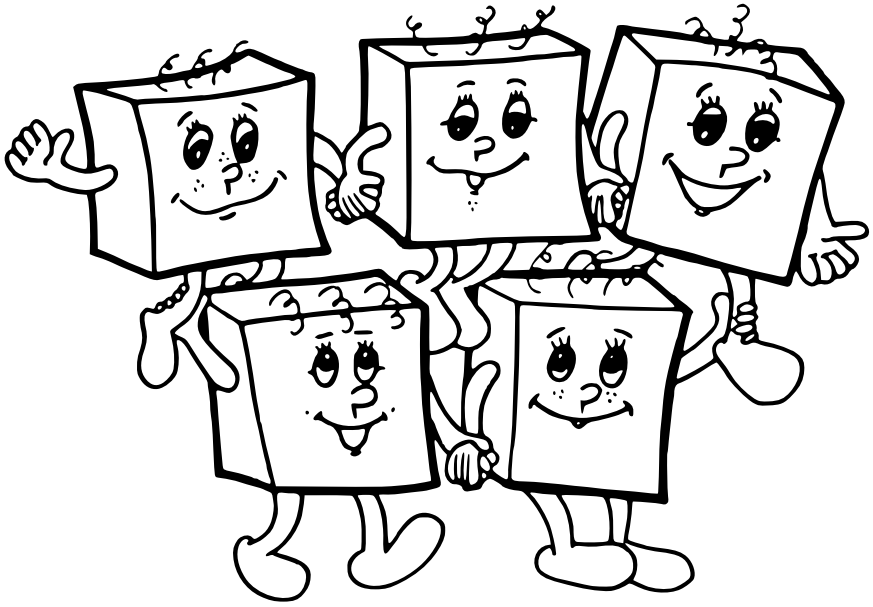
OLYMPIÁDA V INFORMATIKE
NA STREDNÝCH ŠKOLÁCH



dvadsiaty štvrtý ročník
školský rok 2008/09

riešenia domáceho kola
kategória A

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

Tento materiál obsahuje **riešenia** úloh. Jeho cieľom je pomôcť učiteľom na školách pri príprave riešiteľov domáceho kola. Taktiež slúži ako podklad pre opravovanie úloh členmi krajských výborov OI.

Žiakom možno tento materiál poskytnúť až po termíne stanovenom pre odovzdanie riešení úloh (17. novembra 2008). Po tomto termíne bude aj zverejnený na webstránke súťaže.

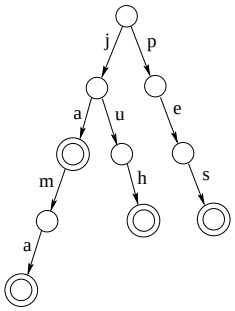
Riešenia kategórie A

A-I-1 Meníme históriu

Myšlienka riešenia bude jednoduchá – postupne budeme prechádzať riadky na vstupe a nahradzovať nevhodné slová vhodnými. Pri nahradzovaní slov si však musíme dať pozor, aby sme to robili efektívne.

Jedno možné efektívne riešenie je použiť dátovú štruktúru nazývanú písmenkový strom (po anglicky *trie*). Písmenkový strom je zakorenený strom, v ktorom každý vrchol má najviac 26 synov, a hrany do synov sú označené rôznymi písmenkami (od `a` po `z`). Každá cesta z koreňa nadol zodpovedá slovu, ktoré si „prečítame“ na hranách, po ktorých ideme.

Písmenkový strom sa dá použiť na uloženie množiny slov. Jednoducho vytvoríme všetky vrcholy, ktoré treba na to, aby sme si mohli na ceste z koreňa dole „prečítať“ každé z našich slov, a následne označíme tie vrcholy, kde niektoré z uložených slov končí. (Napríklad si do toho vrcholu napíšeme poradové číslo slova, ktoré sa tam končí.)



Písmenkový strom pre `ja`, `jama`, `juh` *a* `pes`.

Našu úlohu teda môžeme vyriešiť tak, že vytvoríme písmenkový strom pre zadané nevhodné slová. Pre každé spracovávané slovo zistíme, či sa v strome nachádza (teda či je nevhodné). Ak nie, nechávame ho bezo zmeny. Ak áno, nahradíme ho ekvivalentom.

Všimnime si, že pre slovo dĺžky d bude trvať vyhľadávanie v strome čas $O(d)$. Navyše výstup môže byť oveľa dlhší ako vstup. Totiž krátke nevhodné slovo môže byť nahradené dlhým vhodným slovom. Časová zložitosť je preto lineárna od veľkosti vstupu a výstupu. Čo sa pamäťovej zložitosti týka, stačí si pamätať len posledné načítané slovo a celý písmenkový strom aj s vhodnými náhradami. Teda pamäťová zložitosť je lineárna od veľkosti stromu, čiže od veľkosti slovníkovej časti vstupu.

Listing programu:

```
#include <iostream>
#include <string>
#include <vector>
using namespace std;

class trie {
    class __trieNode { // __trieNode obsahuje vrchol pismenkoveho stromu
        public:
            int endsHere; // cislo slova, ktore sa tu konci
            __trieNode *son[26]; // pointre na synov
            __trieNode() { endsHere=-1; memset(son,0,sizeof(son)); }
    };

    __trieNode *root; // v ramci pismenkoveho stromu si pamatame pointer na koren
    public:
    trie() { root = new __trieNode(); }
    void insert(const string &S, int cislo);
    int find(const string &S);
};

// funkcia na vlozenie noveho slova do stromu
```

```
void trie::insert(const string &S, int cislo) {
    __trieNode *kde = root;

    // ideme postupne po pismenach
    for (unsigned i=0; i<S.size(); i++) {
        int idx = S[i]-'A';
        // ak este takuto vetvu v strome nemame, vytvorime si ju
        if (! kde->son[idx]) kde->son[idx] = new __trieNode();
        // a presunieme sa o uroven hlbsie
        kde = kde->son[idx];
    }
    // v aktualnom vrchole zapiseme cislo slova
    kde->endsHere = cislo;
}

// funkcia na najdenie slova v strome
int trie::find(const string &S) {
    __trieNode *kde = root;

    // opaet ideme postupne po pismenach dole stromom
    for (unsigned i=0; i<S.size(); i++) {
        if (!kde) return -1;
        kde = kde->son[ S[i]-'A' ];
    }
    if (!kde) return -1;
    return kde->endsHere;
}

int main() {
    int N; cin >> N;
    vector<string> vhodne(N);
    trie T;
    // vytvorime pismekovy strom z nevhodnych slov
    for (int i=0; i<N; i++) {
        string nevhodne;
        cin >> nevhodne >> vhodne[i];
        T.insert(nevhodne, i);
    }
    // ideme opravovat text
    string s;
    getline(cin, s); // nacitame koniec riadku za poslednym vhodnym slovom
    while (getline(cin, s)) {
        for (unsigned i=0; i<s.size(); i++) {
            if (isalpha(s[i])) { //znak je pismeno, takze sa tu zacina slovo
                unsigned u = i+1;
                while (u < s.size() && isalpha(s[u])) u++; //najdeme koniec slova
                string slovo = s.substr(i, u-i);
                i = u - 1;
                int index = T.find(slovo);
                if (index == -1) cout << slovo; else cout << vhodne[index];
            }
            else cout << s[i]; // znak nie je pismeno, rovno ho vypiseme
        }
        cout << endl;
    }
}
```

A-I-2 Prechádzka po kanáloch

Vždy, keď sa v zadaní úlohy objaví otázka „aká je najkratšia cesta“ alebo „aký je najmenší počet krokov/operácií“, prirodzeným smerom úvah je zostrojiť si graf zodpovedajúci zadanej úlohe a následne v ňom nájsť najkratšiu cestu.

V úlohách, kde hľadáme cestu v bludisku, vrcholy grafu zodpovedajú rôznym situáciám, v ktorých sa osoba idúca bludiskom (v našom prípade Jacques) môže nachádzať.

Čo v našom prípade jednoznačne popisuje Jacquovu situáciu? Samozrejme, sú to súradnice políčka, na ktorom sa nachádza. To ale nestačí – musíme ešte uviesť, ktoré kľúče už má pri sebe a ktoré nie.

Zistili sme teda, že v každom okamihu sa Jacques nachádza v jednej z $16RS$ možných situácií. (Je na jednom z RS políčok, a má so sebou jednu z $2^4 = 16$ možných kombinácií kľúčov.) Vždy sa môže skúsiť pohnúť jedným zo štyroch smerov, a zakaždým je jednoznačne určené, do akej novej situácie sa dostane.

Skutočne teda môžeme reprezentovať náš problém ako graf, kde vrcholy sú jednotlivé situácie a orientované

hrany predstavujú kroky Jacquesa. V tomto grafe chceme nájsť (jednu ľubovoľnú) najkratšiu cestu zo začiatočnej situácie do ľubovoľnej situácie, kde Jacques stojí na políčku s pokladom.

Prehľadávanie do šírky

Štandardnou metódou na hľadanie najkratšej cesty v grafe s neohodnotenými vrcholmi je prehľadávanie do šírky. Princíp je veľmi jednoduchý. Budeme postupne „ofarbovať“ všetky situácie, do ktorých sa vieme dostať. Pre každú z nich si budeme pamätať, na koľko krokov sa do nej dá dostať, a z ktorej situácie sme sa tam prvýkrát dostali.

Budeme mať frontu, v ktorej čakajú situácie na spracovanie. Na začiatku zoberieme začiatočnú situáciu, nastavíme jej vzdialenosť na 0 a vložíme ju do fronty. Kým sa nám fronta nevyprázdni, opakujeme: Vyberieme z fronty situáciu. Postupne vyskúšame všetky (nanajvýš 4) možné ťahy, ktoré sa v danej situácii dajú spraviť. Dostaneme nové situácie. Tie, ktoré sme už predtým objavili (sú „ofarbené“), odignorujeme. Ostatné „ofarbíme“, nastavíme im potrebný počet krokov o jedno väčší ako má práve spracúvaná situácia, a vložíme ich do fronty na spracovanie.

Všimnime si, ako tento algoritmus prebieha. Najskôr spracujeme začiatočnú situáciu, potom postupne všetky, ktoré sa dajú dosiahnuť na jeden krok, potom všetky dosiahnuteľné na dva kroky, a tak ďalej. A práve vďaka tomu si môžeme byť istí, že akonáhle sa prvýkrát dostaneme do nejakej novej situácie, dostali sme sa do nej určite najmenším možným počtom krokov.

Keďže každú situáciu najviac jedenkrát vložíme do fronty, a spracovať situáciu vieme v konštantnom čase (keďže sú vždy len 4 možné ťahy), je časová zložitosť tohto algoritmu lineárna od počtu dosiahnuteľných situácií.

Implementácia

Najťažšou časťou tejto úlohy bola bezbolestná implementácia vyššie popísaného postupu.

V prvom rade je vhodné zvoliť si číslovanie políčok od 0. Potom môžeme jednoducho vypočítať nový riadok a stĺpec pri pohybe pomocou operácie modulo.

Ďalším problémom je, že vopred nevieme rozmery mapy, a mapa $1\,000\,000 \times 1\,000\,000$ sa nám do pamäte nezmestí. Priamočiare riešenie bolo použiť jednorozmerné pole, a celý popis situácie vždy zakódovať do jedného čísla. Ale výhodnejšie a omnoho prehľadnejšie je použiť dynamické polia – `vector` v C++, prípadne použiť `SetLength` vo FreePascale:

```
type pole = array of array of longint;
var R,S : longint;
    A : pole;
begin
   readln(R,S);
   setlength(A,R,S);
end.
```

Pri skúšaní všetkých možných smerov pohybu je dobré vyhnúť sa „copy and paste“ – načo mať štyri skoro rovnaké kusy programu, ak to zvládne jeden? A navyše hlavnou nevýhodou prístupu „copy and paste“ je, že veľmi pravdepodobne zabudnete na jednom mieste spraviť potrebné zmenu. A takéto chyby sa veľmi ťažko hľadajú.

V našom programe deklarujeme pomocné polia, ktoré popisujú smery, kam sa môžeme pohnúť:

```
// pomocné polia
int dr[] = {-1,1,0,0}, ds[] = {0,0,-1,1}; string ddir = "SJZV";

// sme na políčku (r,s), vyskúšame všetky možné smery:
for (int d=0; d<4; d++) {
  // riadok sa zmení o dr[d], stĺpec o ds[d], tento pohyb sa volá ddir[d]
  int nr=(r+dr[d]+R)%R, ns=(s+ds[d]+S)%S;
  // ...
}
```

Ďalej, v našej implementácii používame pri práci s dverami a kľúčmi bitové operácie pre zjednodušenie zápisu. Jednotlivé farby zodpovedajú v našom programe hodnotám 1, 2, 4 a 8. Teraz každé číslo od 0 do 15 zodpovedá

jednej množine kľúčov. (Hodnota 0 znamená, že nemáme žiadne kľúče, hodnota $13 = 1 + 4 + 8$ znamená, že máme kľúče prvej, tretej a štvrtej farby.)

Teraz napríklad keď stretneme dvere, vieme ľahko zistiť, či máme zodpovedajúci kľúč. (Zoberieme bitový doplnok čísla predstavujúceho kľúče, ktoré máme, a spravíme jeho bitový and s číslom dverí. Ak dostaneme výsledok 0, dvere vieme otvoriť, ak nie, nie.)

Listing programu:

```
#include <algorithm>
#include <iostream>
#include <vector>
#include <string>
#include <queue>
using namespace std;

typedef vector<int> pole1d;
typedef vector<pole1d> pole2d;
typedef vector<pole2d> pole3d;

string sdvere="CZMF", skluce="czmf";
int dr[] = {-1,1,0,0}, ds[] = {0,0,-1,1}; string ddir = "SJZV";

int main() {
  int R, S; cin >> R >> S;
  pole2d stena(R,S), poklad(R,S), dvere(R,S), kluce(R,S);
  int startR, startS, cielR, cielS, cielK;
  bool found=false;

  for (int r=0; r<R; r++) {
    string riadok; cin >> riadok;
    for (int s=0; s<S; s++) {
      if (riadok[s]=='#') stena[r][s]=1;
      if (riadok[s]=='$') poklad[r][s]=1;
      if (riadok[s]=='*') startR=r, startS=s;
      if ((int)sdvere.find(riadok[s])>=0) dvere[r][s] = 1 << sdvere.find(riadok[s]);
      if ((int)skluce.find(riadok[s])>=0) kluce[r][s] = 1 << skluce.find(riadok[s]);
    }
  }
  pole3d dist(R,pole2d(S,pole1d(16,987654321)));
  pole3d from(R,pole2d(S,pole1d(16,-1)));
  dist[startR][startS][0] = 0;
  queue<int> Q;
  Q.push(startR); Q.push(startS); Q.push(0);
  while (!Q.empty()) {
    int r = Q.front(); Q.pop();
    int s = Q.front(); Q.pop();
    int k = Q.front(); Q.pop();
    if (poklad[r][s]) { cielR=r; cielS=s; cielK=k; found=true; break; }
    for (int d=0; d<4; d++) {
      int nr=(r+dr[d]+R)%R, ns=(s+ds[d]+S)%S;
      int nk=k | kluce[nr][ns];
      if (stena[nr][ns]) continue;
      if (dvere[nr][ns] & ~nk) continue;
      if (dist[nr][ns][nk] <= dist[r][s][k]+1) continue;
      dist[nr][ns][nk]=dist[r][s][k]+1;
      from[nr][ns][nk]=16*R*S*d + R*S*k + S*r + s;
      Q.push(nr); Q.push(ns); Q.push(nk);
    }
  }
  if (!found) { cout << "nemozne" << endl; return 0; }
  string res = "";
  while (from[cielR][cielS][cielK] >= 0) {
    int f = from[cielR][cielS][cielK];
    int s=f%S; f/=S;
    int r=f%R; f/=R;
    int k=f%16; f/=16;
    res += ddir[f]; cielR=r; cielS=s; cielK=k;
  }
  reverse(res.begin(),res.end());
  cout << res << endl;
}
```

A-1-3 Horská dráha

Pomalé riešenia

Všetkých možných súvislých podpostupností je $O(N^2)$, pretože každá postupnosť začína na niektorom z N prvkov a končí na niektorom z nasledujúcich. Ak všetky možnosti priamočiaro vyskúšame, dostaneme čas $O(N^3)$, pretože zistenie súčtu pre nejakú podpostupnosť trvá $O(N)$ operácií.

Užitočná a často používaná finta, ktorou si môžeme pomôcť aj v tomto príklade, spočíva vo vytvorení poľa S (nazývaného *prefixové súčty*), v ktorom si budeme pre každý prvok pamätať súčet od začiatku postupnosti po tento prvok. (Navyše sa dohodneme, že $S[0] = 0$.)

Toto pole dokážeme získať postupným počítaním v lineárnom čase. A načo nám bude? V prípade, že potom budeme chcieť vedieť súčet prvkov $d_i, d_{i+1}, \ldots, d_j$, nebudeme potrebovať lineárny čas, ale len použijeme hodnotu $S[j] - S[i-1]$, teda súčet $d_1, \ldots, d_j$ mínus súčet $d_1, \ldots, d_{i-1}$. Takto dostaneme riešenie pôvodnej úlohy v kvadratickom čase.

Pomocná úloha

Pred ďalším vysvetľovaním spravíme krátku odbočku a zamyslíme sa nad nasledovnou úlohou: pre neklesajúcu postupnosť čísel $a_1, \ldots, a_N$ chceme zistiť, či v nej existujú dve čísla, ktorých rozdiel je $X, X > 0$. Úloha sa dá riešiť v lineárnom čase *metódou dvoch ukazovateľov*. Využijeme dve premenné, i a j, ktoré budú ukazovať na niektorý prvok postupnosti A, teda budú nadobúdať hodnoty indexov tejto postupnosti. Na začiatku bude ukazovateľ i nastavený na 1 a ukazovateľ j nastavený na 2. Ukazateľov budeme v texte niekedy nazývať aj *bežcami*, pretože sa na ne da pozrieť ako na dvoch ľudí, ktorí behajú po postupnosti a hľadajú riešenie.

Algoritmus bude postupovať tak, že si najprv pozrie hodnotu $a_j - a_i$. Môžu nastať tri prípady:

- $a_j - a_i$ je rovné X. Vtedy môžeme úspešne skončiť s odpoveďou *ÁNO*.
- $a_j - a_i$ je menšie ako X. Vtedy vieme, že ak existujú čísla s rozdielom X, potom musí to väčšie byť ďalej ako na indexe j. Preto budeme postupovať tak, že zvýšime ukazovateľ j o jedna. V prípade, že by j malo presiahnuť N, skončíme s odpoveďou *NIE*.
- $a_j - a_i$ je väčšie ako X. Vtedy vieme, že ak existujú dve čísla s rozdielom X, potom musí to menšie z nich byť ďalej ako na indexe i. Preto budeme postupovať tak, že zvýšime ukazovateľ i o jedna.

Prečo uvedeným správaním dostaneme korektné riešenie? Odpovedať môžeme pomocou matematickej indukcie ukázaním, že po k krokoch sme nemohli minúť riešenie, ktorého väčší prvok je v poli skorej ako aktuálna pozícia bežca j a zároveň sme nemohli minúť riešenie, ktorého pozícia menšieho prvku je skorej ako pozícia bežca i. Po nula krokoch to zjavne platí, pretože keďže $X > 0$, prvky, ktorých rozdiel hľadáme, musia byť rôzne a preto má ten väčší pozíciu aspoň 2. Uvažujme, že po k krokoch nastal prípad druhý prípad, teda $a_j - a_i < X$. Vieme, že žiadne riešenie nemá vyšší prvok skôr ako je aktuálna pozícia bežca j, obdobne neexistuje riešenie, ktorého nižší prvok je skôr ako pozícia čísla i. Takže ak existuje riešenie, najmenšie pozície môže mať i a j. Ale keďže $a_j - a_i < X$, potom indexy i a j nie sú riešením a vzhľadom k predpokladu musí prípadné riešenie mat vyšší prvok na pozícii minimálne $j + 1$. Preto môžeme príslušný ukazovateľ posunúť a uvedená vlastnosť bude platiť aj po $k + 1$ krokoch. Obdobne sa dokáže aj tretia možnosť. V prípade, že j presiahne rozsah poľa, naozaj môžeme skončiť s neúspechom, pretože prípadné riešenie by muselo končiť mimo postupnosti a teda nemôže existovať.

Časová zložitosť tohto postupu je $O(N)$, pretože každý krok trvá konštantný čas a v každom kroku posunieme jedného bežca, takže ich robíme najviac $2N$. Zamyslite sa, ako nám pri tomto riešení pomáha neklesajúcosť postupnosti $a_1, \ldots, a_N$.

Ľahkou modifikáciou uvedeného postupu sa dajú riešit aj iné úlohy. Jednou z nich je zistiť, či je v postupnosti nezáporných čísel súvislá podpostupnosť s nejakým zadaným súčtom. Toto nám pripomína úlohu zo zadania. A naozaj, keby sme sa obmedzili na nezáporne čísla, vedeli by sme napísať lineárne riešenie. So zápornými číslami sa ale tento postup nedokáže vysporiadať.

Riešenie v čase $O(N \log N)$

Vieme už, že keď si predpočítame pole S, vieme súčet úseku $d_i, \ldots, d_j$ vyjadriť ako $S[j] - S[i-1]$. Našu úlohu teda môžeme preformulovať nasledovne: Nájdite indexy a a b také, že $|S[a] - S[b]|$ je čo najbližšie k X. Keďže a a b môžeme medzi sebou vymeniť, stačí nájsť dvojicu a, b takú, že $S[a] - S[b]$ je čo najbližšie k X. A toto nám už začína pripomínať našu pomocnú úlohu.

Ak by sme mali len zistiť, či sa dá dosiahnuť $S[a] - S[b] = X$, už by sme vedeli ako na to. Stačilo by utriediť pole S podľa veľkosti a následne použiť metódu dvoch ukazovateľov. A teraz si už len stačí uvedomiť, že táto metóda dokonca bez akejkoľvek zmeny vyrieši aj našu všeobecnejšiu úlohu.

Prečo je to tak? Nech Y je najväčšia hodnota menšia ako X, ktorá sa dá dosiahnuť. Keby sme teraz pustili ten istý algoritmus, ale nechali ho hľadať Y namiesto X, jeho priebeh by bol celkom rovnaký – všetky porovnávania by dopadli rovnako, a teda by rovnako posúval ukazovatele. No a vyššie sme dokázali, že ak sa hodnota Y dá dosiahnuť, náš algoritmus to zistí. Inými slovami, keď budeme hľadať rozdiel X, určite bude medzi skúmanými dvojicami indexov aj tá dvojica, pre ktorú je rozdiel Y.

Podobne sa dá dokázať, že počas použitia metódy dvoch ukazovateľov nájdeme aj najmenšiu hodnotu Z, ktorá je väčšia ako X a dá sa dosiahnuť. A na vyriešenie našej úlohy stačí jednoducho vybrať tú lepšiu z týchto dvoch možností.

Ešte raz teda zhrnieme celé riešenie: Predpočítame si prefixové súčty do poľa S. Toto pole následne utriedime. Teraz v ňom chceme nájsť dve hodnoty $S[a]$ a $S[b]$ také, že $S[a] - S[b]$ je čo najbližšie k X. To spravíme tak, že na utriedené pole S pustíme algoritmus používajúci metódu dvoch ukazovateľov – postupne skracujeme alebo predlžujeme skúmaný interval, podľa toho, či je aktuálna hodnota rozdielu menšia alebo väčšia ako hľadané X. Nakoniec spomedzi všetkých dvojíc indexov, ktoré náš algoritmus skúmal, vyberieme tú najlepšiu a vypíšeme ju.

Poznámka k implementácii: Aby sme dokázali nájsť indexy k, l, ktoré nás vo výstupe zaujímajú, tak si ku každému prvku $S[i]$ zapamätáme aj index i, ktorý potom presúvame pri triedení spolu s hodnotou $S[i]$.

Predrátanie poľa S nás stojí čas $O(N)$. Triedenie pomocou efektívneho algoritmu trvá $O(N \log N)$ a algoritmus dvoch ukazovateľov potrebuje čas $O(N)$. Preto je výsledná časová zložitosť $O(N \log N)$, pamäťové nároky sú zjavne $O(N)$.

Iné riešenie v čase $O(N \log N)$

Namiesto metódy dvoch ukazovateľov stačilo použiť efektívnu dátovú štruktúru, do ktorej vieme vkladať prvky, a pre danú hodnotu q nájsť prvok, ktorého hodnota je najbližšia ku q. Pomocou takejto dátovej štruktúry vieme zadanú úlohu ľahko vyriešiť. Pripomeňme, že hľadáme indexy a a b také, že $S[a] - S[b]$ má byť čo najbližšie k X. Začneme tým, že do našej dátovej štruktúry vložíme všetky hodnoty $S[i]$. Teraz postupne vyskúšame všetky možné hodnoty a, a pre každú si v našej dátovej štruktúre nájdeme prvok s hodnotou najbližšou ku $S[a] - X$.

Programátori v C++ majú k dispozícii vyvážený binárny strom – triedu `set`. V Pascale jednoduchou alternatívou bolo utriediť pole S a binárne v ňom vyhľadávať. Oba prístupy dokážu hľadaný prvok nájsť v čase $O(\log N)$, a teda celková časová zložitosť takýchto riešení je $O(N \log N)$.

Listing programu:

```
#include <vector>
#include <iostream>
#include <algorithm>
using namespace std;


int main(){
    int N,X;
    cin >> N >> X;
    vector<int> A(N);
    for (int i=0;i<N;i++) cin >> A[i];

    // spocitame ciastkove sucty od zaciatku po kazdy prvok postupnosti
    // prva zlozka prvku S[i] je sucet od zaciatku postupnosti po prvok A[i-1]
    // druha zlozka prvku S[i] je samotny index i, ktoru potrebujeme pri vystupe
    vector<pair<int,int> > S(N+1);
    S.resize(N+1);
    S[0] = make_pair(0,0);
```

```
    for (int i=1;i<=N;i++) S[i] = make_pair( S[i-1].first+A[i-1], i );
    sort(S.begin(),S.end());

    //nastavime ukazovatele i,j na zaciatok
    int i=0, j=1, suc, besti=S[0].second, bestj=S[1].second, bestsuc=S[1].first-S[0].first;

    while ( i<= N && j<= N ){
        suc = S[j].first - S[i].first;
        //nasli sme lepsiu moznost ako doteraz najlepsiu?
        if (abs(X-suc) < abs(X-bestsuc)) { besti=S[i].second; bestj=S[j].second; bestsuc=suc; }
        //ak sme nasli najlepsiu moznu odpoved, mozeme skoncit
        if (X == suc) break;
        if (X > suc) j++; else i++;
        //prazdnu postupnost nepripustame
        if (i == j) j++;
    }
    if (besti > bestj) swap(besti,bestj);
    cout << besti+1 << " " << bestj << endl;
    return 0;
}
```

A-1-4 Zásobníkové počítače

Podúloha a

V tejto podúlohe sme mali dosiahnuť čo najlepšiu časovú zložitosť. Naše riešenie je lineárne a používa tri zásobníky. Menší počet operácií ako lineárny od dĺžky vstupu dosiahnuť zjavne nemôžeme, pretože na správnu odpoveď musíme načítať celé slovo.

Ak by sme mali dva zásobníky, z ktorých v jednom by bolo uložené vstupné slovo v poradí, ako sme ho načítali a v druhom by bolo uložené v opačnom poradí (teda na vrchu by bol znak, ktorý sme načítali ako prvý), potom by sa nám skutočnosť, či je vstup palindróm, overovala veľmi jednoducho – len by sme čítali znaky zo zásobíkov a pozerali sa, či sú rovnaké.

Riešenie začne tým, že načíta vstup, ktorý si zapamätá do dvoch zásobníkov – $S1$ a $S2$. Potom využije ďalší prázdny zásobník $S3$, do ktorého „presype“ jeden z naplnených – bude čítať znaky z prvého zásobníka a ukladať ich na druhý. Takto získame v zásobníku $S3$ slovo zo vstupu v opačnom poradí. A teraz už stačí len postupne overiť rovnosť všetkých dvojíc písmen.

Načítanie vstupu, preklopenie zásobníka aj overovanie trvá lineárny počet operácií od dĺžky vstupného slova.

Listing programu:

```
program palindrom1;
var S1,S2,S3: stack of char;
    c,d: char;
begin
   while read(c) do begin push(S1, c); push(S2, c); end;
   {naplnenie S3, aby v nom bol vstup opacne ako v S2}
   while not empty(S1) do begin c := pop(S1); push(S3,c); end;
   while not empty(S2) do begin
      c = pop(S2);
      d = pop(S3);
      if (c <> d) then begin write('0'); halt; end;
   end;
   write('1');
end.
```

Podúloha b

Naše riešenie bude používať dva zásobníky - $S1$ a $S2$. Pomocou nich najprv overí, či sa prvý znak zhoduje s posledným, potom či sa druhý zhoduje s predposledným a tak postupne až kým neskontroluje všetky protiľahlé dvojice písmen v slove.

Program na začiatku načíta celý vstup do zásobníka $S1$. Následne vyberie prvý znak a zapamätá si ho. Potom celý zvyšný obsah $S1$ presype do $S2$ - postupne číta znaky z vrchu $S1$ a ukladá ich na vrch zásobníka $S2$. Keď to spraví, je pripravený overiť si zhodu prvého a posledného znaku. Ak sa nerovnajú, môže hneď

prehlásiť, že slovo nie je palindróm. V prípade rovnosti znakov „presypeme“ celý zvyšok slova späť do $S1$ a celý postup začneme odznova.

Ak je dĺžka vstupného slova N, riešenie vykoná pre každú overovanú dvojicu $O(N)$ operácií. Overovaných dvojíc je približne $N/2$, preto je výsledná časová zložitosť $O(N^2)$.

Listing programu:

```
var S1,S2: stack of char;
    c,d: char;

begin
   { načítame vstup }
   while read(c) do push(S1,c);
   while true do begin
      { ak už nezostali žiadne písmená, je to palindróm }
      if empty(S1) then begin write(1); halt; end;
      { zoberieme písmeno z vrchu S1, zvyšok S1 presypeme do S2 }
      c := pop(S1);
      while not empty(S1) do begin d:=pop(S1); push(S2,d); end;
      { ak je S2 prázdny, ostalo len jedno písmeno, je to palindróm }
      if empty(S2) then begin write(1); halt; end;
      { zoberieme písmeno z vrchu S2, porovnáme so zapamätaným z opačného konca }
      d := pop(S2);
      if c<>d then begin write(0); halt; end;
      { "presypeme" celý obsah S2 späť a celý postup opakujeme }
      while not empty(S2) do begin d:=pop(S2); push(S1,d); end;
   end;
end.
```

SLOVENSKÁ KOMISIA OLYMPIÁDY V INFORMATIKE

DVADSIATY ŠTVRTÝ ROČNÍK OLYMPIÁDY V INFORMATIKE

Vydala IUVENTA s finančnou podporou Ministerstva školstva SR
Náklad: 400 výtlačkov
Zodpovedný redaktor: Michal Forišek
Sadzba programom LaTeX