P-II-1 Prádelní salón

Při řešení této úlohy použijeme řešení úlohy P-I-1 Prádelna z domácího kola. Jedinou změnou oproti domácímu kolu je podmínka, že žádný zákazník v době, kdy bude prát, neuvidí dva různé zákazníky používat tutéž pračku.

Jinak řečeno, pračka, kterou používal zákazník Cyril, může být použita až v momentě, kdy ze salónu odejde poslední zákazník, který pral prádlo současně s Cyrilem. Řekněme, že ze všech zákazníků, kteří prali prádlo zároveň s Cyrilem, je Metoděj ten, který odejde ze salónu nejpozději. Potom pračku, kterou používal Cyril, může použít další zákazník až po odchodu Metoděje ze salónu.

Využijeme tohoto pozorování tak, že z původních zakázek vytvoříme nové zakázky, které budeme nazývat „stínové“. Stínová zakázka zákazníka A bude stejná jako původní zakázka zákazníka A, pokud je v době jeho odchodu salón prázdný. Pokud tomu tak není, konec stínové zakázky zákazníka A bude čas, kdy ze salónu odejde poslední zákazník, který pral prádlo zároveň se zákazníkem A.

Řešení úlohy s původními zakázkami a podmínkou, že žádný zákazník v době, kdy bude prát, neuvidí dva různé zákazníky používat tutéž pračku, je shodné s řešením úlohy se stínovými zakázkami bez této podmínky. Stínové zakázky drží nějaké pračky „obsazené“ až do doby, než je může Bořivoj znovu použít. K nalezení řešení úlohy se stínovými zakázkami pak lze použít řešení domácího kola.

Při řešení úlohy budeme postupovat tak, že ze zadaných zakázek vytvoříme stínové. Takto vytvořenou úlohu pak vyřešíme jako v domácím kole. Stínové zakázky vytvoříme následujícím postupem. Jako v domácím kole vytvoříme události příchodu a odchodu pro každého zákazníka. Tyto události setřídíme (události odchodu před příchody, které nastanou ve stejný čas). Setříděné události jednou projdeme (podle vzrůstajícího času) a budeme si udržovat dobu T, kdy ze salónu odejde poslední zákazník, který je v salónu právě přítomen. Dobu T lze snadno udržovat tak, že kdykoliv narazíme na příchod nějakého zákazníka, novou hodnotu doby T zjistíme jako maximum její minulé hodnoty a doby, kdy chce zpracovávaný zákazník ze salónu odejít.

Průchod nad setříděnými událostmi bude probíhat tak, že když narazíme na

- *příchod* zákazníka A, upravíme dobu T.
- *odchod* zákazníka A, vytvoříme stínovou zakázku pro zákazníka A, jejíž příchod nastavíme na příchod zákazníka A a odchod na dobu T (která je v té chvíli větší nebo rovna době, kdy chtěl ze salónu odejít zákazník A).

Tento průchod lze provést v čase $O(N)$. Třídění událostí nám ale zabere čas $O(N \log N)$. Paměti budeme potřebovat $O(N)$ na uložení zadaných zakázek, stínových zakázek a setříděných událostí. Druhá část řešení, algoritmus z domácího kola, má stejnou časovou i paměťovou složitost jako právě popsané vytvoření stínových zakázek, a tak je celková časová složitost našeho řešení $O(N \log N)$ a paměťová pak $O(N)$.

```
#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 událostí: třídí se podle času a v případě shody času 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 ve stacku */

int prirazeni_pracek[MAX_N];
```

```
int pocet_pracek=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 pro slévání */
  prichody[prichody_len++].t=-1;
  odchody[odchody_len++].t=-1;

  /* vytvoření příchodů a odchodů */
  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;
    }

  /* ještě slít příchody a odchody do událostí */
  prichody_len--; /* aby ukazoval ne za platný příchod, ale na něj */
  odchody_len--;  /* to samé s odchody */
  i=2*N-1;        /* index na konec pole udalostí, plníme ho zezadu */
  while (prichody[prichody_len].t!=-1 || odchody[odchody_len].t!=-1)
    if (prichody[prichody_len].t >= odchody[odchody_len].t) /* přidej příchod */
      udalosti[i--]=prichody[prichody_len--];
    else /* přidej odchod */
      udalosti[i--]=odchody[odchody_len--];
}

int main(void)
{
  int i;

  /* Načtení dat */
  printf("Zadejte počet zákazníků:");
  scanf("%d",&N);
  for (i=0;i<N;i++) {
    printf("Doba příchodu a čas praní %d. zákazníka:",i+1);
    scanf("%ld %ld",&zakazky[i].z,&zakazky[i].t);
  }

  /* Vytvoření událostí */
  for (i=0;i<N;i++) {
    /* příchod */
    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);

  /* změna oproti domácímu kolu */
```

```
  uprav_udalosti();

  /* simulace provozu */
  for (i=0;i<2*N;i++)
    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 /* odchod */ {
      stack[stack_top++]=prirazeni_pracek[udalosti[i].zakaznik];
    }

  /* vypsání dat */
  printf("Je třeba alespoň %d praček.\n",pocet_pracek);
  for (i=0;i<N;i++) printf("Zákazník %d půjde k pračce %d.\n",i+1,prirazeni_pracek[i]);

  return 0;
}
```

P-II-2 Zakázané rozdíly

Nejdříve provedeme substituci $x_i = y_i + a_{i,N+1}$ pro i, $1 \le i \le N$, a $x_{N+1} = y_{N+1}$. Například pro soustavu z prvního příkladu v zadání úlohy dostane po substituci:

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

Pro soustavu z druhého příkladu dostaneme po substituci:

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

Nově získaná soustava podmínek (nerovnic) má řešení právě tehdy, pokud ho měla ta původní: Když x_i, $1 \le i \le N+1$, řeší původní soustavu, pak řešení nové soustavy nerovnic je $y_i = x_i - a_{i,N+1}$ pro $i \le N$ a $y_{N+1} = x_{N+1}$; naopak z řešení nové soustavy získáme řešení původní soustavu přičtením hodnot $a_{i,N+1}$.

Povšimněme si, že všechny nerovnosti, ve kterých se vyskytuje y_{N+1}, mají pravou stranu rovnu 0 (po odečtení $a_{i,N+1}$ od obou stran nerovnosti):

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

Podívejme se blíže na nově získanou soustavou nerovnic. Pokud jsou všechny pravé strany rovny nule, znamená to, že všechny neznámé $y_1, \ldots, y_{N+1}$ musí být navzájem různé. Protože ale existuje pouze N čísel s různými zbytky po dělení N, soustava nerovnic nemá řešení. Vzápětí ukážeme, že pokud nejsou všechny pravé strany rovny nule, pak nová soustava nerovnic, a tedy i ta původní mají řešení.

Označme $a'_{i,j}$ pravou stranu nerovnice, jejíž levá strana je $y_i - y_j$. Předpokládejme, že $a'_{i,j} \neq 0$ pro nějaké i a j. Protože $a'_{i,N+1} = 0$ z volby y_i, musí platit $1 \le i < j \le N$. Položme $y_i = y_j = 0$. Protože $a'_{i,j} \neq 0$, nerovnost $y_i - y_j \neq a'_{i,j}$ je tím zjevně splněna. Nyní zvolíme hodnoty ostatních neznámých postupně od y_1 do y_N, tak aby všechny nerovnosti, které je obsahují, byly splněny. V okamžiku, kdy volíme hodnotu neznámé y_k, $1 \le k \le N$, $k \neq i, j$, tak se neznámá y_k vyskytuje v nejvýše $N-1$ nerovnostech s ostatnimi neznámými, jejichž hodnotu jsme již zvolili. Protože každá nerovnost „zakazuje“ přiřazení právě jednoho z čísel $0, \ldots, N-1$ neznámé y_k, je dohromady zakázáno nejvýše $N-2$ hodnot a lze y_k nějakou hodnotu přiřadit.

Zbývá zvolit hodnotu y_{N+1}. Po provedení naší substituce, byly pravé strany všech N nerovností, ve kterých se y_{N+1} vyskytuje rovny nule. Tedy hodnota y_{N+1} musí být různá od hodnot $y_1, \ldots, y_N$. Protože $y_i = y_j$, mají neznámé $y_1, \ldots, y_N$ nejvýše $N-1$ různých hodnot, a proto lze y_{N+1} přiřadit (alespoň) jednu z hodnot $0, \ldots, N-1$.

Naše dosavadní úvahy vedou přímočaře ke kvadratickému algoritmu, který řeší zadanou úlohu. Nejprve provedeme substituci popsanou v prvním odstavci. Pokud jsou všechny pravé strany po substituci rovny 0, nová i původní soustava nerovnic nemají řešení. V opačném případě nalezneme řešení nové soustavy nerovnic postupem popsaným v předchozích

dvou odstavcích. Řešení původní soustavy snadno získáme aplikací substituce inverzní k první provedené substituci. Časová i paměťová složitost našeho algoritmu je kvadratická v N, tj. $O(N^2)$.

```
program rozdily;
const MAX=1000;
var A: array[1..MAX,1..MAX] of integer;         { pravé strany nerovností }
    N: integer;                                 { počet neznámých }
    subst: array[1..MAX+1] of integer;          { substituční posun }
    y: array[1..MAX+1] of integer;              { řešení nové soustavy }

procedure nacti;
  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 spocitej: boolean;
  var i,j,k,l:integer;                          { pomocné proměnné i, j a k jako v popisu algoritmu }
      zakazane:array[0..MAX-1] of boolean;      { pole zakazaných hodnot pro neznámou }
  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
        spocitej:=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;
    spocitej:=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
  nacti;
  substituuj;
  if spocitej then
    vypis
  else
    writeln('Zadaná soustava nerovnic nemá řešení.');
end.
```

P-II-3 Redundantní redundance

1. řešení: Nejprve si ukážeme jednoduché řešení pracující v čase $O(kn)$, založené na příhrádkovém třídění, zvaném též RadixSort či BucketSort. Jak takové třídění funguje? Pokud chceme setřídit m řetězců $s_1, \ldots, s_m$ délky l, nejdříve je setřídíme podle posledního písmene, pak podle předposledního (přičemž pokud se předposlední písmeno shoduje, zachováme pořadí podle posledního písmene) atd. až podle prvního písmene. Třídění podle i-tého písmene (tomu budeme říkat jeden *průchod*) provádíme tak, že si založíme příhrádky indexované písmeny, jednotlivé řetězce (respektive jejich čísla) rozmístíme do příhrádek podle toho, jaké je jejich i-té písmeno, a nakonec příhrádky projdeme od nejmenšího písmene k největšímu a řetězce z nich vysbíráme.

Každý průchod bude trvat $O(m)$ [čas závisí i na velikosti abecedy, protože musíme projít i prázdné příhrádky, ale to pro nás bude konstanta, tudíž se „schová do O-čka"], průchodů je l, takže celkem tříděním strávíme čas $O(lm)$.

Abychom vyřešili naši úlohu, nalezneme v zadaném řetězci všechny podřetězce délky k (těch je $n-k$), setřídíme je RadixSortem a poté v setříděném seznamu najdeme nejdelší úsek tvořený stejnými podřetězci. Ovšem všimneme si, že si při třídění podřetězců nemusíme pamatovat celé podřetězce, ale stačí nám jejich začátky ve „velkém" řetězci. Takto vše zvládneme v čase $O(kn)$ a prostoru $O(n)$.

Ještě jedna poznámka k implementaci: abychom zbytečně neplýtvali pamětí, neukládáme jednotlivé příhradky jako oddělená pole, ale všimneme si, že všechny příhrádky dohromady obsahují jen n prvků, takže je naskládáme do jednoho n-prvkového pole, jen si v druhém poli pamatujeme, kde která příhrádka začíná. Z vysbírání hodnot z příhrádek se pak stane jen zkopírování jednoho pole do druhého.

```
program RedundancePoprve;
const max=100;
var T:string[max];                      { Zadaný řetězec }
    k:integer;                          { Jak dlouhé podřetězce hledáme }
    n:integer;                          { Kolik existuje podřetězců délky k }
    a,b:array [1..max] of integer;      { Pole, které třídíme; pole s příhrádkami }
    p:array [char] of integer;          { Velikosti/začátky příhrádek }
    i,j,l,maxcnt,maxpos:integer;        { Pomocné proměnné }
    c:char;

begin
   readln(T);                           { Přečteme si vstup }
   readln(k);
   n := length(T)-k+1;
   for i:=1 to n do                     { Všechny možné pozice podřetězců }
      a[i] := i;

   for j:=k-1 downto 0 do begin         { RadixSort; podle j-tého písmene }
      for c:=#0 to #255 do              { Spočteme velikosti příhrádek }
```

```
         p[c] := 0;
      for i:=1 to n do
         inc(p[T[a[i]+j]]);
      l := 1;                          { Spočteme jejich začátky }
      for c:=#0 to #255 do begin
         l := l + p[c];
         p[c] := l - p[c];
         end;
      for i:=1 to n do begin           { Rozdělíme do příhrádek }
         c := T[a[i]+j];
         b[p[c]] := a[i];
         inc(p[c]);
         end;
      for i:=1 to n do                 { A opět z nich vybereme }
         a[i] := b[i];
      end;

   i := 1;                             { Hledáme nejdelší 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));          { A vypíšeme výstup, hurá! }
   writeln(maxcnt);
end.
```

2. řešení: Existuje algoritmus řešící zadanou úlohu a pracující v čase $O(n)$, kde n je délka textu T, je však poměrně komplikovaný. My si místo něj ukážeme o něco jednodušší *randomizovaný* algoritmus, tj. algoritmus používající při výpočtu náhodná čísla a pracující v čase $O(n)$ alespoň v průměrném případě. Tím je míněno, že pokud budeme mít smůlu na to, jaká čísla nám padají z generátoru náhodných čísel, může to trvat i déle, nicméně ve většině případů nám (nezávisle na vstupních datech) dá výsledek v čase $O(n)$.

I toto řešení je dosti trikové a využívá netriviální výsledky. Nepředpokládáme samozřejmě, že byste tyto techniky měli znát či dokázat aktivně používat – uvádíme ho spíše pro zajímavost a případně jako inspiraci pro hlubší zájemce o obor.

Náš randomizovaný algoritmus bude fungovat tak, že nejprve řešení „uhodne“, ověří si, že uhodnuté řešení je opravdu správné, a pokud nebude, celý postup zopakuje. Takový algoritmus samozřejmě nemusí nikdy skončit, ale my si dokážeme, že uhodnuté řešení bude správné s pravděpodobnostní alespoň $1/2$, takže v průměrném případě budeme potřebovat nejvýše 2 pokusy. Jak hádání, tak ověřování budou zabírat čas $O(n)$, takže tuto časovou složitost bude mít v průměru i celý algoritmus.

Nejprve si rozmysleme, že pokud uhodneme, že se nějaký řetězec délky k opakuje v textu l-krát, můžeme si snadno ověřit, že tomu tak skutečně je. K tomu použijeme řešení úlohy z domácího kola – sestrojíme si vyhledávací automat pro tento podřetězec, projedeme s ním text a spočítáme počet jeho výskytů, tj. počet průchodů finálním stavem. To nám zabere čas $O(n)$ – v čase $O(k)$ sestrojíme automat, v čase $O(n)$ projedeme text automatem, a $k < n$.

Fáze hádání funguje takto: Nejprve si zvolíme hashovací funkci. To je nějaká funkce h, zobrazující řetězce délky k na celá čísla mezi 0 a $p-1$ (hodnotu p si zvolíme později). Samozřejmě se nám může stát, že dva řetězce zobrazíme na stejné číslo (tomu se říká kolize), nicméně pokud bude hashovací funkce dobrá, nebude se to stávat často. Nyní každý podřetězec textu T délky k zobrazíme touto funkcí a spočítáme počty řetězců, které se zobrazí na jedno číslo. Z těchto počtů si vezmeme ten největší a vrátíme jeden z případně více řetězců, které se na příslušné číslo zobrazily. Pokud tento řetězec nekolidoval s žádným jiným, máme vyhráno, protože všechny ostatní řetězce (i když jsme některé kvůli kolizím nedokázali od sebe rozlišit) nejsou častější; pokud kolidoval, odhalí to kontrola.

Zbývá domyslet detaily tak, abychom vše zvládli v lineárním čase:

Nalezení nejčastější hodnoty hashovací funkce: Poslouží nám opět RadixSort: každé číslo si zapíšeme ve tvaru $a_3n^3 + a_2n^2 + a_1n + a_0$, kde a_i jsou menší než n (to není nic jiného, než zápis čísla v soustavě o základu n) a budeme ho třídit jako řetězec $a_3a_2a_1a_0$ nad n-prvkovou abecedou. Jak už víme, RadixSort takové třídění zvládne v čase $O(n)$.

Počítání hashovací funkce musíme udělat šikovně (ve skutečnosti toto je hlavní trik celého řešení, zbytek jsou jen technické detaily), jinak bychom zde potřebovali čas nk nebo větší. Trik je v tom, že si zvolíme takovou hashovací funkci, abychom (pro w slovo délky $k-1$, s a t písmena) dokázali $h(ws)$ spočítat se znalostí $h(tw)$ v konstantním čase. Pak pokud budeme hashovací funkce podřetězců počítat postupně od začátku a posunovat se vždy o jedno písmeno, spotřebujeme skutečně čas jen $O(n)$.

Hashovací funkci si zvolíme takto: p bude náhodně zvolené prvočíslo mezi $kn^3/2$ a kn^3,

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

(předpokládáme, že abeceda [nebo spíše ASCIIbeceda] má 256 znaků).

Výraz v závorce si označme $X(s)$. $X(s)$ samozřejmě může být větší než je rozsah čísla reprezentovaného v počítači, nicméně modulení p lze při jeho výpočtu provádět průběžně, takže nemůžeme přetéct. Postupné počítání funkce h je pak jednoduché, neboť zřejmě $h(ws) = (256 \cdot h(tw) - (256^k \bmod p)t + s) \bmod p$.

Je snadné nahlédnout, že pravděpodobnost, že dojde ke kolizi, je menší než 1/2: aby dvěma různým řetězcům s_1 a s_2 byla přiřazena stejná hodnota, musel by rozdíl $X(s_1) - X(s_2)$ být dělitelný p. Velikost tohoto rozdílu je nanejvýš 256^k, tedy ho dělí nanejvýš $8 \log_2 k$ různých prvočísel (každé z nich má velikost alespoň 2 a kdyby jich bylo víc, jejich součin by musel být větší než 256^k). Různých řetězců je nanejvýš n, tedy jejich dvojic je nanejvýš $n^2/2$ a „špatných“ prvočísel nanejvýš $4kn^2$. Poměrně netriviální výsledek z teorie čísel nám říká, že počet prvočísel mezi $kn^3/2$ a kn^3 je přibližně $\frac{kn^3}{2 \log kn^3}$, tedy pravděpodobnost, že se trefíme do špatného prvočísla, je menší než $\frac{8 \log kn^3}{n} \leq \frac{8 \log n^4}{n} = \frac{32 \log n}{n}$, což je menší než 1/2 pro $n \geq 381$ – ve skutečnosti jsou uvedené odhady poměrně hrubé, takže tato pravděpodobnost je ještě podstatně menší.

Zbývá jediný technický detail – jak nalézt náhodné prvočíslo. To lze provést v čase $O(\log^d n)$, kde d je nějaká konstanta, nicméně není to úplně triviální. Místo toho v programu volíme pouze náhodné číslo v daném intervalu; do prvočísla se trefíme s pravděpodobností přibližně $1/\log n$, tedy časová složitost se tím zhorší nanejvýš na $O(n \log n)$.

```
program RedundancePodruhe;
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;

{ Spočítá počet výskytů řetězce začínajícího na pozici idx a uloží tuto hodnotu do nRep }
procedure Verify (idx : integer; var nRep : integer);
var back : pole;
    f, i : integer;
begin
  { spočítá zpětnou funkci pro řetězec od pozice 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;

  { spočítá počet výskytů řetězce }
  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;

{ Spočítá hashovací funkci pro začátek 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;

{ Posune hashovací funkci h na pozici 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;

{ Vybere z a n-tou číslici }
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;

{ Setřídí pole a délky m podle n-té číslice a výsledek uloží do sorted }
procedure SortByDigit (var a : pole; m, n : integer; var sorted : pole);
var nReps : pole;
    i, d : integer;
begin
  { spočítá pocty opakování }
  for i := 0 to 2*l do
    nReps[i] := 0;
  for i := 1 to m do
    inc (nReps[digit (a[i], n) + 1]);

  { spočítá začátky úseků prvků }
  for i := 1 to 2*l do
    nReps[i] := nReps[i - 1] + nReps[i];

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

```
{ Setřídí pole a délky m a výsledek 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;

{ Nalezne v poli a délky m číslo, které se v něm nejčastěji opakuje,
  počet opakování 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;

{ "Uhodne" podřetězec, který se v T vyskytuje nejčastěji.  Počet výskytů uloží do nRep,
  začátek prvního 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;

{ Hlavní 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 ALÍK

Tuto úlohu bychom mohli řešit podobně jako úlohy z minulého kola v čase $O(\log N)$ s $O(N)$-bitovými čísly – stačí si všimnout, že otočení čísla délky N lze provést prohozením jeho polovin (což zvládneme na konstantní počet operací) a následným otočením obou polovin (což můžeme udělat současně). Pro $N = 8$ by to vypadalo takto:

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

Ovšem pokud využijeme násobení a dělení, zvládneme totéž na konstantní počet operací, i když, pravda, budeme potřebovat čísla délky $O(N^2)$.

Náš algoritmus bude založen na tom, že zadané číslo $x = x_{N-1} \ldots x_0$ nejprve N-krát zkopírujeme (to lze provést násobením), pak i-tou kopii nahradíme číslem, které bude na i-tém místě obsahovat x_{N-1-i} a všude jinde nuly, načež všechny kopie sečteme (k tomu lze opět použít násobení, ale my si ukážeme pěkný trik se zbytkem po dělení).

Kopírování provedeme takto: číslo x vynásobíme číslem $(\mathbf{0}^{N-1}\mathbf{1})^{N+1}$, čímž získáme $(x_{N-1} \ldots x_0)^{N+1}$, čili $N + 1$ kopií zadaného čísla. Tento výsledek si ale také můžeme představit rozdělený na N úseků délky $N+1$: nejnižší úsek bude obsahovat číslice $x_0x_{N-1} \ldots x_0$, ten nad ním $x_1x_0x_{N-1} \ldots x_1$ atd. Tedy i-tý úsek zespoda (počítáno od nuly) bude mít na nejnižším místě x_i.

Nyní každý úsek vyplníme hodnotou jeho nejnižšího bitu. K tomu stačí úsek upravit do tvaru $\mathbf{10}^{N-1}x_i$ (použijeme $\wedge$ a $\vee$) a odečíst od něj jedničku. Pokud x_i bylo $\mathbf{1}$, vyjde $\mathbf{10}^N$, jinak dojde k přenosu a dostaneme $\mathbf{01}^N$. V každém případě se na nejvyšším místě objeví x_i a na ostatních $\neg\, x_i$, což jedním *xor*em opravíme na x_i všude. Navíc nemohlo dojít k přenosu mimo úsek, takže se úseky navzájem neovlivňují, a tudíž tuto operaci můžeme provést pro všechny současně.

Nyní stačí vhodným *and*ováním v každém úseku ponechat x_i na pozici, na které má ve výsledku skončit, ostatní výskyty x_i vynulovat a všechny úseky sečíst. Sečtení můžeme provést pěkným trikem: pokud máme v registru r uloženo číslo tvaru $t_1 \ldots t_z$, kde t_i jsou k-bitová čísla, a spočteme $r \,\%(2^k - 1)$, vyjde $(t_1 + \ldots + t_z) \,\%(2^k - 1)$. V našem případě navíc víme, že $t_1 + \ldots + t_z < 2^k - 1$, protože sčítáme $(N + 1)$-bitové úseky s N-bitovým výsledkem, a tak se modulo na výsledku neprojeví a dostaneme přímo hledaný součet.

Proč modulo $2^k - 1$ funguje jako součet bloků, můžeme nahlédnout z toho, jak by se choval „školní“ algoritmus pro dělení čísel na papíře, ale také to můžeme snadno dokázat indukcí: pro $z = 1$ trik určitě funguje; když už víme, že funguje pro $z - 1$ a chceme dokázat, že funguje i pro z, všimneme si, že:

$$\begin{aligned} t_1 \ldots t_z \,\%(2^k - 1) &= \big((t_1 \ldots t_{z-1}) \cdot 2^k + t_z\big) \,\%(2^k - 1) = \\ &= \Big(\underbrace{\big(t_1 \ldots t_{z-1} \,\%(2^k - 1)\big)}_{\text{indukční předpoklad}} \cdot \underbrace{\big(2^k \,\%(2^k - 1)\big)}_{=1} + t_z \Big) \,\%(2^k - 1), \end{aligned}$$

což dá dohromady $(t_1 + \ldots + t_z) \,\%(2^k - 1)$, přesně, jak jsme chtěli.

Ke všem výpočtům jsme potřebovali registry délky $N \cdot (N + 1) = O(N^2)$. Operací bylo dohromady konstantně mnoho. Zbývá už jen program:

	$x = x_{N-1} \ldots x_0$
$y := x * (\mathbf{0}^{N-1}\mathbf{1})^{N+1}$	$y = (x_{N-1} \ldots x_0)^{N+1}$
	$= (x_{N-1} \ldots x_0x_{N-1}) \ldots (x_1x_0x_{N-1} \ldots x_1)(x_0x_{N-1} \ldots x_0)$
$y := y \wedge (\mathbf{0}^N\mathbf{1})^N$	$y = (\mathbf{0}^N x_{N-1}) \ldots (\mathbf{0}^N x_1)(\mathbf{0}^N x_0)$

$y := y \vee (\mathbf{10}^N)^N$
$y := y - (\mathbf{0}^N\mathbf{1})^N$
$y := y \oplus (\mathbf{01}^N)^N$
$y := y \wedge (\mathbf{0}^N\mathbf{1})(\mathbf{0}^{N-1}\mathbf{10})\dots(\mathbf{010}^{N-1})$
$y := y \,\%\, \mathbf{1}^{N+1}$

$y = (\mathbf{10}^{N-1}x_{N-1})\dots(\mathbf{10}^{N-1}x_1)(\mathbf{10}^{N-1}x_0)$
$y = (x_{N-1}(\neg\, x_{N-1})^N)\dots(x_0(\neg\, x_0)^N)$
$y = (x_{N-1}^{N+1})\dots(x_0^{N+1})$
$y = (\mathbf{0}^N x_{N-1})(\mathbf{0}^{N-1}x_{N-2}\mathbf{0})\dots(\mathbf{00}x_1\mathbf{0}^{N-2})(\mathbf{0}x_0\mathbf{0}^{N-1})$
$y = \mathbf{0}x_0x_1\dots x_{N-1}$