P-I-1 Prádelna

Nejprve se zaměřme na to, kolika pračkami bude potřeba vybavit prádelnu. Je zřejmé, že je potřeba alespoň tolik praček, kolik má Bořivoj nejvýše v jednom okamžiku zákazníků. Později ukážeme, že tento počet praček je i dostatečný. Nejprve si však rozmysleme, jak určit maximální počet zákazníků prádelny v jednom okamžiku. Budeme simulovat dění v prádelně podle času a zároveň počítat počet zákazníků v prádelně. *Událostí* nazveme příchod nebo odchod některého ze zákazníků. Pro každého zákazníka si vytvoříme dvojici událostí: jednu pro příchod a druhou pro odchod zákazníka. Vytvořené události setřídíme podle jejich času a budeme sledovat příchody a odchody v pořadí, v němž jsou setříděny. Události se stejným časem zpracujeme naráz. Při odchodu snížíme počet zákazníků v prádelně, naopak při příchodu ho zvýšíme. Hledaný maximální počet zákazníků v jednom okamžiku je pak zřejmě maximum z takto získaných počtů zákazníků v prádelně.

Nyní ukážeme, že tento počet praček je dostatečný tím, že nalezneme přiřazení praček zákazníkům prádelny. Znovu spustíme simulaci příchodů a odchodů zákazníků. U každé pračky si budeme pamatovat, jestli je obsazená. V jednom okamžiku nejprve uvolníme pračky, které byly používány zákazníky, co právě odešli, a pak přiřadíme volné pračky nově příchozím zákazníkům. Zřejmě každému zákazníkovi takto přiřadíme jednu pračku a tedy počet praček je dostatečný.

Při samotné implementaci popsaného řešení ještě můžeme použít několik triků, jak náš algoritmus malinko vylepšit. Abychom se vyhnuli složitému zpracování událostí se stejným časem naráz, setřídíme události tak, že události odchodu jsou zařazeny před příchody, které nastanou ve stejný čas. Pak můžeme nejprve zpracovat odchody a poté příchody – tím zůstane korektnost algoritmu při obou simulacích zachována.

Další drobné vylepšení spočívá ve sloučení obou simulací do jedné. Začneme s prádelnou, ve které nejsou žádné pračky. Při příchodu zákazníka ho zkusíme umístit k nepoužívané pračce. Pokud žádná taková pračka neexistuje, „vytvoříme“ v prádelně novou pračku, ke které zákazníka pošleme. Při odchodu zákazníka jeho pračku uvolníme. Hledaný minimální počet praček je počet praček, které jsme v prádelně museli „vytvořit“ (počet praček, které existují, je roven maximálnímu počtu zákazníků, kteří byli dosud v prádelně ve stejném okamžiku).

Zbývá odhadnout časovou a paměťovou složitost našeho algoritmu. Nejprve musíme události setřídit, na což spotřebujeme čas $O(N \log N)$ při použití vhodného třídícího algoritmu. Připomeňme, že N je počet zákazníků prádelny. V implementaci jsme použili algoritmus quicksort, který má složitost $O(N \log N)$ sice jen v průměrném případě, ale místo něj lze samozřejmě použít libovolný třídící algoritmus s časovou složitostí $O(N \log N)$ v nejhorším případě. Poté provedeme jeden průchod setříděnými událostmi. Na zpracování jedné události nám stačí konstantní časová složitost. Celá simulace dění v prádelně má tak časovou složitost $O(N)$. Celková časová složitost našeho algoritmu je tedy $O(N \log N)$. Paměťová složitost je $O(N)$, což je množství paměti potřebné na uložení zakázek, volných a používaných praček (těch je nejvíce tolik, kolik je zákazníků) a setříděných příchodů a odchodů do paměti.

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

struct zakazka {
  long z,t;                             // začátek a doba trvání
};

struct udalost {
  long t;                               // čas události
  char prichod;                         // příchod nebo odchod
  int zakaznik;                         // číslo zákazníka této události
};

// porovnání udalostí: třídí se podle času a v případě shody časů nejdříve 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 prvního neobsazeného místa v stacku

int prirazeni_pracek[MAX_N];
int pocet_pracek=0;
```

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

int main(void)
{
  int i;

  fr=fopen("pradelna.in","r");          // načtení dat
  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++) {                   // vytvoření událostí
    udalosti[2*i].t=zakazky[i].z;               // příchod
    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++)                   // simulace provozu
    if (udalosti[i].prichod) {
      int pracka;
      if (!stack_top) /* nová pračka */ pracka=++pocet_pracek;
      else pracka=stack[--stack_top];
      prirazeni_pracek[udalosti[i].zakaznik]=pracka;
    } else {
      stack[stack_top++]=prirazeni_pracek[udalosti[i].zakaznik];
    }

  fw=fopen("pradelna.out","w");         // vypsání výstupu
  fprintf(fw,"%d\n",pocet_pracek);
  for (i=0;i<N;i++) fprintf(fw,"%d\n",prirazeni_pracek[i]);
  fclose(fw);

  return 0;
}
```

P-I-2 Závody

Tuto úlohu budeme řešit přímočaře. Budeme postupně číst čísla švábů, vždy zjistíme, kolik již doběhlo švábů s číslem vyšším, než je číslo právě zpracovávaného švába, a tento počet přičteme k výslednému počtu opačně uspořádaných dvojic. Takto zjevně započítáme každou opačně uspořádanou dvojici švábů právě jednou a získáme tedy hledanou míru promíchanosti. Pokud bychom se spokojili s kvadratickou časovou složitostí, mohli bychom počet švábů s vyšší číslem zjišťovat přímým průchodem již načtených čísel. My bychom ale rádi dosáhli lepší časové složitosti, a proto musíme být chytřejší. Nadále budeme předpokládat, že N (počet švábů) je mocnina dvojky – pokud není, můžeme si snadno představit, že vstupní posloupnost je doplněna tak, aby její délka byla mocninou dvou a nic tím nezkazíme (vstup si prodloužíme nejvýše dvakrát a počet dvojic ve špatném pořadí se nezmění). Budeme si průběžně udržovat informaci, kolik dosud načtených čísel bylo v intervalu $[N/2, \ldots, N-1]$, kolik v intervalech $[N/4, \ldots, N/2-1]$ a $[N/2, \ldots, 3N/4-1]$ a tak dále. Na počátku jsou tyto počty zjevně nulové. Když přečteme další číslo, podíváme se, zda je menší než $N/2$. Pokud ano, přičteme k výsledku počet čísel z intervalu $[N/2, \ldots, N-1]$ a pokračujeme v testování na intervalu $[0, \ldots, N/2-1]$. Jinak zvýšíme počet čísel v intervalu $[N/2, \ldots, N-1]$ a pokračujeme v testování na intervalu $[N/2, \ldots, N-1]$. Obecně na každém intervalu se podíváme, zda je nové číslo menší než střed intervalu. Pokud je, přičteme k výsledku počet čísel z horní poloviny intervalu a pokračujeme v dolní polovině. Pokud není, zvýšíme počet čísel v horní polovině intervalu o jedna a pokračujeme na ní v testování. Takto postupujeme, dokud interval nemá délku jedna. Tímto způsobem jsme dokázali zjistit počet již načtených čísel větších než aktuální číslo a zároveň jsme do naší struktury aktuální číslo zahrnuli. Protože se v každém kroku délka intervalu zkrátí na polovinu, jedno zjištění počtu větších čísel bude trvat $O(\log N)$. Celková časová složitost tedy je $O(N \log N)$. Paměťová složitost našeho algoritmu je $O(N)$.

Ve vzorovém programu se používá dvou ne zcela obvyklých obratů, a proto je zde alespoň stručně vysvětlíme. Zjišťování, do které poloviny intervalu aktuální číslo spadá, provádíme tak, že se podíváme na příslušný bit čísla (označme $b = \log_2 N$; nejdříve testujeme $b-1$-tý bit, pak $b-2$-tý až nakonec nultý bit) a pokud je 0, číslo je v dolní polovině intervalu, pokud 1, číslo je v horní polovině. Počet čísel v jednotlivých intervalech máme uložen v poli `Vetsi`. Na první pozici je uložen počet čísel

v jediném intervalu délky $N/2$ (to je interval $[N/2, \ldots, N]$), na dalších dvou pozicích počet čísel ve dvou intervalech délky $N/4$ (to jsou intervaly $[N/4, \ldots, N/2-1]$ a $[3N/4, \ldots, N]$), na dalších čtyřech pozicích počet čísel ve čtyřech intervalech délky $N/8$ atd. Intervaly stejné délky jsou vždy uspořádány vzestupně podle svých počátků. Rozmyslete si, že pokud je počet čísel v horní polovině nějakého intervalu uložen na pozici i, tak počet čísel ve druhé čtvrtině tohoto intervalu je uložen na pozici $2 \cdot i$ a počet čísel ve čtvrté čtvrtině je uložen na pozici $2 \cdot i + 1$.

```
program Zavody;
const
  MAXN = 8192;
  IFILE = 'zavody.in';
  OFILE = 'zavody.out';
var
  Vetsi : Array[1..MAXN] of Integer;    {Pole s počty větších prvků}
  N, C : Integer;                       {Počet čísel; Aktuálně načtené číslo}
  bits : Integer;                       {Nejbližší vyšší mocnina dvou od N}
  Inverzi : Longint;                    {Počet opačně uspořádaných dvojic}
  i : Integer;                          {Pomocná promenná}
  inp, out : Text;                      {Vstupní a výstupní soubor}

{Spočte dosavadní počet čísel vetších než C a C zařadí}
function Zarad(C : Integer) : Integer;
var
  i : Integer;                          {Pomocná proměnná}
  pocet : Integer;                      {Počet vetších čísel než C}
  act : Integer;                        {Aktuální vrchol ve stromu}
begin
  pocet := 0;
  act := 1;
  for i := bits-1 downto 0 do
    if (C and (1 shl i)) = 0 then begin {Spadá číslo do spodní poloviny?}
      pocet := pocet + Vetsi[act];      {Započteme čísla z horní poloviny}
      act := 2*act;
    end
    else begin                          {Číslo spadá do horní poloviny}
      Inc(Vetsi[act]);                  {Zvýšíme počet čísel v horní polovině}
      act := 2*act+1;
    end;
  Zarad := pocet;
end;

begin
  Inverzi := 0;
  Assign(inp, IFILE);
  Reset(inp);
  ReadLn(inp, N);
  bits := 0;                             {Zjistíme nejbližší vyšší mocninu dvou}
  while (1 shl bits < N) do
    Inc(bits);
  for i := 1 to (1 shl bits) do          {Inicializace}
    Vetsi[i] := 0;
  for i := 1 to N do begin               {Výpočet začíná}
    Read(inp, C);
    Inverzi := Inverzi + Zarad(C);
  end;
  Close(inp);
  Assign(out, OFILE);
  Rewrite(out);
  WriteLn(out, Inverzi);
  Close(out);
end.
```

P-I-3 Fylogenetika

Nejprve si zadefinujme některé pojmy a značení. Pojmy *slovo* a *řetězec* budou v následujícím textu znamenat to samé – konečnou posloupnost znaků z nějaké konečné abecedy (v našem případě '`A`', '`C`', '`G`' a '`T`'). Například `AAA` a `GGAGCTG` jsou řetězce. Speciální případ řetězce je prázdný řetězec, tj. takový, který neobsahuje žádné znaky. Budeme ho značit λ. *Délka* řetězce je počet znaků, z nichž se skládá. Délku řetězce v si budeme značit $|v|$. Tedy například $|\texttt{AAA}| = 3$, $|\lambda| = 0$.

Jsou-li u a v řetězce, označíme uv jejich *spojení.* Tedy pokud třeba $u = \texttt{AGG}$ a $v = \texttt{CCT}$, je $uv = \texttt{AGGCCT}$.

Počátečnímu úseku řetězce budeme říkat jeho *prefix* – tedy slovo u je prefixem slova v, pokud existuje řetězec w takový, že $uw = v$. Koncový úsek řetězce označíme jako jeho *suffix.* Slovo samotné je svým prefixem i suffixem (položíme-li w rovno prázdnému řetězci). Prefix (resp. suffix) slova v je *netriviální,* pokud je různý od v.

Podřetězec řetězce v je libovolný souvislý úsek znaků obsažený ve v – tedy w je podřetězec v, jestliže existují (ne nutně neprázdná) slova z a k taková, že $v = zwk$; jinak řečeno, pokud je w suffixem nějakého prefixu slova v.

Nyní se můžeme pustit do řešení úlohy. Zadané řetězce si označme s_1, s_2, ..., s_n. S bude součet jejich délek, tedy $S = |s_1| + |s_2| + \ldots + |s_n|$. Ze slov s_i si sestavíme *vyhledávací automat.* Vyhledávací automat je struktura, umožňující pro libovolný řetězec t určit všechna slova s_i, která se vyskytující jako podřetězec v t, v lineárním čase. Když toto budeme umět, řešení úlohy je již jednoduché – tento postup provedeme postupně pro $t = s_1$, $t = s_2$, ..., $t = s_n$, čímž si zjistíme postupně předky prvního, druhého, ..., n-tého organismu. Časová složitost bude $O(S + \text{čas na konstrukci automatu} + \text{délka výstupu})$.

Nyní zbývá vytvořit takový automat. Vyhledávací automat se skládá ze dvou částí – *trie* postavené z řetězců s_i a takzvané *zpětné funkce.*

Popis trie začneme příkladem – takto vypadá trie pro řetězce ze vzorového vstupu v zadání:

Trie je strom, jehož vrcholy odpovídají všem prefixům řetězců s_i. Když má několik řetězců s_i stejný začátek, odpovídá tomuto společnému prefixu jen jeden vrchol. Synové vrcholu odpovídajícího řetězci w jsou vrcholy odpovídající řetězcům wx, kde x je nějaký znak abecedy, v našem případě `A`, `C`, `G` nebo `T`. Každý vrchol má tedy nejvýše čtyři syny, ale může mít i méně, pokud žádné z s_i nezačíná na wx. Kořen stromu odpovídá prázdnému řetězci.

Některé vrcholy trie odpovídají slovům z s_i – například vrcholy, které nemají žádné syny, ale mohou to být i vrcholy uvnitř trie, pokud je některé ze slov prefixem jiného. Ostatní vrcholy budeme označovat jako *pomocné.*

O slovu budeme říkat, že je *reprezentováno* v trii, pokud je to jedno ze slov s_i, pro které jsme trii postavili. O slovu budeme říkat, že je v trii *obsaženo*, pokud mu odpovídá nějaký vrchol trie (může být i pomocný). Každé slovo, které je v trii reprezentováno, je v ní samozřejmě i obsaženo, opačné tvrzení obecně neplatí. Ve zbytku textu budeme občas ztotožnovat vrcholy trie se slovy, která jim odpovídají. Budeme-li například mít funkci, která vrcholu trie odpovídajícímu slovu v přiřazuje jiný její vrchol odpovídající slovu w, budeme občas pro zjednodušení říkat, že tato funkce přiřazuje slovu v slovo w.

V každém vrcholu trie budeme mít ukazatele na jeho čtyři syny odpovídající jednotlivým písmenům. Někteří z jeho synů samozřejmě nemusí existovat, v tom případě bude příslušný ukazatel `nil`. Kromě toho bude vrchol obsahovat hodnotu typu `boolean` udávající, zda daný vrchol odpovídá nějakému slovu reprezentovanému v trii, nebo zda je pouze pomocný.

Zjistit, zda je v trii obsaženo nějaké slovo, lze snadno v čase lineárním v délce tohoto slova: Začneme v kořeni. Z něj se posuneme do syna odpovídajícího prvnímu písmenu slova, z tohoto syna dále po hraně odpovídající druhému písmenu, atd. Pokud narazíme na `nil` dříve, než dojdeme ke konci slova, toto slovo se v trii nevyskytuje. Až dojdeme do vrcholu, který odpovídá zadanému slovu, ještě zkontrolujeme, zda je toto slovo v trii reprezentováno, tj. zda příznak vrcholu, do kterého jsme přišli, je `true`.

Obdobně můžeme trii postavit – provádíme postup analogický hledání, jen ve chvíli, kdy „vypadneme“ z trie (tj. narazíme na `nil`), začneme stavět novou cestu. Tento postup nám dohromady zabere čas $O(S)$.

Dále si nadefinujeme *zpětnou funkci* f, která každému vrcholu trie přiřadí nějaký jiný, odpovídající kratšímu slovu (proto zpětná). Pro vrchol w bude $f(w)$ definováno jako vrchol v takový, že v je nejdelší netriviální suffix w obsažený v trii.

Jednoduchý způsob, jak zpětnou funkci spočítat, je tedy tento: Vezmeme si slovo w a zahodíme z něj první písmeno. Pokud je takto vzniklé slovo v obsaženo v trii, je $f(w) = v$. Jinak z něj znovu zahodíme první písmeno a postup opakujeme, dokud hodnotu zpětné funkce neurčíme – to se určitě stane, neboť se zcela jistě zastavíme nejpozději na prázdném řetězci. Pro 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 funkce f je tento: Nechť máme nějaký text a chceme zjistit, která slova obsažená v trii končí na zadané pozici v tomto textu. Nechť nám navíc někdo prozradí, že s je nejdelší takové slovo. Pak $f(s)$ je druhé nejdelší, $f(f(s))$ třetí nejdelší, atd., dokážeme je tedy v lineárním čase vypsat (a navíc seřazené podle délky). To se nám bude hodit, protože při hledání pomocí automatu si budeme pamatovat pro každou pozici v textu vždy právě toto slovo s (přesněji řečeno vrchol v trii, který mu odpovídá).

Výše popsaný triviální postup, jak f spočítat, by nám zabral čas $O(S^3)$ – pro každý z nejvýše S vrcholů bychom museli provést hledání řádově S řetězců, jejichž délka může být až S. Samozřejmě bychom to chtěli zvládnout rychleji. K tomu použijeme postup založený na dynamickém programování. Funkci f budeme postupně počítat od nejkratších slov k delším. Pro jednopísmenná slova je $f(x) = \lambda$. Uvažujme nyní slovo wx, kde x je jeho poslední písmeno. Označme $v = f(wx)$. Víme, že v musí být nějaký suffix wx, tedy v končí na x (pokud není prázdné), tedy $v = v'x$ pro nějaké slovo v', které je suffixem w. Nyní využijeme toho, co jsme si ukázali v předchozím odstavci – všechny suffixy w, které jsou obsaženy v trii, jsou $f(w)$, $f(f(w))$, atd. Nás zřejmě zajímá nejdelší z nich, který se dá rozšířit o písmeno x tak, aby výsledné slovo bylo obsaženo v trii. Algoritmus na určení $f(wx)$ bude tedy tento:

Označme $w' = f(w)$, tuto hodnotu již máme spočítanou. Pokud $w'x$ je obsaženo v trii, je $f(wx) = w'x$, neboť w' je nejdelší suffix w v trii a tedy $f(wx)$ nemůže být delší. Pokud $w'x$ v trii není, vyzkoušíme $w'' = f(w')$ a takto pokračujeme, dokud buď nenalezneme hodnotu pro $f(wx)$, nebo nedorazíme k prázdnému řetězci – pak $f(wx) = \lambda$. Testování, zda $w'x$ je v trii, zvládneme v konstantním čase, neboť známe vrchol v trii odpovídající řetězci w'.

Jaká je časová složitost tohoto postupu? Při stanovování f pro jeden vrchol se nám může stát, že se funkcí f budeme muset vracet až S-krát, na první pohled by se tedy mohlo zdát, že časová složitost bude $O(S^2)$. Můžeme ovšem nahlédnout, že toto se nám nemůže stát příliš často: $f(wx)$ bude jen o jedna delší, než slovo $w^{''\cdots'}$, k němuž jsme dospěli při vracení se, tedy oproti $f(w)$ může být nanejvýš o jedna delší. Na to, abychom se mohli vracet o k hladin tedy nejprve musíme udělat alespoň k kroků, v nichž se nevracíme vůbec a tedy je zvládneme v konstantním čase. Tato úvaha není zcela přesná, nicméně není obtížné ji domyslet do všech detailů. Dohromady se tedy budeme vracet nejvýše $S\times$ a časová složitost bude $O(S)$.

Vraťme se ještě k motivaci pro definici funkce f. Řekli jsme si, že $f(s)$, $f(f(s))$, atd. jsou všechny suffixy slova s obsažené v trii. Nijak jsme však nerozlišovali, zda se jedná o původně zadaná slova, která jsou trií reprezentována, nebo pouze o nějaké jejich prefixy – tedy pomocné vrcholy, které žádné z původně zadaných slov nereprezentují. Samozřejmě někde mezi nimi jsou i všechna ta slova, která nás zajímají, nicméně mohlo by se stát, že „balastu“ okolo bude mnoho a jeho přeskakování nám zhorší časovou složitost.

Proto si ještě spočítáme funkci f', která pro vrchol v bude udávat nejdelší řetězec u různý od v takový, že u je reprezentováno v trii a dá se k němu dostat z v po zpětné funkci. Tato funkce bude dělat přesně to, co chceme – $f'(v)$, $f'(f'(v))$, atd. jsou právě všechna slova z původní množiny, končící na dané pozici. Spočítat f' je snadné – postupujeme od nejkratších slov a využijeme toho, že $f'(v) = f(v)$ pokud $f(v)$ je reprezentováno v trii a $f'(v) = f'(f(v))$ jinak. Toto lze také zřejmě provést v čase $O(S)$.

Tím jsme dokončili konstrukci automatu. Nyní zbývá ještě vysvětlit, jak takto získaný automat použít. Jak jsme řekli v úvodu, automat nám umožňuje rychle nalézt pro libovolný text všechna slova s_i, která jsou jeho podřetězcem. Označme si prohledávaný text t.

Pro každý počáteční úsek zadaného textu (slova) t bychom chtěli najít nejdelší suffix tohoto úseku, který je v trii obsažen (pak můžeme s použitím funkce f' snadno nalézt všechna s_i, která jsou suffixem libovolného počátečního úseku). Tyto suffixy postupně spočítáme pro počáteční úsek t délky 0, 1, 2, atd. Označme $l(i)$ hledaný suffix pro úsek délky i.

Počáteční úsek t délky 0 je prázdné slovo a tedy $l(0) = \lambda$.

Nechť jsme již spočítali l pro délku i a $l(i) = s$. Nechť písmeno na $(i+1)$-ní pozici v t je x. $l(i+1)$ má být nejdelší řetězec obsažený v trii, který se vyskytuje jako podřetězec t končící na pozici $i+1$, tedy $l(i+1)$ musí končit na x. $l(i+1)$ si tedy můžeme napsat jako rx pro nějaký řetězec r, který je také obsažen v trii. Navíc r musí to být suffix s. Projít všechny suffixy s už umíme – stačí projít s, $f(s)$, $f(f(s))$, atd. Mezi nimi hledáme nejdelší, který se dá rozšířit o x tak, abychom zůstali v trii. Všimněme si, že nejdelší takový suffix musí být první, na který narazíme.

Toto opakujeme, dokud nezpracujeme celý řetězec t. Podobný postup jsme již prováděli při konstrukci zpětné funkce, proto nás asi nepřekvapí, že i zde dosáhneme lineární časové složitosti: s se posune směrem od kořene nanejvýš $|t|$-krát (v každém kroku o jednu), tedy směrem ke kořeni se můžeme pohnout také nejvýše $|t|$-krát. Celková časová složitost je tedy $O(t)$.

Pro každou počáteční úsek s textu si navíc označíme slova ze zadání, která jsou jeho suffixy. Jak jsme si již rozmysleli, jsou to právě slova $f'(s)$, $f'(f'(s))$, atd., plus případně slovo s, je-li jedním ze zadaných. Chtěli bychom, aby nám na vypisování výstupu stačila časová složitost $O(\text{délka výstupu})$. To by se nám mohlo pokazit, pokud by se nějaké slovo r v textu vyskytovalo mnohokrát – pak totiž budeme mnohokrát zbytečně procházet posloupnost slov $f'(r)$, $f'(f'(r))$, ... To snadno napravíme tak, že jakmile narazíme na označené slovo, dál se již funkcí f' vracet nebudeme – víme totiž, že všechna další slova, ke kterým bychom se takto mohli dostat, jsme již vypsali, když jsme zde byli poprvé. Takto každé vypsané slovo navštívíme právě jednou, a tedy dosáhneme požadované časové složitosti.

Celková složitost řešení zadané úlohy je $O(S + \text{délka výstupu})$. To je zjevně optimální, protože minimálně musíme načíst vstup a vypsat výsledek. Paměťová složitost je $O(S)$.

```
program Phylogenetics;

type pList = ^List;                             { typ pro seznam organismů }
     List =
       record
         code : string;
         organism : integer;
         next : pList;
       end;

type Base = (bA, bC, bG, bT);
     pTrieNode = ^TrieNode;                     { typ pro vrchol trie }
     TrieNode =
       record
         sons : array[Base] of pTrieNode;       { větve odpovídající jednotlivým písmenům }
         back : pTrieNode;                      { zpětná funkce }
         backInSet : pTrieNode;                 { zpětná funkce iterovaná až k prvnímu vrcholu
                                                  reprezentujícímu organismus }
         organism : integer;                    { číslo organismu representovaného vrcholem
                                                  nebo 0, pokud takový není }
         seen : integer;                        { byl-li již tento organismus vypsán při
                                                  prohledávání jeho potomka, číslo tohoto potomka }
         next : pTrieNode;                      { následující vrchol trie ve frontě }
       end;

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

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

function NewTrieNode : pTrieNode;               { vytvoří nový vrchol trie }
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;

{ přidá suffix řetězce S od pozice 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
      { jsme na konci řetězce }
      root^.organism := organism;
    end
  else
    begin
      { přidáme organismus do větve určené aktuální bazi }
      AddToTrie (root^.sons[BaseToNumber (s[p])], s, p + 1, organism);
    end;
end;

{ spočítá zpětnou funkci pro vrchol V trie, jehož otec je O a do v se 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;

{ spočítá zpětnou funkci pro trii s vrcholem 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;
```

```
{ vytvoří vyhledávací automat ze zadaného seznamu řetězců }
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;

procedure FindAncestors (o : pList; a : pTrieNode);  { nalezne všechny předky organismu O }
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;

{ nalezne všechny předky organismu v seznamu L }
procedure FindAllAncestors (l : pList; a : pTrieNode);
begin
  while l <> nil do
    begin
      FindAncestors (l, a);
      l := l^.next;
    end;
end;

function ReadOrganisms : pList;                 { načte seznam organismů }
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;

procedure DumpAuto(x : pTrieNode; indent : integer);  { vypíše automat odsazený o indent mezer }
  procedure ind;
  var j : integer;
  begin
    for j := 1 to indent do write (' ');
  end;
var i : Base;
begin
  ind; writeln ('node ', integer(x));
  ind; writeln ('  organism ', x^.organism);
  ind; writeln ('  back ', integer(x^.back));
  ind; writeln ('  backInSet ', integer(x^.backInSet));
  writeln;
  for i := bA to bT do
    if (x^.sons[i] <> nil) and (x^.sons[i] <> x) then
      begin
        ind; writeln ('  sub ', baseToChar (i));
        DumpAuto (x^.sons[i], indent + 4);
      end;
end;

var organisms : pList;
    automaton : pTrieNode;

begin
  organisms := ReadOrganisms;
  automaton := CreateAutomaton (organisms);
  { DumpAuto(automaton, 0); }
  FindAllAncestors (organisms, automaton);
end.
```

P-I-4 ALIK

a) Úlohu budeme řešit podobně jako Příklad 2 ze studijního textu postupným převáděním na stále jednodušší problémy. Bez újmy na obecnosti budeme předpokládat, že velikost vstupu N je mocnina dvojky (kdyby nebyla, doplníme si vstup nulami, čímž se výsledek evidentně nezmění a N se maximálně zdvojnásobí).

Výpočet rozdělíme na fáze, přičemž na konci i-té fáze budou bity registru y pomyslně rozděleny na bloky o 2^i bitech a v každém bloku bude uložen počet jedničkových bitů v odpovídajícím bloku vstupu. Takové číslo se jistě do 2^i bitů vejde, protože $2^i > i$ pro každé i. Hodnotu registru y na konci i-té fáze si označme y_i.

Počáteční y_0 je rovno vstupu x, protože každý bit obsahuje počet jedniček v sobě samém. Pro $i = \log_2 N$ dostaneme požadovaný výsledek, neboť y_i se bude skládat z jediného bloku, v němž bude uložen počet jedniček v celém vstupním čísle. Stačí tedy vyřešit, jak z y_i spočíst y_{i+1}: Každý *velký blok* v y_{i+1} obsahuje součet dvou *malých bloků* poloviční velikosti v y_i, které leží na místě horní, resp. spodní poloviny velkého bloku. Proto stačí posunout si vyšší z malých bloků na pozici nižšího a oba sečíst. To můžeme provést pro všechny velké bloky najednou následujícím programem: (b_j zde značí jednotlivé malé bloky velikosti $b = 2^i$, B_j jsou výsledné velké bloky velikosti $B = 2b = 2^{i+1}$)

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

Jednu fázi tedy provedeme v konstantním čase. Celý program proto proběhne v čase $O(\log N)$ s registry délky $O(N)$.

Ukázka výpočtu pro vstup délky 8:

	$x = \mathbf{0\ 1\ 1\ 1\ 1\ 0\ 1}$
$y := x$	$y = \mathbf{0}\|\mathbf{1}\|\mathbf{1}\|\mathbf{1}\|\mathbf{1}\|\mathbf{0}\|\mathbf{1} = y_0$
$p := y \wedge \mathbf{01010101}$	$p = \cdot\ \mathbf{1}\|\cdot\ \mathbf{1}\|\cdot\ \mathbf{1}\|\cdot\ \mathbf{1}$
$q := (y \gg 1) \wedge \mathbf{01010101}$	$q = \cdot\ \mathbf{0}\|\cdot\ \mathbf{1}\|\cdot\ \mathbf{1}\|\cdot\ \mathbf{0}$

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

b) Nechť zadané číslo x, ke kterému máme najít nejbližší vyšší číslo y se stejným počtem jedniček, obsahuje alespoň jednu jedničku (pokud ne, hledané y neexistuje a my můžeme vydat libovolný výsledek). Najdeme-li v x poslední souvislý úsek jedniček (může to být i jediná jednička), musí před ním být $\mathbf{0}$ (v opačném případě je x nejvyšší číslo s daným počtem jedniček a y opět neexistuje). Číslo x tedy lze zapsat ve tvaru $\alpha\mathbf{01}^k\mathbf{0}^l$.

Ukážeme, že hledané číslo y bude rovno číslu $q = \alpha\mathbf{10}^{l+1}\mathbf{1}^{k-1}$:

1. q obsahuje stejný počet jedniček jako x.
2. $q > x$ – pro každá α, β, γ, kde β a γ mají stejnou délku, je totiž $\alpha\mathbf{1}\beta > \alpha\mathbf{0}\gamma$.
3. Mezi x a q již žádné číslo se stejným počtem jedniček není – každé číslo mezi totiž musí buďto vzniknout z x zvyšením části $\mathbf{0}^l$, čímž by přibyly přespočetné jedničky, nebo z q snížením části $\mathbf{1}^{k-1}$, a tehdy by jedniček příliš ubylo.

Jak ale číslo q zkonstruovat? Nejprve postupem podobným jako v Příkladu 1 spočítáme několik pomocných hodnot:

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

Nyní by stačilo jedničky v f posunout k pravému okraji čísla a zkombinovat s jedničkami v d a dostali bychom kýžené číslo q. Operace $>>$ to ale sama o sobě nedokáže, neboť posunutí nemáme dáno počtem bitů, nýbrž podmínkou „první jednička se dotkne okraje“.

Znovu na to půjdeme postupným zjednodušováním problému a budeme bez újmy na obecnosti předpokládat, že N je mocnina dvojky. Fáze si tentokrát očíslujeme pozpátku: od $\log_2 N$-té po nultou. V i-té fázi zařídíme, aby f končilo na méně než 2^i nul. Opět v počáteční fázi nemáme co na práci a v koncové dostaneme očekávaný výsledek. Ostatní fáze budou fungovat takto: dostaneme f, které končí na nejvýše 2^{i+1} nul a potřebujeme ho posunout doprava tak, aby končilo na nejvýše 2^i nul. Stačí se tedy podívat, zda je nejnižších 2^i bitů nulových, a pokud ano, f posunout doprava o 2^i míst. To lze provést například pomocí konstrukce $r := if(s, t, u)$ z Příkladu 2:

$g := f \wedge \mathbf{1}^{2^i}$	$g =$ dolních 2^i bitů f_{i+1}
$h := if(g, 0, 2^i)$	$h =$ o kolik posouváme
$f := f >> h$	$f =$ výsledek i-té fáze f_i

Po poslední fázi 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 jsme hotovi. Trvalo nám to celkem $O(\log N)$ kroků (konstantně mnoho na inicializaci, na konečný výpočet y a rovněž na každou fázi), potřebovali jsme registry o $O(N)$ bitech.

Příklad výpočtu pro $N = 8$:

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

$h := if(g, 0, 2^0)$	$h = \mathbf{00000001}$
$f := f >> h$	$f = \mathbf{00000011} = f_0$
$y := d \vee f$	$y = \mathbf{11000011}$