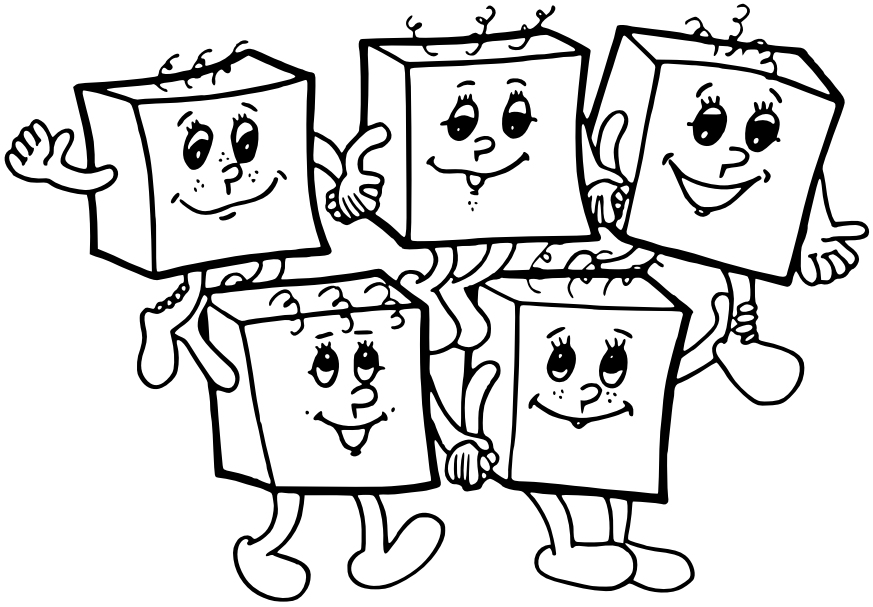
OLYMPIÁDA V INFORMATIKE
NA STREDNÝCH ŠKOLÁCH



dvadsiaty štvrtý ročník
školský rok 2008/09

riešenia domáceho kola
kategória B

- **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 (1. decembra 2008). Po tomto termíne bude aj zverejnený na webstránke súťaže.

Riešenia kategórie B

B-I-1 Diaľnica

Pomalšie riešenia

Najjednoduchšie, priamočiare riešenie spočíva v prehľadaní všetkých dvojíc čerpacích staníc. Pre každú dvojicu staníc si vypočítame ich vzdialenosť, a ak je menšia alebo rovná K, zväčšíme si počítadlo.

```
P := 0;
for i := 1 to N do
   for j := i + 1 to N do
      if(a[j] - a[i] <= K) then P := P + 1;
writeln(P);
```

Všetkých dvojíc staníc je $N(N-1)/2$, čo je rádovo N^2. Preto má takéto riešenie časovú zložitosť $O(N^2)$. Malo získať aspoň 5 bodov.

Ak sa podrobnejšie pozrieme na predchádzajúce riešenie, všimneme si, že konštrukcia dvoch cyklov `for` nám generuje všetky dvojice čerpacích staníc – teda aj tie, ktorých vzdialenosť je väčšia ako K. Zjavne ale platí, že akonáhle už pre nejaké j je stanica j príliš daleko od stanice i, tak aj všetky stanice od $j+1$ po N sú už pridaleko. Vnútorný cyklus teda môžeme prerušiť akonáhle vzdialenosť od stanice i presiahne hodnotu K.

```
P := 0;
for i := 1 to N do
   for j := i + 1 to N do
      if(a[j] - a[i] <= K) then P := P + 1
      else break;
writeln(P);
```

Tým dostaneme riešenie, ktorého časová zložitosť je $O(N+P)$, kde P je hodnota výsledku. Takéto riešenie je lepšie od predchádzajúceho pre „riedke“ vstupy, avšak v najhoršom prípade (ak sú každé dve stanice vo vzdialenosti $\leq K$) je jeho časová zložitosť nadalej kvadratická od N. Toto riešenie malo získať aspoň 7 bodov.

Vzorové riešenie

Ak chceme riešenie, ktoré bude mať vždy lepšiu ako kvadratickú časovú zložitosť, nesmieme sa „dotknúť“ všetkých hľadaných dvojíc, kedže tých môže byť až kvadraticky veľa.

Čo ešte vieme na predchádzajúcom riešení zlepšiť? Nech m_i je najväčšie číslo stanice, ktorá je dosiahnuteľná zo stanice i. Keď teraz budeme spracúvať stanicu $i+1$, určite budú aj z nej dosiahnute stanice $i+2$ až m_i, lebo sú k nej bližšie ako ku stanici i. Tieto dvojice staníc teda nemusíme kontrolovať a môžeme ich rovno započítať do výsledku.

Iný možný pohľad na toto isté pozorovanie: Práve sme zdôvodnili, že platí $m_{i+1} \geq m_i$. Zjavne platí, že hodnota $m_i - i$ udáva počet staníc, ktoré sú napravo od i a sú z nej dosiahnuteľné. Preto výsledkom našej úlohy je $P = \sum_{i=1}^{n} m_i - i$. Na to, aby sme zistili P, teda stačí spočítať hodnoty m_i.

Listing programu:

```
const maxN = 210000;
var N,K,P,i : longint;
    a,m : array[0..maxN] of longint;
begin
   read(K); read(N); for i:=1 to N do read(a[i]);
   P := 0;
   m[0] := 1;
   for i:=1 to N do begin
      m[i] := m[i-1];
      while (m[i] < N) and (a[ m[i]+1 ] - a[i] <= K) do inc(m[i]);
      P := P + m[i] - i;
   end;
   writeln(P);
end.
```

Všimnime si, že aktuálna hodnota m_i sa v priebehu programu nikdy nezmenší. Preto sa príkaz `inc(m[i])` počas celého programu vykoná najviac $N-1$ ráz. A preto je aj celková časová zložitosť tohto riešenia lineárna od N.

(Na riešenie sa môžeme dívať tak, že si na pole ukazujeme dvoma prstami – ľavým na práve spracúvanú stanicu, pravým na stanicu, s ktorou ju práve porovnávame. Počas behu programu prejdeme oboma prstami po poli zľava doprava, dokopy teda pohneme prstom menej ako $2N$-krát.)

B-I-2 Kolotoč

Najjednoduchšie riešenie úlohy spočívalo jednoducho v implementácii postupu zo zadania: Napíšeme si procedúru, ktorá celý kolotoč otočí o jedno miesto, a následne ju K-krát zavoláme. Takéto riešenie má časovú zložitosť $O(KN^2)$, keďže K-krát posunieme každé z N^2 detí. Správny program s touto myšlienkou mal získať aspoň 5 bodov.

Prvé možné zlepšenie je všimnúť si, ako vlastne ten kolotoč otáčame. Kolotoč sa skladá z približne $N/2$ „obručí“, ktoré sa točia nezávisle na sebe. Všimnime si napríklad vonkajšiu obruč kolotoča $N \times N$. Tú tvorí $4N-4$ sedadiel. To ale znamená, že každých $4N-4$ sekúnd budú deti, ktoré sú na tejto obruči, sedieť presne na tých miestach ako na začiatku.

Ak je počet otočení K výrazne väčší ako N, vieme vďaka tomuto pozorovaniu dosť práce ušetriť: pre každú obruč si spočítame jej dĺžku D, a následne ju otočíme (K mod D)-krát.

Pri tomto riešení posunieme každé dieťa najviac $(4N-5)$-krát, celková časová zložitosť je teda $O(N^3)$. Program s touto myšlienkou mal získať aspoň 7 bodov.

Vzorové riešenie ide ešte o krok ďalej: keď máme obruč a vieme, koľkokrát ju chceme otočiť, môžeme rovno pre každé dieťa spočítať, kde bude sedieť po danom počte otočení.

Presnejšie, očíslujme si pozície na danej obruči číslami od 0 do $D-1$ v smere točenia. Dieťa, ktoré teraz sedí na pozícii x, bude po K otočeniach sedieť na pozícii $(x+K)$ mod D. Stačí si teda spraviť nové pole a do neho postupne vypĺňať deti na miesta, kde budú sedieť na konci.

Pri tomto riešení každé dieťa rovno umiestnime na pozíciu, kde bude na konci. Celková časová zložitosť je teda $O(N^2)$. Program s touto myšlienkou mal získať 10 bodov.

V nasledujúcom listingu programu si všimnite elegantný spôsob implementácie funkcie `otoc_obruc`: do polí `row` a `col` si vyplníme súradnice políčok obruče, a následne pomocou tejto informácie hravo vyplníme túto obruč vo výstupnom poli.

Listing programu:

```
const maxN = 1024;
var vstup, vystup : array[0..maxN,0..maxN] of string[8];
    N,K,i : longint;

procedure nacitaj;
var riadok : string;
    zac,kon,row,col,i : longint;
begin
   readln(N);
   for row:=0 to N-1 do begin
      { nacita a spracuje riadok mien }
      readln(riadok); riadok:=riadok+' ';
      zac:=1; kon:=1;
      for col:=0 to n-1 do begin
         while (riadok[kon]<>' ') do inc(kon);
         vstup[row][col]:='';
         for i:=zac to kon-1 do vstup[row][col]:=vstup[row][col]+riadok[i];
         zac:=kon+1; kon:=kon+1;
      end;
   end;
   readln(K);
end;

procedure otoc_obruc(roh : longint);
var s,D,i,j : longint;
    row,col : array[0..4*maxN] of longint;
```

```
begin
   s:=n-2*roh-1; D:=4*s;
   if D=0 then begin D:=1; row[0]:=roh; col[0]:=roh; end;
   for i:=0 to s-1 do begin row[0*s+i]:=roh;     col[0*s+i]:=roh+i;   end;
   for i:=0 to s-1 do begin row[1*s+i]:=roh+i;   col[1*s+i]:=roh+s;   end;
   for i:=0 to s-1 do begin row[2*s+i]:=roh+s;   col[2*s+i]:=roh+s-i; end;
   for i:=0 to s-1 do begin row[3*s+i]:=roh+s-i; col[3*s+i]:=roh;     end;
   for i:=0 to D-1 do begin
      j:=(i+K) mod D;
      vystup[ row[j] ][ col[j] ] := vstup[ row[i] ][ col[i] ];
   end;
end;

procedure vypis;
var i,j : longint;
begin
   for i:=0 to N-1 do begin
      for j:=0 to N-1 do write(vystup[i][j]:7); writeln;
   end;
end;

begin
   nacitaj;
   for i:=0 to (N-1) div 2 do otoc_obruc(i);
   vypis;
end.
```

Uvedieme ešte veľmi stručnú implementáciu toho istého algoritmu v jazyku C++.

Listing programu:

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

int main() {
  int N,K;
  cin >> N;
  vector< vector<string> > vstup(N,N), vystup(N,N);
  for (int i=0; i<N; i++) for (int j=0; j<N; j++) cin >> vstup[i][j];
  cin >> K;
  for (int roh=0; roh<(N+1)/2; roh++) {
    int s=N-2*roh-1, D=max(1,4*s);
    vector<int> row(D), col(D);
    if (D==1) row[0] = col[0] = roh;
    for (int i=0; i<s; i++) row[0*s+i]=roh,     col[0*s+i]=roh+i;
    for (int i=0; i<s; i++) row[1*s+i]=roh+i,   col[1*s+i]=roh+s;
    for (int i=0; i<s; i++) row[2*s+i]=roh+s,   col[2*s+i]=roh+s-i;
    for (int i=0; i<s; i++) row[3*s+i]=roh+s-i, col[3*s+i]=roh;
    for (int i=0; i<D; i++) vystup[ row[(i+K)%D] ][ col[(i+K)%D] ] = vstup[ row[i] ][ col[i] ];
  }
  for (int i=0; i<N; i++) { for (int j=0; j<N; j++) cout << setw(7) << vystup[i][j]; cout << endl; }
}
```

B-I-3 Balíčky

Pre vyriešenie úlohy stačí zistiť, ktoré balíčky budú a ktoré nebudú nainštalované. Celkovú veľkosť už potom zistíme ľahko.

Vytvorme si teda pole `nainstalovany[]` typu boolean, pričom cieľom nášho riešenia bude nastaviť hodnotu `nainstalovany[i]` na `true`, ak balíček musíme inštalovať, a na `false` inak. Na začiatku všetky hodnoty v poli `nainstalovany` nastavíme na `false`.

Úlohu budeme riešiť pomocou rekurzie. Pre inštaláciu i-teho balíčka si vytvoríme funkciu `instaluj(i)`. Táto funkcia bude mať za úlohu zabezpečiť to, aby všetky balíčky potrebné na fungovanie balíčka i mali priradenú hodnotu v poli `nainstalovany` na `true`. V nasledujúcom kóde predpokladáme, že závislosti balíčka i sú uložené v poli `zavislost[i]`, na pozíciách 1 až `p[i]`.

```
   procedure instaluj(i : longint);
   var j : longint;
   begin
```

```
      nainstalovany[i] := true;
      for j := 1 to p[i] do
         if( not nainstalovany[zavislost[i][j]] ) then instaluj(zavislost[i][j]);
   end;
```

Pre vyriešenie našej úlohy teraz postupne zavoláme funkciu `instaluj` pre každý balíček, ktorý chce používateľ.

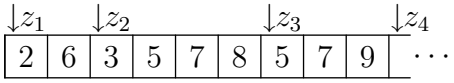
Poďme teraz analyzovať toto riešenie. Hodnota `nainstalovany[i]` sa počas behu programu nastaví na `true` práve vtedy, ak sa počas behu zavolá funkcia inštaluj s parametrom i. Všimnime si ale, že počas behu celého programu sa funkcia `instaluj` zavolá s parametrom i maximálne raz. (Akonáhle ju zavoláme, nastaví sa `nainstalovany[i]` na `true`, vďaka čomu ju už nikdy nezavoláme znova.)

Pri volaní procedúry `instaluj` s parametrom i vykonáme jedno priradenie, a následne postupne spracujeme všetkých p_i balíčkov, na ktorých balíček i závisí. Dokopy teda spravíme najviac N zmien v poli `nainstalovany`, a pri kontrolách závislostí spracujeme dokopy nanajvýš $\sum_{i=1}^{n} p_i$ balíčkov. Preto bude časová zložitosť nášho riešenia $O(N + \sum_{i=1}^{n} p_i)$. Už na načítanie vstupných údajov potrebujeme takýto čas, a preto je toto riešenie optimálne.

Poslednou vecou, ktorou sa v našom riešení potrebujeme zaoberať, je dátová štruktúra, ktorú použijeme na uchovávanie závislostí. Pre každý balíček si potrebujeme zapamätať zoznam balíčkov, na ktorých závisí. Tento zoznam ale môže obsahovať až $N - 1$ balíčkov. Ak by sme si tento zoznam chceli pamätať v poli dĺžky N, pre každé i, dostali by sme maticu veľkosti $N \times N$. Takéto riešenie by nám ale zvýšilo pamäťovú (a v závislosti od implementácie možno aj časovú) zložitosť na $O(N^2)$.

Na vyriešenie tohoto problému máme niekoľko možností.

Prvou možnosťou je dynamická alokácia pamäti – vytvoriť pre každý balíček pole dĺžky rovnej počtu jeho závislostí. Druhým možným riešením by bolo na uloženie závislostí implementovať vhodnú vlastnú dátovú štruktúru (napr. spájaný zoznam). Treťou možnosťou je zapamätať si závislosti všetkých balíčkov v jedinom poli s. Pre každý balíček i si okrem počtu jeho závislostí p_i budeme tiež pamätať index z_i, ktorý bude určovať, kde v poli s začína zoznam závislostí pre balíček i. Čísla balíčkov, na ktorých závisí balíček i sa teda budú nachádzať na pozíciách $z_i, z_i + 1, z_i + 2, \ldots, z_i + p_i - 1$. Nasledujúci obrázok vykresľuje nami predstavené riešenie, kde prvý balíček závisí na balíčkoch 2 a 6, druhý balíček závisí na balíčkoch 3,5,7,8 a tretí balíček závisí na balíčkoch 5,7,9.



Implementáciu takejto štruktúry používame aj v našom programe.

A ešte otázka na záver: Fungovalo by toto riešenie aj vtedy, keby mohli byť v zozname závislostí aj cykly?

Listing programu:

```
var N,M,i,j,q:longint;
    c:char;
    v,p,z:array[0..10000] of longint;     { veľkosti balíčkov, počty závislostí, začiatky v poli s }
    s:array[1..100000] of longint;        { pole obsahujúce závislosti všetkých balíčkov }
    nainstalovany:array[1..10000] of boolean;
    vysledok:longint;

procedure instaluj(i : longint);
var j : longint;
begin
   nainstalovany[i] := true;
   for j := 1 to p[i] do
      if ( not nainstalovany[ s[z[i]+j-1] ] ) then instaluj( s[z[i]+j-1] );
end;

begin
   readln(N);
   q := 1;                                { q určuje prvú ešte nezaplnenú pozíciu v poli s }
   for i := 1 to N do begin
      repeat read(c); until c=' ';        { prečítame názov balíčka a odignorujeme ho :-) }
      read(v[i],p[i]);
      z[i] := q;
      for j := 1 to p[i] do begin         { uložíme závislosti nášho i-teho balíčka do poľa s }
         read(s[q]);
```

```
         inc(q);
      end;
      readln;
   end;
   for i := 1 to N do nainstalovany[i] := false;
   readln(M);
   for j := 1 to M do begin
      read(i);
      instaluj(i);
   end;
   vysledok:=0;
   for i := 1 to N do if (nainstalovany[i]) then vysledok := vysledok + v[i];
   writeln(vysledok);
end.
```

B-I-4 Divné jazyky

Podúloha a)

Túto podúlohu bolo najľahšie vyriešiť vypísaním všetkých možností. Oplatí sa pritom byť systematický – môžeme odpozorovať závislosti, ktoré nám pomôžu neskôr.

Vypíšme teda všetky symetrické slová (odborne sa im hovorí *palindrómy*) s počtom 1 až 7 písmen:

- A, U
- AA, UU
- AAA, AUA, UAU, UUU
- $AAAA$, $AUUA$, $UAAU$, $UUUU$
- $AAAAA$, $AAUAA$, $AUAUA$, $AUUUA$, $UAAAU$, $UAUAU$, $UUAUU$, $UUUUU$
- $AAAAAA$, $AAUUAA$, $AUAAUA$, $AUUUUA$, $UAAAAU$, $UAUUAU$, $UUAAUU$, $UUUUUU$
- $AAAAAAA$, $AAAUAAA$, $AAUAUAA$, $AAUUUAA$, $AUAAAUA$, $AUAUAUA$, $AUUAUUA$, $AUUUUUA$, $UAAAAAU$, $UAAUAAU$, $UAUAUAU$, $UAUUUAU$, $UUAAAUU$, $UUAUAUU$, $UUUAUUU$, $UUUUUUU$

Celkový počet vypísaných slov je 44.

Podúloha b)

Všimnime si počty symetrických slov pre jednotlivé dĺžky: máme 2 slová dĺžky 1, 2 slová dĺžky 2, 4 slová dĺžky 3, atď. Dostávame takto postupnosť: 2, 2, 4, 4, 8, 8, 16, ...

Pomerne ľahko sa dá uhádnuť, ako bude táto postupnosť pokračovať: 16, 32, 32, 64, 64, ... Tvrdíme teda, že aj symetrických slov s $2k-1$ písmenami, aj symetrických slov s $2k$ písmenami je práve 2^k.

Nasledujúcim krokom riešenia je toto pozorovanie nejak dokázať. Jeden možný dôkaz: Predstavte si, že ideme napísať symetrické slovo tvorené $2k$ písmenami. Akonáhle napíšeme prvých k písmen, je vždy práve jedna správna možnosť, ako doplniť zvyšných k – musíme napísať tie isté písmená v opačnom poradí. No a pri výbere každého z prvých k písmen máme dve možnosti (A alebo U). Preto existuje 2^k spôsobov, ako napísať prvých k písmen, a teda existuje práve 2^k symetrických slov dĺžky $2k$.

Tento dôkaz môžeme povedať aj inými slovami: Všetky symetrické slová dĺžky $2k$ vieme vyrobiť tak, že zoberieme úplne všetky slová dĺžky k (ktorých je zjavne 2^k) a za každé slovo dopíšeme dotyčné slovo odzadu.

(Rozmyslite si, čo sa v tomto dôkaze zmení, ked budeme dokazovať, že aj počet symetrických slov s $2k-1$ písmenami je 2^k.)

Riešenie tejto podúlohy je teda $B = 2+2+\cdots+2^{23}+2^{23}+2^{24} = 50331644$. Pre zaujímavosť podotkneme, že pomocou známeho vzorca pre súčet geometrického radu ľahko vyjadríme, že $B = 2^{25}+2^{24}-4$.

Podúloha d)

Kmeň Strapatých má naozaj podivuhodný jazyk. Všetky slová v ich jazyku majú presne 7 hlások. Navyše platí, že ak sa domorodec z tohto kmeňa pomýli v jednej hláske ľubovoľného slova, aj tak sa dá spoznať, ktoré slovo chcel povedať. Dokážeme, že tento jazyk môže mať najviac **16 slov**.

Ku každému slovu S v jazyku kmeňa Strapatých existuje práve 7 reťazcov, ktoré sa od neho líšia v práve jednom znaku. Týchto 7 reťazcov nazveme *kamarátmi* slova S.

Všimnime si, že medzi kamarátmi slova S nemôže byť ani žiadne iné slovo, ani kamarát žiadneho iného slova.

Predstavme si teraz, že by sme zobrali list linajkového papiera a vypísali naň slová nášho jazyka, každé do nového riadku. Následne za každé slovo dopíšeme jeho 7 kamarátov.

Je zjavné, že na našom papieri nemôže byť žiaden reťazec uvedený dvakrát.

Všetkých reťazcov dĺžky 7 je $2^7 = 128$. V každom riadku je uvedených 8 reťazcov. Preto riadkov môže byť najviac $128/8 = 16$. Ale počet riadkov je práve rovný počtu slov v našom jazyku. Tým sme dokázali, že takýto jazyk môže mať nanajvýš 16 slov.

Podúloha c)

Zoraďme si všetkých $2^7 = 128$ reťazcov podľa abecedy. V tomto poradí ich budeme prechádzať. Pre každý reťazec otestujeme, či ho môžeme pridať do jazyka (t. j., či sa od každého už vybraného slova líši v aspoň troch znakoch), a ak áno, pridáme ho tam.

V tomto okamihu si treba uvedomiť, že nie je zaručené, že nám takýto postup nájde najväčší možný jazyk. Teoreticky by sa mohlo stať, že keby sme niektorý reťazec preskočili, umožnilo by nám to neskôr pridať viac iných reťazcov. Výhodou uvedeného „pažravého“ postupu však je, že sa veľmi ľahko naprogramuje, a teda určite neuškodí vyskúšať ho. Dostaneme tento jazyk:

AAAAAAA, *AAAAUUU*, *AAUUAAU*, *AAUUUUA*, *AUAUAUA*, *AUAUUAU*, *AUUAAUU*, *AUUAUAA*, *UAAUAUU*, *UAAUUAA*, *UAUAAUA*, *UAUAUAU*, *UUAAAAU*, *UUAAUUA*, *UUUUAAA*, *UUUUUUU*.

Zostrojili sme teda jazyk tvorený 16 slovami. Z výsledku podúlohy d) vieme, že lepšie to nejde. Môžeme teda spokojne prehlásiť, že sme našli najväčší možný jazyk spĺňajúci podmienky zo zadania.

V programe sme pre jednoduchšiu manipuláciu namiesto hlások A a U použili hodnoty 0 a 1. Takto sa nám 7-znakové reťazce zmenia na binárne zápisy čísel $0000000 = 0$ až $1111111 = 127$.

Ako súčasť riešenia uvádzame aj druhý program, založený na metóde prehľadávania s návratom (backtrackingu), ktorý postupne nájde všetky jazyky spĺňajúce uvedenú podmienku a vyberie najväčší z nich. Tento program vznikol jednoduchou úpravou prvého riešenia – vždy, keď nájdeme slovo, ktoré môžeme pridať do jazyka, vyskúšame postupne obe možnosti, aj pridať ho, aj nepridať. U tohto programu máme istotu, že nájde najväčšie možné riešenie.

Listing programu:

```
program b_1_4_greedy;

var slova : array[1..16] of longint;
    N, i : longint;

function vzdialenost(s1, s2 : longint) : longint;
var vysledok, i : longint;
begin
   vysledok := 0;
   for i:=0 to 6 do { porovname i-ty bit cisel s1 a s2 }
      if (s1 and (1 shl i)) <> (s2 and (1 shl i)) then inc(vysledok);
   vzdialenost := vysledok;
end;

function mozeme_pouzit(slovo : longint) : boolean;
var i : longint;
begin
   mozeme_pouzit := true;
   for i := 1 to N do if vzdialenost(slovo,slova[i]) <= 2 then mozeme_pouzit := false;
end;

procedure vypis(slovo : longint);
var i : longint;
begin
   for i:=6 downto 0 do if (slovo and (1 shl i)) > 0 then write('U') else write('A');
   writeln;
end;

begin
   N := 0;
```

```
   for i := 0 to 127 do if mozeme_pouzit(i) then begin inc(N); slova[N] := i; end;
   for i:=1 to N do vypis(slova[i]);
end.
```

Listing programu:

```
program b_1_4_backtracking;

var teraz, najlepsie : array[1..16] of longint;
    N, najlepsieN, i : longint;

function vzdialenost(s1, s2 : longint) : longint;
var vysledok, i : longint;
begin
   vysledok := 0;
   for i:=0 to 6 do { porovname i-ty bit cisel s1 a s2 }
      if (s1 and (1 shl i)) <> (s2 and (1 shl i)) then inc(vysledok);
   vzdialenost := vysledok;
end;

function mozeme_pouzit(slovo : longint) : boolean;
var i : longint;
begin
   mozeme_pouzit := true;
   for i := 1 to N do if vzdialenost(slovo,teraz[i]) <= 2 then mozeme_pouzit := false;
end;

procedure skus;
var posledne, nove, i : longint;
begin
   posledne := teraz[N];
   for nove := posledne+1 to 127 do
      if mozeme_pouzit(nove) then begin
         { pridame nove slovo do jazyka }
         inc(N); teraz[N] := nove;
         { pozrieme ci sme nasli vacsi jazyk ako doteraz najvacsi }
         if N > najlepsieN then begin
            najlepsieN := N;
            for i := 1 to N do najlepsie[i] := teraz[i];
         end;
         { skusime vsetky sposoby ako pridat dalsie slova }
         skus;
         { opat toto slovo odoberieme a skusime namiesto neho pridat ine }
         dec(N);
      end;
end;

procedure vypis(slovo : longint);
var i : longint;
begin
   for i:=6 downto 0 do if (slovo and (1 shl i)) > 0 then write('U') else write('A');
   writeln;
end;

begin
   N := 1;
   teraz[1] := 0;
   najlepsieN := 0;
   skus;
   for i:=1 to najlepsieN do vypis(najlepsie[i]);
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