λ. *Dĺžka* reťazca je počet jeho znakov. Dĺžku reťazca v budeme označovať $|v|$. Teda napríklad $|\texttt{AAA}| = 3$, $|\lambda| = 0$. Ak u a v sú reťazce, označíme uv ich *zreťazenie*. Teda pre $u = \texttt{AGG}$ a $v = \texttt{CCT}$ je $uv = \texttt{AGGCCT}$.

Počiatočnému úseku reťazca budeme vravieť jeho *prefix*. Teda slovo u je prefixom slova v, ak existuje reťazec w taký, že $uw = v$. Koncový úsek reťazca budeme nazývať jeho *sufix*. Každé slovo je svojim vlastným prefixom aj sufixom (pre $w = \lambda$). Prefix alebo sufix slova v je *netriviálny,* ak sa nerovná v.

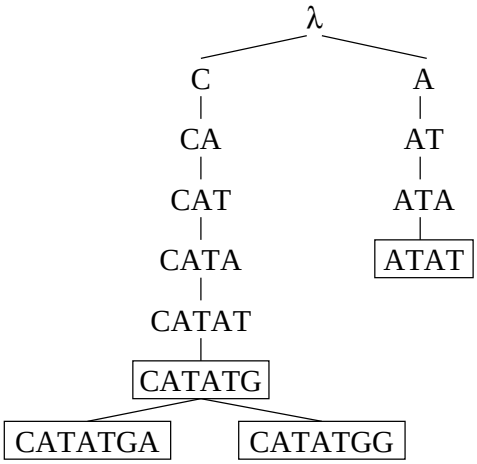
Podreťazec reťazca v je ľubovoľný súvislý úsek znakov z v – teda w je podreťazec v, ak w je sufixom nejakého prefixu slova v.

Teraz sa môžeme pustiť do riešenia úlohy. Zadané reťazce označme s_1, s_2, ..., s_n. S bude súčet ich dĺžok, teda $S = |s_1|+|s_2|+\ldots+|s_n|$. Zo slov $s_1, \ldots, s_n$ zostavíme *vyhľadávací automat.* Vyhľadávací automat je štruktúra, umožňujúca v lineárnom čase pre ľubovoľný reťazec t určiť

všetky slová s_i, ktoré sa vyskytujú ako podreťazec v t. Keď budeme toto vedieť spraviť, riešenie úlohy je jednoduché – tento postup vykonáme postupne pre $t = s_1$, $t = s_2$, ..., $t = s_n$, čím zistíme postupne predkov prvého, druhého, ..., n-tého organizmu. Časová zložitosť bude $O(S +$ čas na konštrukciu automatu + dĺžka výstupu$)$.

Zostáva vytvoriť taký automat. Vyhľadávací automat sa skladá z dvoch častí – *trie* postaveného z reťazcov $s_1, \ldots, s_n$ a takzvanej *spätnej funkcie.*

Popis trie začneme príkladom – toto je trie pre reťazce zo vzorového vstupu v zadaní:



Trie je strom, ktorého vrcholy zodpovedajú všetkým prefixom reťazcov $s_1, \ldots, s_n$. Ak má niekoľko reťazcov s_i rovnaký začiatok, zodpovedá tomuto spoločnému prefixu iba jeden vrchol. Synovia vrcholu zodpovedajúceho reťazcu w sú vrcholy zodpovedajúce reťazcom wx, kde x je znak abecedy, v našom prípade '`A`', '`C`', '`G`' alebo '`T`'. Každý vrchol má teda najviac štyroch synov, ale môže mať aj menej, ak žiadne zo slov $s_1, \ldots, s_n$ nezačíná na wx. Koreň stromu zodpovedá prázdnemu reťazcu.

Niektoré vrcholy trie zodpovedajú slovám z s_i – napríklad vrcholy, ktoré nemajú žiadnych synov, ale môžu to byť aj vnútorné vrcholy trie, ak je niektoré zo slov prefixom iného. Ostatné vrcholy budeme nazývať *pomocné.*

O slove budeme vravieť, že je *reprezentované* v trie, ak je to jedno zo slov $s_1, \ldots, s_n$, pre ktoré sme trie postavili. O slove budeme vravieť, že sa v trie *nachádza*, ak mu zodpovedá nejaký vrchol trie (môže byť aj pomocný). Každé slovo, ktoré je v trie reprezentované, sa v ňom samozrejme aj nachádza, opačné tvrdenie však vo všeobecnosti neplatí. Vo zvyšku textu budeme občas ztotožňovať vrcholy trie so slovami, ktoré im zodpovedajú. Ak napríklad budeme mať funkciu, ktorá vrcholu trie zodpovedajúcemu slovu v priradí vrchol zodpovedajúci inému slovu w, budeme občas pre zjednodušenie vravieť, že táto funkcia priraďuje slovu v slovo w.

V každom vrchole trie budeme mať smerníky na jeho štyroch synov. Niektorí synovia nemusia existovať, v tom prípade bude príslušný smerník `nil`. Okrem toho bude vrchol obsahovať hodnotu typu `boolean`, ktorá určuje, či vrchol zodpovedá nejakému slovu reprezentovanému v trie, alebo či je iba pomocný.

Či sa v trie nachádza určité slovo, môžeme jednoducho zistiť v čase lineárnom od dĺžky tohto slova: Začneme v koreni. Z neho sa posunieme do syna zodpovedajúceho prvému písmenu slova, z tohoto syna ďalej po hrane zodpovedajúcej druhému písmenu, atď. Ak narazíme na `nil` skôr, ako prídeme na koniec slova, toto slovo sa v trie nevyskytuje. Keď dôjdeme do vrcholu, ktorý zodpovedá zadanému slovu, môžeme ešte zistiť, či je toto slovo v trie reprezentované, t.j. či príznak vrcholu, do ktorého sme prišli, je `true`.

Podobne môžeme trie vytvoriť – postupujeme analogicky ako pri vyhľadávaní, ale keď „vypadneme“ z trie (t.j. narazíme na `nil`), začneme stavať novú cestu. Tento postup bude trvať $O(S)$ spolu pre všetky reťazce.

Ďalej si definujeme *spätnú funkciu* f, ktorá každému vrcholu trie priradí nejaký iný vrchol, zodpovedajúci kratšiemu slovu (preto spätná). Pre vrchol w bude $f(w)$ definované ako vrchol v taký, že v je najdlhší netriviálny sufix w nachádzajúci sa v trie. Funkciu môžeme reprezentovať tak, že pre každý vrchol v trie si uložíme smerník na vrchol $f(v)$.

Jednoduchý spôsob, ako spätnú funkciu spočítať, je tento: Vezmeme slovo w a zahodíme z neho prvé písmeno. Ak sa slovo v, ktoré takto dostaneme, nachádza v trie, je $f(w) = v$. Inak z v znovu zahodíme prvé písmeno a postup opakujeme, až kým hodnotu spätnej funkcie neurčíme – to sa určite stane, lebo v najhoršom prípade sa zastavíme na prázdnom reťazci. Pre vzorový vstup $f(\texttt{ATA}) = f(\texttt{CA}) = f(\texttt{CATATGA}) = \texttt{A}$, $f(\texttt{ATAT}) = f(\texttt{CAT}) = \texttt{AT}$, $f(\texttt{CATA}) = \texttt{ATA}$, $f(\texttt{CATAT}) = \texttt{ATAT}$, $f(\texttt{A}) = f(\texttt{C}) = f(\texttt{AT}) = f(\texttt{CATATG}) = f(\texttt{CATATGG}) = \lambda$.

Význam funkcie f je tento: Nech máme nejaký text a chceme zistiť, ktoré slová nachádzajúce sa v trie končia na zadanej pozícii v tomto texte. Navyše nech vieme, že s je najdlhšie také slovo. Potom $f(s)$ je druhé najdlhšie, $f(f(s))$ tretie najdlhšie, atď., dokážeme ich teda v lineárnom čase vypísať (a naviac zoradené podľa dĺžky). To sa nám bude hodiť, pretože pri hľadaní pomocou automatu si budeme pamätať pre každú pozíciu v texte vždy práve toto slovo s (presnejšie vrchol v trie, ktorý mu zodpovedá).

Vyššie popísaný triviálny postup ako spočítať f zaberie čas $O(S^3)$ – pre každý z najviac S vrcholov by sme museli vyhľadať rádovo S reťazcov, ktorých dĺžka môže byť až S. Samozrejme by sme to chceli zvládnuť rýchlejšie. K tomu použijeme postup založený na dynamickom programovaní. Funkciu f budeme postupne počítať od najkratších slov po dlhšie. Pre jednopísmenové slová je $f(x) = \lambda$. Uvažujme teraz slovo wx, kde x je jeho posledné písmeno. Označme $v = f(wx)$. Vieme, že v musí byť nejaký sufix wx, teda v končí na x (ak nie je prázdne), teda $v = v'x$ pre nejaké slovo v', ktoré je sufixom w. Teraz využijeme to, čo sme ukázali v predchádzajúcom odstavci – všetky sufixy w, ktoré sa nachádzajú v trie, sú $f(w)$, $f(f(w))$, atď. Nás zrejme zaujíma najdlhší z nich, ktorý sa dá rozšíriť o písmeno x tak, aby sa výsledné slovo nachádzalo v trie. Algoritmus na určenie $f(wx)$ teda funguje takto:

Označme $w' = f(w)$, túto hodnotu už máme spočítanú. Ak sa $w'x$ nachádza v trie, máme $f(wx) = w'x$, lebo w' je najdlhší sufix w v trie a teda $f(wx)$ nemôže byť dlhší. Ak sa $w'x$ v trie nenachádza, vyskúšame $w'' = f(w')$ a takto pokračujeme, až kým buď nenájdeme hodnotu pre $f(wx)$, alebo nedorazíme k prázdnemu reťazcu – potom $f(wx) = \lambda$. Testovanie, či $w'x$ je v trie, zvládneme v konštantnom čase, lebo poznáme vrchol v trie zodpovedajúci reťazcu w'.

Aká je časová zložitosť tohto postupu? Pri stanovovaní f pre jeden vrchol sa nám môže stať, že funkciu f budeme musieť použiť až S-krát. Na prvý pohľad by sa teda mohlo zdať, že časová zložitosť bude $O(S^2)$. Všimnime si však, že toto sa nám nemôže stať príliš často: slovo $f(wx)$ bude len o jedna dlhšie, než slovo $w''^{\ldots'}$, ku ktorému sme dospeli pri vracaní sa, teda oproti $f(w)$ môže byť nanajvýš o jedna dlhšie. Na to, aby sme sa mohli vracať o k hladín teda najprv musíme spraviť aspoň k krokov, v ktorých sa nevraciame vôbec a teda ich zvládneme v konštantnom čase. Táto úvaha nie je velkom presná, nie je však ťažké domyslieť všetky detaily. Spolu sa teda budeme vracať najviac S krát a časová zložitosť bude $O(S)$.

Vráťme sa ešte k motivácii definície funkcie f. Povedali sme, že $f(s)$, $f(f(s))$, atď. sú všetky sufixy slova s nachádzajúce sa v trie. Vôbec sme však nerozlišovali, či sú to pôvodne zadané slová, ktoré sú v trie reprezentované, alebo iba nejaké ich prefixy – teda pomocné vrcholy, ktoré žiadne z pôvodne zadaných slov nereprezentujú. Samozrejme niekde medzi nimi sú aj všetky slová, ktoré nás zaujímajú, ale mohlo by sa stať, že „balastu“ okolo bude veľa a jeho preskakovanie nám zhorší časovú zložitosť.

Preto ešte spočítame funkciu f', ktorá pre vrchol v vráti najdlhší reťazec u rôzny od v taký, že u je reprezentované v trie a dá sa k nemu dostať z v pomocou spätnej funkcie. Táto funkcia bude robiť presne to, čo chceme – $f'(v)$, $f'(f'(v))$, atď. sú práve všetky slová z pôvodnej množiny, končiace na danej pozícii. Spočítať f' pre všetky vrcholy trie je jednoduché – postupujeme

od najkratších slov a využijeme to, že $f'(v) = f(v)$ ak $f(v)$ je reprezentované v trie a $f'(v) = f'(f(v))$ inak. Tento výpočet vyžaduje čas $O(S)$.

Tým sme dokončili konštrukciu automatu. Zostáva ešte vysvetliť, ako takto získaný automat použiť. Ako sme povedali v úvode, automat nám umožňuje rýchlo nájsť pre ľubovoľný text všetky slová s_i, ktoré sú jeho podreťazcom. Označme prehľadávaný text t.

Pre každý prefix u zadaného textu (slova) t by sme chceli nájsť najdlhší sufix u, ktorý sa nachádza v trie (potom môžeme s použitím funkcie f' jednoducho nájsť všetky s_i, ktoré sú sufixom u). Tieto sufixy postupne spočítame pre prefix t dĺžky 0, 1, 2, atď. Označme $l(i)$ hľadaný sufix pre prefix dĺžky i.

Prefix t dĺžky 0 je prázdne slovo a teda $l(0) = \lambda$.

Predpokladajme, že sme už spočítali l pre dĺžku i a $l(i) = s$. Nech písmeno na $(i+1)$-vej pozícii v t je x. Potom $l(i+1)$ má byť najdlhší reťazec nachádzajúci sa v trie, ktorý sa vyskytuje ako podreťazec t končiaci na pozíci $i+1$, teda $l(i+1)$ musí končiť na x. $l(i+1)$ teda môžeme zapísať ako rx pre nejaký reťazec r, ktorý sa tiež nachádza v trie. Navyše r musí byť sufix s. Prejsť cez všetky sufixy s už vieme – stačí prezrieť s, $f(s)$, $f(f(s))$, atď. Mezi nimi hľadáme najdlhší, ktorý sa dá rozšíriť o x tak, aby sme zostali v trie. Všimnime si, že najdlhší taký sufix musí byť prvý, na ktorý narazíme.

Toto opakujeme, až kým nespracujeme celý reťazec t. Podobný postup sme použili pri konštrukcii spätnej funkcie, preto nás asi neprekvapí, že aj tu dosiahneme lineárnu časovú zložitosť: s sa posunie smerom od koreňa najviac $|t|$-krát (v každom kroku o jedna), teda smerom ku koreňu sa môžeme pohnúť tiež najviac $|t|$-krát. Celková časová zložitosť je teda $O(t)$.

Pre každý prefix u textu si naviac označíme slová zo zadania, ktoré sú jeho sufixy. Ako sme už uviedli, sú to práve slova $f'(s)$, $f'(f'(s))$, atď., plus prípadne slovo s, ak je jedným zo zadaných slov. Chceli by sme, aby nám na vypisovanie výstupu stačila časová zložitosť $O(\text{dĺžka výstupu})$. To by sa nám mohlo pokaziť, ak by sa nejaké slovo r v textu vyskytovalo veľakrát – potom totiž budeme veľakrát zbytočne prechádzať postupnost slov $f'(r)$, $f'(f'(r))$, ... To ľahko napravíme tak, že len čo narazíme na označené slovo, ďalej sa už funkciou f' vracať nebudeme – vieme totiž, že všetky ďalšie slová, ku ktorým by sme sa takto mohli dostať, sme už vypísali, keď sme tu boli prvý krát. Takto každé vypísané slovo navštívime práve raz, a teda dosiahneme požadovanú časovú zložitosť.

Celková zložitosť riešenia zadanej úlohy je $O(S + \text{dĺžka výstupu})$. To je zjavne optimálne, pretože minimálne musíme načítať vstup a vypísať výsledok. Pamäťová zložitosť je $O(S)$.

```
program Phylogenetics;

type pList = ^List;                    { typ pre zoznam organizmov }
     List =
       record
         code : string;
         organism : integer;
         next : pList;
       end;

type Base = (bA, bC, bG, bT);          { abeceda }
     pTrieNode = ^TrieNode;            { typ pre vrchol trie }
     TrieNode =
       record
         sons : array[Base] of pTrieNode;   { synovia zodpovedajú písmenám abecedy }
```

```
    back : pTrieNode;              { spätná funkcia}
    backInSet : pTrieNode;         { spätná funkcia iterovaná
                                     až k prvému vrcholu
                                     reprezentujúcemu organizmus }
    organism : integer;            { číslo reprezentovaného organizmus
                                     alebo 0 pre pomocný vrchol }
    seen : integer;                { ak už bol tento organizmus
                                     vypísaný pri prehľadávání jeho
                                     potomka, číslo tohto potomka }
    next : pTrieNode;              { nasledujúci vrchol trie vo fronte }
  end;


{ zmení znaky ACGT na hodnoty typu Base }
function BaseToNumber(ch : char) : Base;
begin
  case ch of
    'A' : BaseToNumber := bA;
    'C' : BaseToNumber := bC;
    'G' : BaseToNumber := bG;
    'T' : BaseToNumber := bT;
  end;
end;

{ zmení hodnotu typu base na znaky ACGT }
function BaseToChar(b : Base) : char;
begin
  case b of
    bA : BaseToChar := 'A';
    bC : BaseToChar := 'C';
    bG : BaseToChar := 'G';
    bT : BaseToChar := 'T';
  end;
end;

{ vytvorí nový vrchol trie }
function NewTrieNode : pTrieNode;
var ret : pTrieNode;
    i : Base;
begin
  new (ret);
  for i := bA to bT do
    ret^.sons[i] := nil;
  ret^.back := nil;
  ret^.backInSet := nil;
  ret^.organism := 0;
  ret^.seen := 0;
  NewTrieNode := ret;
end;
```

```
{ pridá suffix reťazca S od pozície P do trie ROOT }
procedure AddToTrie(var root : pTrieNode; var s : string; p, organism : integer);
begin
  if root = nil then
    root := NewTrieNode;

  if p > length (s) then
    begin
      { sme na konci reťazca }
      root^.organism := organism;
    end
  else
    begin
      { pridáme organizmus do vetve určenej aktuálnym znakom }
      AddToTrie (root^.sons[BaseToNumber (s[p])], s, p + 1, organism);
    end;
end;

{ spočíta spätnú funkciu pre vrchol v, ktorého otec je o
  a do v sa dostaneme hranou číslo c. }
procedure ComputeBack (v, o : pTrieNode; c : Base);
var last : pTrieNode;
begin
  repeat
    last := o;
    o := o^.back;
  until (o = nil) or (o^.sons[c] <> nil);

  if o = nil then
    v^.back := last
  else
    v^.back := o^.sons[c];

  if v^.back^.organism = 0 then
    v^.backInSet := v^.back^.backInSet
  else
    v^.backInSet := v^.back;
end;

{ spočíta spätnú funkciu pre trie s vrcholom r. }
procedure ComputeBackFunction (r : pTrieNode);
var queue : pTrieNode;
    queueEnd : pTrieNode;
    son : pTrieNode;
    c : Base;
begin
  r^.back := nil;
```

```
  queue := r;
  queue^.next := nil;
  queueEnd := queue;

  repeat
    for c := bA to bT do
      begin
        son := queue^.sons[c];

        if son <> nil then
          begin
            ComputeBack (son, queue, c);
            queueEnd^.next := son;
            queueEnd := son;
            queueEnd^.next := nil;
          end;
      end;
    queue := queue^.next;
  until queue = nil;
end;

{ vytvorí vyhľadávací automat zo zadaného zoznamu reťazcov }
function CreateAutomaton (l : pList) : pTrieNode;
var root, head : pTrieNode;
    i : integer;
begin
  root := nil;

  while l <> nil do
    begin
      AddToTrie (root, l^.code, 1, l^.organism);
      l := l^.next;
    end;

  root^.back := nil;
  ComputeBackFunction (root);
  CreateAutomaton := root;
end;

{ nájde všetkých predkov organizmu o }
procedure FindAncestors (o : pList; a : pTrieNode);
var i : integer;
    v, son : pTrieNode;
begin
  for i := 1 to length (o^.code) do
    begin
      while (a^.back <> nil) and (a^.sons[BaseToNumber (o^.code[i])] = nil) do
        a := a^.back;
      son := a^.sons[BaseToNumber (o^.code[i])];
```

```
      if son <> nil then
        a := son;

      if a^.organism = 0 then
        v := a^.backInSet
      else
        v := a;

      while (v <> nil) and (v^.seen <> o^.organism) do
        begin
          if v^.organism <> o^.organism then
            writeln (v^.organism, ' ', o^.organism);
          v^.seen := o^.organism;
          v := v^.backInSet;
        end;
    end;
end;

{ nájde všetkých predkov organizmu v zozname L }
procedure FindAllAncestors (l : pList; a : pTrieNode);
begin
  while l <> nil do
    begin
      FindAncestors (l, a);
      l := l^.next;
    end;
end;

function ReadOrganisms : pList;          { načíta zoznam organizmov }
var ret, act : pList;
    i : integer;
begin
  ret := nil;
  i := 1;
  while not eof do
    begin
      new (act);
      readln (act^.code);
      act^.organism := i;
      inc (i);
      act^.next := ret;
      ret := act;
    end;
  ReadOrganisms := ret;
end;

var organisms : pList;
    automaton : pTrieNode;
```

begin
  organisms := *ReadOrganisms*;
  automaton := *CreateAutomaton* (*organisms*);
  FindAllAncestors (*organisms*, *automaton*);
end.

P-I-4

Časť a)

Úlohu budeme riešiť podobne ako Príklad 2 zo študijného textu postupným prevádzaním na stále jednoduchšie problémy. Bez ujmy na všeobecnosti budeme predpokladať, že veľkosť vstupu N je mocnina dvojky (keby nebola, doplníme vstup nulami, čím sa výsledok evidentne nezmení a N sa maximálne zdvojnásobí).

Výpočet rozdelíme na fázy, pričom na konci i-tej fázy budú bity registra y rozdelené na bloky dĺžky 2^i bitov a v každom bloku bude uložený počet jednotkových bitov v príslušnom bloku vstupu. Také číslo sa iste do 2^i bitov vojde, pretože $2^i > i$ pre každé i. Hodnotu registra y na konci i-tej fázy označme y_i.

Počiatočné y_0 sa rovná vstupu x, pretože každý bit obsahuje počet jednotiek v sebe samom. Pre $i = \log_2 N$ dostaneme požadovaný výsledok, lebo y_i sa bude skladať z jediného bloku, v ktorom bude uložený počet jednotiek v celom vstupnom čísle. Stačí teda vyriešiť, ako z y_i spočítať y_{i+1}: Každý *veľký blok* v y_{i+1} obsahuje súčet dvoch *malých blokov* polovičnej veľkosti v y_i, ktoré ležia na mieste hornej, resp. dolnej polovice veľkého bloku. Preto stačí posunúť vyšší z malých blokov na pozíciu nižšieho a oba sčítať. To môžeme spraviť pre všetky veľké bloky naraz nasledujúcim programom: (b_j tu znamená jednotlivé malé bloky veľkosti $b = 2^i$, B_j sú výsledné veľké bloky veľkosti $B = 2b = 2^{i+1}$)

	$y = b_0 b_1 \ldots b_{N/b-1} = y_i$
$p := y \wedge (\mathbf{0}^b\mathbf{1}^b)^{N/B}$	$p = \mathbf{0}^b b_1 \mathbf{0}^b b_3 \ldots \mathbf{0}^b b_{N/b-1}$
$q := (y \gg b) \wedge (\mathbf{0}^b\mathbf{1}^b)^{N/B}$	$q = \mathbf{0}^b b_0 \mathbf{0}^b b_2 \ldots \mathbf{0}^b b_{N/b-2}$
$y := p + q$	$y = \ B_0 \ B_1 \ldots \ B_{N/B-1} = y_{i+1}$

Jednu fázu vykonáme v konštantnom čase. Celý program preto beží v čase $O(\log N)$ s registrami dĺžky $O(N)$.

Ukážka výpočtu pre vstup dĺžky 8:

	$x = \mathbf{0\,1\,1\,1\,1\,1\,0\,1}$
$y := x$	$y = \mathbf{0}\vert\mathbf{1}\vert\mathbf{1}\vert\mathbf{1}\vert\mathbf{1}\vert\mathbf{0}\vert\mathbf{1} = y_0$
$p := y \wedge \mathbf{01010101}$	$p = \cdot\ \mathbf{1}\vert\cdot\ \mathbf{1}\vert\cdot\ \mathbf{1}\vert\cdot\ \mathbf{1}$
$q := (y \gg 1) \wedge \mathbf{01010101}$	$q = \cdot\ \mathbf{0}\vert\cdot\ \mathbf{1}\vert\cdot\ \mathbf{1}\vert\cdot\ \mathbf{0}$
$y := p + q$	$y = \mathbf{0\,1}\vert\mathbf{1\,0}\vert\mathbf{1\,0}\vert\mathbf{0\,1} = y_1$
$p := y \wedge \mathbf{00110011}$	$p = \cdot\ \cdot\ \mathbf{1\,0}\vert\cdot\ \cdot\ \mathbf{0\,1}$
$q := (y \gg 2) \wedge \mathbf{00110011}$	$q = \cdot\ \cdot\ \mathbf{0\,1}\vert\cdot\ \cdot\ \mathbf{1\,0}$
$y := p + q$	$y = \mathbf{0\,0\,1\,1}\vert\mathbf{0\,0\,1\,1} = y_2$
$p := y \wedge \mathbf{00001111}$	$p = \cdot\ \cdot\ \cdot\ \cdot\ \mathbf{0\,0\,1\,1}$
$q := (y \gg 4) \wedge \mathbf{00001111}$	$q = \cdot\ \cdot\ \cdot\ \cdot\ \mathbf{0\,0\,1\,1}$
$y := p + q$	$y = \mathbf{0\,0\,0\,0\,0\,1\,1\,0} = y_3$

Časť b)

Nech zadané číslo x, ku ktorému máme nájsť najbližšie vyššie číslo y s rovnakým počtem jednotiek, obsahuje aspoň jednu jednotku (ak nie, hľadané y neexistuje a môžeme vrátiť ľubovoľný výsledok). Ak nájdeme v x posledný súvislý úsek jednotiek (môže to byť aj jediná jednotka), musí pred ním byť $\mathbf{0}$ (v opačnom prípade je x najvyššie číslo s daným počtom jednotiek a y opäť neexistuje). Číslo x sa teda dá zapísať v tvare $\alpha\mathbf{01}^k\mathbf{0}^l$.

Ukážeme, že hľadané číslo y sa rovná číslu $q = \alpha\mathbf{10}^{l+1}\mathbf{1}^{k-1}$:

- q obsahuje rovnaký počet jednotiek ako x.
- $q > x$ – pre každé α, β, γ, kde β a γ majú rovnakú dĺžku, totiž platí $\alpha\mathbf{1}\beta > \alpha\mathbf{0}\gamma$.
- Medzi x a q už žiadne číslo s rovnakým počtom jednotiek nie je – každé číslo medzi totiž musí buď vzniknúť z x zvýšením časti $\mathbf{0}^l$, čím by pribudli jednotky navyše, alebo z q znížením časti $\mathbf{1}^{k-1}$, a vtedy by jednotiek ubudlo.

Ako ale číslo q skonštruovať? Najprv postupom podobným ako v Príklade 1 spočítame niekoľko pomocných hodnôt:

	$x = \alpha\mathbf{01}^{k-1}\mathbf{10}^l$
$a := x - 1$	$a = \alpha\mathbf{01}^{k-1}\mathbf{01}^l$
$b := x \vee a$	$b = \alpha\mathbf{01}^{k-1}\mathbf{11}^l$
$c := x \wedge a$	$c = \alpha\mathbf{01}^{k-1}\mathbf{00}^l$
$d := b + 1$	$d = \alpha\mathbf{10}^{k-1}\mathbf{00}^l$
$e := c \oplus d$	$e = \mathbf{11}^{k-1}\mathbf{00}^l = \mathbf{1}^k\mathbf{0}^{l+1}$
$f := (e - 1) \wedge e$	$f = \mathbf{1}^{k-1}\mathbf{0}^{l+2}$

Teraz by stačilo jednotky v f posunúť k pravému okraju čísla a skombinovať s jednotkami v d a dostali by sme žiadané číslo q. Operácia $>>$ to ale sama o sobe nedokáže, lebo posunutie nemáme dané počtom bitov, ale podmienkou „prvá jednotka sa dotkne okraja“.

Znovu na to pôjdeme postupným zjednodušovaním problému a budeme bez ujmy na všobecnosti predpokladať, že N je mocnina dvojky. Fázy tentokrát očíslujeme odzadu: od $\log_2 N$-tej po nultú. V i-tej fáze zariadime, aby f končilo na menej než 2^i núl. Opäť v počiatočnej fáze nemáme čo na práci a v koncovej dostaneme očakávaný výsledok. Ostatné fázy budú fungovať takto: dostaneme f, ktoré končí na najviac 2^{i+1} núl a potrebujeme ho posunúť doprava tak, aby končilo na najviac 2^i núl. Stačí sa teda pozrieť, či je najnižších 2^i bitov nulových, a ak áno, f posunúť doprava o 2^i miest. To sa dá spraviť napríklad pomocou konštrukcie $r := \mathit{if}(s, t, u)$ z Príkladu 1:

$g := f \wedge \mathbf{1}^{2^i}$	$g =$ dolných 2^i bitov f_{i+1}
$h := \mathit{if}(g, 0, 2^i)$	$h =$ o koľko posúvame
$f := f >> h$	$f =$ výsledok i-tej fázy f_i

Po poslednej fáze zakončíme:

	$d = \alpha\mathbf{10}^{k-1}\mathbf{00}^l$
	$f = \mathbf{00}^{l+1}\mathbf{1}^{k-1}$
$y := d \vee f$	$y = \alpha\mathbf{10}^{l+1}\mathbf{1}^{k-1}$

a sme hotoví. Trvalo to celkovo $O(\log N)$ krokov (konštantný počet na inicializáciu, na koncový výpočet y a tiež na každú fázu). Potrebovali sme registre s $O(N)$ bitmi.

Príklad výpočtu pre $N = 8$:

	$x = \mathbf{10111000}$
$a := x - 1$	$a = \mathbf{10110111}$
$b := x \vee a$	$b = \mathbf{10111111}$
$c := x \wedge a$	$c = \mathbf{10110000}$
$d := b + 1$	$d = \mathbf{11000000}$
$e := c \oplus d$	$e = \mathbf{01110000}$
$f := (e - 1) \wedge e$	$f = \mathbf{01100000} = f_3$
$g := f \wedge \mathbf{00001111}$	$g = \cdots\mathbf{0000}$
$h := \mathit{if}(g, 0, 2^2)$	$h = \mathbf{00000100}$
$f := f >> h$	$f = \mathbf{00000110} = f_2$
$g := f \wedge \mathbf{00000011}$	$g = \cdots\mathbf{10}$
$h := \mathit{if}(g, 0, 2^1)$	$h = \mathbf{00000000}$
$f := f >> h$	$f = \mathbf{00000110} = f_1$
$g := f \wedge \mathbf{00000001}$	$g = \cdots\mathbf{0}$
$h := \mathit{if}(g, 0, 2^0)$	$h = \mathbf{00000001}$
$f := f >> h$	$f = \mathbf{00000011} = f_0$
$y := d \vee f$	$y = \mathbf{11000011}$

SLOVENSKÁ KOMISIA MATEMATICKEJ OLYMPIÁDY

54. ROČNÍK MATEMATICKEJ OLYMPIÁDY

Riešenia 1. kola kategórie P

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