Korešpondenčný seminár z programovania
XXIII. ročník, 2005/06
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

KSP finančne podporujú: aSc – Applied Software Consultants spol. s r.o.
Gnome spol. s r.o.
MICROSTEP-MIS spol. s r.o.

Vzorové riešenia 2. kola letnej časti

Milé naše riešiteľky, milí naši riešitelia!

Letné prázdniny sú už na dohľad, tak sme pre vás opravili poslednú sériu KSP. Taktiež sme pre vás aj vzorové riešenia napísali. Tak si ich chytro bežte prečítať a potom hor sa na prázdniny. Dúfame ale, že na nás nezabudnete a budete nás riešiť aj v ďalšom školskom roku. A my sľubujeme, že pre vás pripravíme zaujímavé príklady, aby ste vy zase mali čo riešiť. Tak už prestaň čítať tento úvod a začni čítať radšej vzoráky `:-)`

Ak to nepokazí júl a august, tak tento rok bola pekná zima

KSPáci

1. O guličkách a tuneloch

opravoval Mic
(max. 15 bodov)

Takže deti, poďme si povedať, ako som bodoval.

- 15 bodov dostal ten, čo riešenie podobné vzoráku napísal
- 11 bodov som udeľoval tým jedincom, čo riešenie bežiace v čase $O(N^2)$ vymysleli
- menej bodov som udeľoval za ostatné riešenia
- samozrejme, že som strhával aj za nedostatočný popis, program, zlý odhad časovej zložitosti a podobne

Bodovanie máme, čo znamená, že sa môžeme pustiť do riešenia. To bolo preveľmi jednoduché. Poďme si najskôr transformovať našu úlohu tak, aby sme nehovorili o tuneloch, ale o niečom rozumnejšom. Samozrejme, že si bludisko transformujeme na graf. Rozdvojky, roztrojky, atď. začneme nazývať vrcholy. Hrana bude viesť z vrcholu a do vrcholu b práve vtedy, keď bude vedieť guličkа prejsť z a do b. Keďže rúrky sú jednosmerné, tak aj hrany budú jednosmerné. Dokonca aj v zadaní je napísané, že ten graf bude acyklický. Čo to ale znamená?

Zoberme si ľubovoľný vrchol v. Aký je z neho počet ciest do vrcholu N? Vezmime všetky vrcholy, do ktorých vedie z vrcholu v hrana (to budú jeho „potomkovia“), i-ty z nich bude u_i. Označme C_i počet ciest z vrcholu u_i do vrcholu N. Zoberme teraz všetky možné cesty z vrcholu v. Tie prirodzene idú cez nejaký z vrcholov u_i. Ak dve cesty idú cez rôzne vrcholy, tak to znamená, že tie cesty sú nutne rôzne. To znamená že počet ciest z vrcholu v do vrcholu N je súčtom počtov ciest z vrcholu v do N cez vrcholy u_i. Ale počet ciest z vrcholu v cez vrchol u_i do vrcholu N je C_i. To znamená, že počet ciest z vrcholu v do vrcholu N je $\sum_i C_i$.

Takže teraz máme už dostatočné znalosti na to, aby sme vedeli napísať efektívne riešenie. Ako? Najskôr si spravme také hlúpe riešenie, pomocou rekurzie. Naša rekurzívna funkcia bude mať ako parameter vrchol v a vráti počet ciest z vrcholu v do vrcholu N. Ako to spraví? Tak, ako je vyššie napísané. Ak sa zavoláme na vrchol s číslom N, vrátime 1, v inom prípade sa zavoláme na všetkých potomkov vrcholu v a vrátime súčet návratových hodnôt. Toto nám dáva algoritmus, ktorý síce raz skončí, ale nebude príliš rýchly. Dokonca bude veľmi pomalý. Ako to napraviť? Najskôr si musíme uvedomiť, čo je na pôvodnom prístupe pomalé.

My sa na každý vrchol zavoláme niekoľko(veľa)-krát. Nie je ale zbytočné niekoľkokrát začať rátať to isté? Je, a preto si vždy, keď vypočítame počet ciest pre daný vrchol, ten počet zapamätáme a keď sa druhýkrát zavoláme, tak iba vrátime zapamätanú hodnotu. Toto nám už dáva algoritmus, ktorý pobeží v čase $O(N + M)$, čo je aj najlepší možný čas. Pamäť je tiež $O(N + M)$. Tak, to bolo na dnes všetko, pre záujemcov som ešte nakódil aj program...

Listing programu:

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

vector<vector<int> > G;
vector<long long> V;

long long pocet(int v){
  if (V[v] >= 0) return V[v]; // uz sme to pocitali
  long long sum=0;
  for(unsigned int i=0;i<G[v].size();i++)
    sum+=pocet(G[v][i]);
  V[v] = sum; // ulozime si vysledok
  return sum;
}

int main(){
  int N,M;
  cin >>N>>M;
  G.resize(N); V.resize(N,-1);
  V[N-1]=1;
  for(int i=0;i<M;i++){
    int a,b;
    cin >> a >> b;
    G[a-1].push_back(b-1);
  }
  cout << pocet(0) << endl;
  return 0;
}
```

2. Odíď, vírus

opravoval Mirko
(max. 15 bodov)

Máme n slov, súčet dĺžok slov je s a text je dlhý m znakov. Najjednoduchšie riešenie funguje tak, že pre každú pozíciu v texte a každé slovo skúsime, či sa na tej pozícii nachádza. Na jedno „skúsenie“ slova treba v najhoršom prípade rádovo toľko porovnaní, koľko má to slovo písmen, čo je pre každú pozíciu rádovo s operácií, spolu $s \cdot m$.

V čom je pomalosť tohto riešenia? No predsa v tom, že pre každú pozíciu zisťujeme pre každé slovo, či tam je. Dá sa to urobiť aj ináč: pre každú pozíciu overíme všetky slová naraz. Nebudeme to teda robiť tak, že pre každé slovo najprv prečítame nejaké písmenká, a potom povieme, či sa tam nachádza alebo nie a následne sa vrátime a skúsime ďalšie slovo, ale po prečítaní nejakých písmeniek budeme vždy vedieť, ktoré slová mohli začínať na tej pozícii, kde sme začali (prípadne že žiadne, alebo že už sme nejaké našli).

Myšlienka je jednoduchá: V každom kroku si budeme udržiavať množinu slov, ktoré začínajú na práve prečítané písmenká. A ku každej množine iba potrebujeme vedieť, ktorú množinu slov si máme zobrať keď prečítame 0 alebo 1. Ale ako to spraviť efektívne? Stačí si uvedomiť, že pre množinu, ktorú máme je charakteristické to, že všetky slová v nej začínajú

rovnako, teda sa zhodujú na prvých x písmenkách, teda po zotriedení podľa abecedy by tvorili nejaký suvislý interval. Teda stačí si pre každý takýto interval zrátať, kam ďalej na nejaké prečítané písmenko. (Toto skrýva jeden drobný problém, premyslite si aký).

Vzorové riešenie je však jednoduchšie, vo svojej podstate je to to isté, ale oveľa lahšie sa dá implementovať. Použijeme písmenkový strom, odporúčam pozrieť riešenie domáceho kola MOP-54, 3. príklad, kde nájdete dosť formálny popis algoritmu, ktorý použijeme. Písmenkový strom, alebo tiež „trie“, je štruktúra, ktorá slúži na reprezentovanie nejakej množiny slov. Každý uzol stromu má toľko synov, koľko je písmen v abecede. Niektoré hrany nemusia mať definovaný vrchol, kam idú. Na začiatku má strom len jeden vrchol a žiadna hrana nemá nič definované, iba to, že je nedefinovaná.

Slovo pridávame do stromu tak, že začneme v koreni a pozrieme sa na hranu pre začínajúce písmenko pridávaného slova. Ak taká hrana existuje, tak odtrhneme prvé písmenko zo slova, skočíme tam, kam hrana ukazovala a pokračujeme ďalším písmenkom. Ak už je slovo prázdne, tak si iba do vrcholu napíšeme, že tam nejaké slovo končí (najlepšie aj to, že ktoré). Ak hrana neexistuje, tak vytvoríme úplne nový vrchol, a pridáme hranu, ktorá naň ukazuje a pokračujeme ďalej rovnako ako v predchádzajúcom prípade.

Takto môžeme povedať, že každý uzol predstavuje nejakú množinu slov. Takže náš algoritmus môže vyzerať tak, že začneme na nejakom písmenku a zakaždým, keď prečítame ďalšie, tak sa podľa toho posunieme aj v strome. Keď prídeme do uzla, v ktorom končí nejaké slovo, tak ho vypíšeme. Keď prídeme do uzla, z ktorého nevieme na ďalšie písmenko pokračovať, tak sa vrátime a začneme hľadať od ďalšieho písmenka.

No ale prečo by sme mali znovu čítať už prečítané písmenká? No nemusíme, stačí si pre každú množinu pamätať, do ktorej množiny máme prejsť, keď odtrhneme prvé písmenko z každého slova. Toto však skrýva dva problémy. Prvý je, že tá množina bude prázdna, a druhý je, že tá množina bude prázdna preto, že prečítaný úsek je moc dlhý, teda že keby sme prečítali menej, tak by sme nejaké slovo našli. Druhý prípad sa ošetrí ľahko, lebo nemôže ani nastať (premyslite si prečo – hintom je podmienka zo zadania, že žiadne slovo nie je podstringom iného). Prvý prípad ošetríme tak, že si zapamätáme prvú takú množinu, ktorá je neprázdna. Teda pokúsime sa postupne odtrhnúť jedno, dve, tri, ... písmenká a zapamätáme si prvú neprázdnu množinu. Prvú preto, lebo k ďalším sa môžeme dostať tak, že budeme trhať neskôr. Tým, že si pamätáme množinu, tak vieme aj reťazec, akým slová v nej začínajú.

Ale ako to spravíme efektívne? Takéto návraty si budeme v písmenkovom strome pamätať ako takzvané spätné hrany. Spätnú hranu pre koreň vieme. Koreň predstavuje, že máme prečítané prázdne slovo (0 znakov). No a keď z prázdneho slova odstránime prvé písmenko (ktoré tam ale nie je), tak dostaneme brutálne prázdne slovo. Brutálne prázdne slovo sa vyznačuje tým, že keď za neho pridáme nejaké písmenko, dostaneme normálne prázdne slovo. Teda pridáme si jeden uzol, ktorý bude predstavovať brutálne prázdne slovo. Predpokladajme, že už máme zrátané všetky spätné hrany pre každý uzol v hĺbke n. Ak chceme vyrátať hranu pre slovo $w = a_0a_1a_2a_3 \dots a_na_{n+1}$, teda chceme nájsť množinu, kde všetky slová začínajú na $a_ia_{i+1}a_{i+2} \dots a_na_{n+1}$, kde i je najmenšie možné a väčšie ako 0. Vieme už ale, kde je podobná množina pre slovo $u = a_0a_1a_2a_3 \dots a_{n-1}a_n$. Nech je to $a_ja_{j+1} \dots a_{n-1}a_n$, teraz ak $a_jaj + 1 \dots a_{n-1}a_na_{n+1}$ je platná množina (teda máme v písmenkovom strome definovanú hranu na písmenko a_{n+1} z uzla, kam sa dostaneme spätnou hranou z uzla pre slovo u) tak vlastne sme našli, kam má ísť spätná hrana z uzla pre slovo w (teda pre množinu slov, ktoré začínajú na w). Ak z w hrana na písmenko a_{n+1} nevedie, skúsime použiť spätnú hranu slova $a_ja_{j+1} \dots a_{n-1}a_n$, čo bude zase iba nejaké slovo $a_ka_{k+1} \dots a_{n-1}a_n$, toto opakujeme do konečna, vďaka tomu, že máme brutálne prázdne slovo (ako spätná hrana pre koreň t.j. pre prázdne slovo).

Teraz, ked už máme túto peknú štruktúru, stačí po jednom písmenku pozerať vstup a vždy, keď môžeme v strome ísť po hrane, tak pôjdeme. Keď nemôžeme, použijeme spätnú hranu (a znovu skúsime ísť po hrane pre posledné prečítané písmenko). Keď prídeme do uzla,

v ktorom nejaké slovo končí, tak daný výskyt tohoto slova vypíšeme. To bude mať zložitosť $O(n + s)$. Za ňu bolo 15 bodov, za $O(m \cdot s)$ bolo 8, a za niečo medzi tým niečo medzi tým.

Ešte je užitočné vedieť, čo treba robiť, ak by sme nemali podmienku, že žiadne slovo nie je podstringom iného. Vtedy si treba do stromu zaznačiť, keď tam pridávame nejaké slovo, kde končí, a potom, keď sme v nejakom stave, pozrieme sa, na ktoré uzly by sme sa dostali čisto po spätných hranách. Ak sa môžeme na nejaké, kde končí slovo, tak sme tam toto slovo našli. Toto si však vieme spočítať pre každé slovo na začiatku. Alebo aj na konci dynamikou od konca. Iba si zapamätáme na akej pozícii sme boli v tom stave keď sme prečítali znak.

Listing programu:

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

// definujeme si triedu "pismenkovy strom"
class trie {

  // definujeme si triedu "vrchol v pismenkovom strome"
  class __trieNode {
    public:
      // pamatame si poradove cisla slov co tu koncia
      vector<int> endsHere;
      // pointer na nasledujuci vrchol, kde koncia nejake slova
      __trieNode *nextOutput,
      // a pointre na synov (pre kazde pismeno jeden)
      *son[32], *higher;
      // konstruktor len vsetky pointre nastavi na NULL
      __trieNode() { nextOutput=higher=NULL; memset(son,0,sizeof(son)); }
  };

  // staci si pamatat pointer na koren stromu
  __trieNode *root;
  // pocet slov ulozenych v strome
  int __size;

  public:

  // konstruktor: nemame koren, mame 0 slov
  trie() { root = NULL; __size = 0; }

  // insert(S): vloz slovo S
  inline void insert(const string &S) {
    // ak nemame koren, vytvor koren
    if (!root) root = new __trieNode();
    __trieNode *kde = root;
    // zlezieme dole po strome, pripadne vyrabame nove vrcholy
    for (unsigned i=0; i<S.size(); i++) {
      int idx = S[i]-'0';
      if (! kde->son[idx]) kde->son[idx] = new __trieNode();
      kde = kde->son[idx];
    }
    // zaznacime si, ze tu konci slovo
    kde->endsHere.push_back(__size++);
  }

  // createAutomaton(): vytvor spaetne linky,
```

```
  //   aby sme vedeli efektivne vyhladavat ulozene slova v texte
  inline void createAutomaton() {
    if (!root) return;
    // vytvorime pomocny vrchol pre "brutalne prazdne slovo"
    __trieNode *superRoot = new __trieNode();
    for (int i=0; i<26; i++) superRoot->son[i] = root;
    root->higher = superRoot;
    // prehladavanim do sirky budujeme spaetne linky
    queue<__trieNode*> Q;
    Q.push(root);
    while (!Q.empty()) {
      __trieNode *kde = Q.front(); Q.pop();
      for (int i=0; i<26; i++) if (kde->son[i]) {
        __trieNode *cand = kde->higher;
        while (! cand->son[i]) cand = cand->higher;
        kde->son[i]->higher = cand->son[i];
        if (kde->son[i]->higher->endsHere.empty()) {
          kde->son[i]->nextOutput = kde->son[i]->higher->nextOutput;
        } else {
          kde->son[i]->nextOutput = kde->son[i]->higher;
        }
        Q.push(kde->son[i]);
      }
    }
  }

  // reportMatches(): najdi v S vsetky nase slova
  inline void reportMatches(const string &S) {
    __trieNode *kde = root;
    // ideme po pismenach stringu
    for (unsigned i=0; i<S.size(); i++) {
      int idx = S[i]-'0';
      // updatneme aktualny vrchol
      while (! kde->son[idx]) kde = kde->higher;
      kde = kde->son[idx];
      // a vypiseme slova, ktore tu koncia (if any)
      __trieNode *out = kde;
      while (out) {
        for (unsigned j=0; j < out->endsHere.size(); j++)
          cout << "na pozicii " << i << " konci slovo cislo " << out->endsHere[j] << endl;
        out = out->nextOutput;
      }
    }
  }
};

int main() {
  trie T;
  int N;
  cin >> N;
  string S;
  while (N--) { cin >> S; T.insert(S); }
  cin >> S;
  T.createAutomaton();
  T.reportMatches(S);
}
```

3. O podenkách

opravoval Lukáš
(max. 15 bodov)

Tento priklad bol ťažký na vymyslenie, ale programoval sa jednoducho. Prišli len dva typy riešení. Riešenie v čase $O((N-1)^M)$ pracuje nasledovne: budeme simulovať, ako sa liahnu podenky. Pre každú generáciu si vyrátame, s akou pravdepodobnosťou môže byť v tejto generácii k podeniek. Na konci môže byť až $(N-1)^M$ podeniek, čo je veľmi veľa.

Vzorové riešenie si v každej generácii ráta len jednu pravdepodobnosť. Nech r_i je pravdepodobnosť, že populácia tvorená jednou podenkou zahynie do i generácii (v triviálnom prípade je $r_1 = p_0$). Potom pravdepodonosť, že k podeniek zahynie do i generácií, je r_i^k. Totiž každá z k udalostí nastane s pravdepodobnosťou r_i a my chceme vedieť, s akou pravdepodobnosťou nastanú všetky udalosti naraz. Teraz sa poďme zamyslieť, ako pomocou r_i vyrátame r_{i+1}. V tejto generácii je len jedna podenka. Tá môže mať 0 až $N-1$ potomkov, teda máme N možností, koľko podeniek môže byť v ďalšej generácii. S pravdepodobnosťou p_j bude v ďalšej generácii j podeniek a tie musia všetky zahynúť do i generácií. Keďže tieto možnosti sú na sebe nezávislé, pravdepodobnosti sčítame. Dostávame vzorec:

$$r_{i+1} = \sum_{j=0}^{N-1} p_j \cdot r_i^j$$

Ak vám nie je úplne jasné, prečo sme niektoré pravdepodobnosti sčítali a iné vynásobili, prečítajte si tento odstavec. Predstavme si, že chceme zrátať, aká je pravdepodobnosť, že hodíme na kocke 3-krát za sebou päťku alebo 3-krát za sebou šestku. Pravdepodobnosť, že padne trikrát za sebou tá istá (nami vopred zvolená) hodnota, je $(\frac{1}{6})^3$, lebo máme tri udalosti s pravdepodobnosťou $\frac{1}{6}$ a musia nastať všetky tri po sebe. Keďže môžeme hodiť päťku alebo šestku, dokopy je pravdepodobnosť $(\frac{1}{6})^3 + (\frac{1}{6})^3$. Totiž tieto dve udalosti sú na sebe nezávislé.

Časová zložitosť nášho riešenia je $O(NM)$, pamäťová $O(N)$. Ešte jedna malá poznámka. Keďže na konci spočítame hodnotu r_M a v akváriu je K podeniek, vypíšeme hodnotu r_M^K.

Listing programu:

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

int main() {
    int n, k, m; cin>>n>>k;
    vector<double> p(n);
    for (int i=0; i<n; i++) cin>>p[i];
    cin>>m;

    double pravd=0, nova, pom;
    for (int j=0; j<m; j++) {
        nova=0; pom=1;
        for (int i=0; i<n; i++) {
            nova+=p[i]*pom; pom*=pravd;
        }
        pravd=nova;
    }
    cout.setf(ios::fixed, ios::floatfield); cout.precision(5);
    cout<<pow(pravd, k)<<endl;
}
```

4. O duplikátoch a pánovi Dirichletovi...

opravoval Kubo
(max. 15 bodov)

Riešenia ku mne sa dostaviace by sa dali rozdeliť do troch kategórií.

Zlý čas/zlá pamäť: S lineárnou pamäťou (navyše) je každý frajer. Čísla sú v rozsahu $1 \dots n$, takže si vieme pre každé číslo z rozsahu zapätať, koľkokrát sa v poli nachádza. Dobrý čas, ale zlá pamäť.

Druhá myšlienka, ktorá musela každému zísť na um je skontrolovať všetky dvojice. Tých je rádovo $O(n^2)$ – dobrá pamäť, zlý čas; predsa len, toto je riešenie, ktoré mal každý napísať, keď ho nenapadlo nič lepšie.

Len o log horšie ako vzorové riešenie: Číslo, ktoré sa opakuje, budeme hľadať metódou binárneho vyhľadávania. Nech je hľadané číslo v rozsahu $l \dots r$ (na začiatku vieme, že číslo z rozsahu $1 \dots n$ sa opakuje). Rozdeľme tento interval (približne) na dve polovice: nech $m = \lfloor (l+r)/2 \rfloor$. Jedným prechodom poľa vieme zistiť, koľko čísel je v rozsahu $l \dots m$ a koľko je v rozsahu $m+1 \dots r$. Ak je v prvom rozsahu viac ako $m-l+1$ čísel, resp. ak je v tom druhom viac ako $r-m$ čísel, podľa Dirichletovho princípu sa tam musí nejaké číslo opakovať. Ďalej teda budeme hľadať iba v danom intervale. Takýmto spôsobom vieme jediným prechodom poľa (v lineárnom čase) zmenšiť na polovicu rozsah čísel, v ktorom je určite hľadané číslo. Preto bude týchto prechodov iba $\lceil \log_2 n \rceil$. Výsledná časová zložitosť teda bude $O(n \log n)$.

Vzorák: Ako napovedalo už zadanie, vzorák bude lineárny. Základom celého riešenia bolo pozrieť sa na pole ako na zobrazenie – ako na funkciu, ktorá číslu 0 prideľuje hodnotu $A[0]$, číslu 1 hodnotu $A[1]$; resp. všeobecne číslu i hodnotu $A[i]$. Vyzývam všetkých riešeniachtivých (myslím takých, ktorí chcú ešte riešiť, nie si len prečítať riešenie), nech sa kedykoľvek od vysvetľovania vzoráku odpoja a zvyšok riešenia si domyslia sami.

Len pre porovnanie, vezmime si najskôr pole od 1 po n s číslami od 1 po n, každé práve raz. Takéto pole by sme si mohli predstaviť aj takto: napíšme si čísla od 1 po n a ak $A[i] = j$, nakreslime šípku z čísla i do čísla j. Čo dostaneme? Zjavne permutáciu. Keďže každé číslo sa v poli nachádza práve raz, z každého čísla pôjde práve jedna šípka a do každého čísla bude viesť práve jedna šípka. Ak sa vydáme po šípkach nejakým smerom, o chvíľu sa vrátime do toho istého miesta (matici by povedali, že permutácia sa dá rozložiť na cykly).

To, čo dostaneme v našej úlohe, však určite nebude permutácia. Ako to teda bude vyzerať? Napríklad pole $A = (1, 3, 2, 4, 1, 2)$ zo zadania si vieme znázorniť tak, že si napíšeme čísla od 0 po 5 (indexujeme od nuly) a pridáme šípky $0 \to 1$, $1 \to 3$, $2 \to 2$, $3 \to 4$, $4 \to 1$ a $5 \to 2$. Už to nebude také pekné ako pri permutáciach. Z každého čísla bude stále vychádzať práve jedna šípka, avšak do každého čísla nemusí vchádzať 1 šípka – môže ich byť viac a nemusí doň vchádzať ani jedna šípka! Napríklad do takej nuly nikdy nevchádza žiadna šípka (nie je z rozsahu $1 \dots n$). V našom príklade do 1 a do 2 vchádzajú dve šípky – to sú čísla, ktoré sa v poli opakujú.

Ak sa v našom príklade vydáme po nejakej ceste po šípkach podobne ako v prípade permutácií, už sa nemusíme dostať do toho istého miesta, kde sme začali. V našom príklade, ak sa vyberieme z 0 alebo 5, dostaneme sa do cyklu $1 \to 3 \to 4 \to 1$, resp. $2 \to 2$, ale nevrátime sa späť do 0, resp. 5. A v prípade nuly to nebola náhoda!

Ak začneme z nuly a budeme postupovať po šípkach, po nejakom čase sa dostaneme do miesta, v ktorom sme už boli (ale nie do nuly – ako sme už povedali, do nuly nevchádza žiadna šípka). Toto číslo budeme volať „začiatok cyklu“; dostali sme sa doň dvoma cestami: prvýkrát z 0, druhýkrát po obehnutí jedného kolečka. Začiatok cyklu je teda to číslo, ktoré hľadáme.

Ako teda nájdeme začiatok cyklu? Posledná možnosť odpojiť sa a domyslieť si to sám... Číslo nula volajme štart a index do poľa volajme „bežec“[1]. Keď sa „pohneme bežcom“, myslíme tým jeden prechod po šípke. Vo všeobecnosti na cyklické štruktúry funguje metóda

[1] bežec vyzerá asi ako panáčik z človečka, ale má na pohľad svalnatejšie nohy

dvoch bežcov – pustíme zo začiatku jedného pomalého a druhého dvakrát rýchlejšieho[2]. Bežci sa po nejakom čase dostanú do cyklu a tam po čase rýchlejší dobehne toho pomalšieho[3]. No a z miesta, kde sa stretnú, sa dá ešte rôznymi myšlienkovými pochodmi a pochodmi bežcov prísť až na štart cyklu.

Myšlienkovo asi najjednoduchšie a najzrozumiteľnejšie riešenie je od Pershinga: Postavíme jedného bežca na štart. Pohneme ním n-krát. Keďže máme $n+1$ čísel, v danom momente sme sa už určite dostali do cyklu. Bežca teraz naklonujeme[4] a klon pustíme obehnúť 1 kolečko (hýbeme ním, kým neprídeme na pôvodné miesto – aj keď je tento program sklerotický jak fras, toto miesto spoznáme podľa bežca, ktorý tam na svoj klon čaká). Takto vieme zistiť dĺžku cyklu – označme ju c. Postavme teraz na štart druhých dvoch bežcoch (rovnako rýchlych). Prvému dajme náskok c krokov[5]. Kde sa tým pádom musia stretnúť? To si povieme nabudúce. Pozrite si zdroják.

Bodovanie: Prišlo nám 16 riešení, body boli od 0 po 15, takže ani pán Dirichlet by vám o bodoch veľa nepovedal. Preto vám o nich poviem dačo ja. Bodovalo sa takto:

- lineárna pamäť alebo kvadratický čas – max. 5 bodov
- konštantná pamäť a čas $O(n \log n)$ – max. 10 bodov
- konštantná pamäť a lineárny čas – 15 bodov
- stŕhalo sa za rôzne prehrešky rôzne

Listing programu:

```
#include <cstdio>
#define START 0
#define POHYB(b) b=a[b]
#define REP(i,n) for(int i=0; i<n; ++i)

int n, a[10000], b, B, c;

int main() {
  // nacitame
  scanf ("%d", &n); REP(i,n+1) scanf("%d", &a[i]);
  // dostan bezca B do cyklu
  B = START; REP(i,n) POHYB(B);
  // bezcom b obehni kolecko a zisti dlzku cyklu
  b = B; POHYB(b); c = 1;
  while (b!=B) { POHYB(b); c++; }
  // postav teraz b a B spaet na start
  // daj B naskok jedneho cyklu -- tak sa stratnu na zaciatku cyklu
  b = START; B = START;
  REP(i,c) POHYB(B);
  while (b!=B) { POHYB(b); POHYB(B); }
  // vypiseme
  printf("%d\n", b);
  return 0;
}
```

[2] kvízová otázka: je tento dvakrát rýchlejší bežec rýchly?

[3] vydýchajú sa a pôjdu spolu na pivo

[4] informatici nemajú zmysel pre etiku, či čo?

[5] skokov? Otázka pre športovcov: pri kráčaní robíme kroky; čo robíme pri bežaní?

5. O tabuľke profesora Kolmogorova

opravoval MišoF.
(max. 15 bodov)

Ktorá z postupností 0110110101011010... a 0101010101010101... vyzerá náhodnejšie? Oko kukne, oko vidí, oko vie že tá prvá. Ale ako to matematicky definovať?

Tu prichádza na scénu práve pán Kolmogorov, spomínaný v zadaní, so svojim pozorovaním: *Objekt obsahuje toľko bitov informácie, aký dlhý je najkratší popis, podľa ktorého ho vieme vygenerovať.* (Tomuto sa na jeho počesť hovorí Kolmogorovská zložitosť objektu.)

A čo to znamená, ak sú naše objekty súbory? Kolmogorovská zložitosť súboru je vlastne veľkosť najmenšieho programu, ktorý nám daný súbor vygeneruje. Nepripomína vám to niečo? Áno, správne, zaváňa to kompresiou, ten program je vlastne akoby samorozbaľovací archív.

Vráťme sa teraz na chvíľu k našim dvom postupnostiam zo začiatku. Určite by sme ľahko napísali kratučký program, ktorý by vygeneroval druhú postupnosť (pre zadanú dĺžku). Na druhej strane, ak je postupnosť skutočne náhodná, znamená to, že neobsahuje žiadne závislosti – a teda nebude existovať principiálne jednoduchší spôsob jej generovania, ako `writeln(postupnost);`.

Toto pozorovanie môžeme použiť na definíciu náhodnosti: *Postupnosť je náhodná, ak sa nedá skomprimovať.*

V našom prípade tabuľka zo zadania zďaleka neobsahovala náhodné čísla. A práve toto sa dalo využiť pri hľadaní čo najkratšieho predpisu, ktorý našej tabuľke zodpovedá. (Použitím vyššie definovanej terminológie, vašou úlohou bolo napísať program, ktorý sa veľkosťou čo najviac priblíži Kolmogorovskej zložitosti zadanej tabuľky.)

Takto vyzerá naše riešenie:

Listing programu:

```
unsigned n;
main() {
  while (scanf("%u",&n)>0)
    printf("%d\n",
      ( n>17123 && n<47325
        ?
        n = n*n/13 - 3*n + 2
        :
        (n *= n/8, n>999 && (n/=47))
        ,
        n ^= 2*n & 28,
        n+47 )
    );
}
```

Výsledková listina po 2. kole kategórie KSP

	Meno a priezvisko	Škola	Trieda		21	22	23	24	25	Σ
1	Perešíni Peter	Gym. Tajovského B. Bystrica	4	66	15	14	15	15	15	140
2	Herman Peter	Gym. Jura Hronca BA	3	47	15	14	15	6	10	107
3	Okruhlica Adam	Gym. Jura Hronca BA	3	41	15	14	3	15	9	97
4	Brezáni Samuel	Gym. Rajec	3	46	4	10	7	10	9	86
5	Mikuláš Ondrej	Gym. Haličská Lučenec	3	31	15	12		8	5	71
6	Tříska Martin	Gym. P. de Coubertina Piešťany	3	40	13	8		5		66
7	Danilák Michal	Gym. Hubeného BA	3	0	15	15	15	10	10	65
8	Rampášek Ladislav	Gym. Jura Hronca BA	3	38	15				9	62
9	Nagy Anton	Gym. maďarské BA	3	27	9	12	2	5		55
10	Petrucha Michal	Gym. Metodova BA	1	24	7	12				43
11	Tomcsányi György	Gym. H. Selyeho Komárno	3	14	6	7		10		37
12	Bílka Ondřej	Gym. Zlín	4	7	0	7	10	10	2	36
12	Boža Vladimír	Gym. Tatarku Poprad	2	0	15	11		10		36
14	Pančík Andrej	Gym. Tajovského B. Bystrica	3	13	6	9		5		33
15	Imriška Jakub	Gym. Jura Hronca BA	4	30						30
16	Sucha Martin	Gym. Jura Hronca BA	2	16		7		5		28
17	Bundala Daniel	Gym. Jura Hronca BA	4	13					6	19
18	Jerguš Ján	Gym. Alejová Košice	3	17						17
19	Buštor Ivan	Gym. Jura Hronca BA	3	0		11		5		16
20	Čevorová Kristína	Škola pre mim. nadané deti BA	3	0	1	8		2		11
20	Labaj Martin	Gym. Varšavská Žilina - Vlčince	1	0					11	11
22	Danko Juraj	Gym. Jura Hronca BA	3	9						9
23	Schlosáriková Lucia	Gym. P. de Coubertina Piešťany	3	0	4			4		8
24	Svonava Daniel	Gym. Jura Hronca BA	4	5						5
25	Bartoš Peter	Gym. Fándlyho Šaľa	4	0					4	4
26	Hapák Samuel	Gym. Grösslingová BA	2	0					3	3
27	Korcsok Peter	Gym. Mládežnícka Šahy	2	2						2