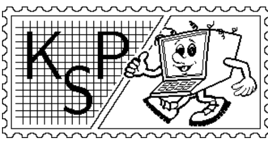


Korešpondenčný seminár z programovania
XXVIII. ročník, 2010/11
Katedra základov a vyučovania informatiky FMFI UK,
Mlynská Dolina, 842 48 Bratislava

KSP finančne podporujú: MICROSTEP-MIS spol. s r.o.
Agentúra na podporu výskumu a vývoja

Vzorové riešenia 2. kola zimnej časti

opravovala Táňa
(max. 10 bodov)

1. Zápis predmetov

Na začiatku bolo treba prísť na to, že údaj o dĺžke vlasov je zavádzajúci, lebo dĺžka radu nezávisí od toho, koho konkrétne teta zavolá dnu. Zaujíma nás len, že jeden študent z radu odišiel. To sa väčšine podarilo, takže vaše riešenia sa líšili len v detailoch. Pozrime sa na ideálne riešenie.

Najprv načítame počet udalostí, potom už len reagujeme na každú osobitne a priebežne upravujeme aktuálnu dĺžku radu – keď príde 0, rad sa skráti o jedného študenta, a keď príde kladné číslo, rad sa zase o jedného predĺži. Potrebujeme zistiť, akú najväčšiu dĺžku dosiahol rad počas celého dňa. Na to si budeme v osobitnej premennej pamätať najväčšiu dĺžku radu, akú sme zatiaľ videli. Keď bude rad dlhší ako naše maximum, tak toto maximum prepíšeme.

Hoci nám to nezlepší asymptotickú zložitosť riešenia, zamyslime sa nad malou optimalizáciou. Kontrolu, či aktuálna dĺžka radu nepresiahla zapamätané maximum, nie je nutné vykonávať po každej udalosti. Stačí nám kontrolovať to len po predĺžení radu, alebo len pred skrátením radu (a ešte raz úplne nakoniec). Druhá možnosť je výhodnejšia: rad sa nemôže skrátiť viackrát, ako sa predĺžil. Takto vykonáme najviac $N/2$ kontrol.

Časová zložitosť riešenia je lineárna vzhľadom na veľkosť vstupu, $O(N)$, lebo pri spracovaní každej z N udalostí vykonám len konštantný počet príkazov. Pamäťová zložitosť je konštantná, $O(1)$, lebo si pamätám vždy len pár premenných – aktuálnu a maximálnu dĺžku radu, počet udalostí, aktuálnu udalosť a iterátor.

Hodnotenie: Plný počet bodov dostalo riešenie s najmenšou možnou zložitosťou, jej dobrým odhadom a dostatočným popisom. Jeden bod som strhávala za zlý alebo chýbajúci odhad pamäťovej alebo časovej zložitosti. Za pamäťovú zložitosť $O(N)$ boli 4 body dole a riešenie so zlou úvahou nedostalo viac ako 3 body.

Dôležitá vec – zlé riešenie s popisom má šancu získať viac bodov ako zlé riešenie bez popisu, lebo je vidieť vaša úvaha, ktorá nemusí byť tak scestná ako sa javí program.

Viac nemám čo dodať, jedine kód :)

Listing programu:

```
program zapis;
var N, dlzka, max, i, vlasy: integer;

begin
  { na začiatku máme rad dĺžky 0, takže aj maximum bude nula }
  dlzka := 0;
  max := 0;
  ReadLn(N);

  for i := 1 to N do begin
    ReadLn(vlasy);
```


Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

```
    if vlasy > 0 then Inc(dlzka)
    else begin
      if dlzka > max then max := dlzka; { ak dostanem 0, pozriem, či je rad }
      Dec(dlzka);                       { dlhší ako max, potom ho skrátim }
    end;
  end;

  if dlzka > max then WriteLn(dlzka) { ak sa rad nakoniec predlžoval, }
  else WriteLn(max);                 { mohol presiahnuť maximum }
end.
```

2. Zastavme nezbedníkov!

vzorák písal Tomi
(max. 10 bodov)

Nezbedník sa nutne musel predbehnúť práve vtedy, ak je niekde neskôr po ňom študent s menším číslom. Napríklad ak je niekde v rade 4, a niekde za ním je 2, tak sa 4 predbehol. Mohli by sme sa pre každého študenta pozrieť na celý zvyšok radu, či tam nie je nejaké menšie číslo, ale rýchlejšie je ísť odzadu a pamätať si, aké najmenšie číslo sme zatiaľ videli. Tým pádom keď uvidíme číslo x, ale už sme niekde doteraz (čiže ďalej v rade) videli aj číslo menšie ako x, tak sa x predbehol. Nakoniec vypíšeme tie x, čo sa predbehli, vo vzostupnom poradí.

Prechádzanie vstupným poľom trvá $O(N)$, lebo pri spracovaní každého študenta vykonáme len konštantný počet operácií. Načítanie aj výstup trvá tiež $O(N)$, preto má celé riešenie časovú zložitosť $O(N)$. Keďže si pamätáme rad študentov a pre každého aj to, či sa predbehol, potrebujeme $O(N)$ pamäte.

Listing programu:

```
program nezbednici;
var n, i, x, minimum: longint;
    studenti: array [1..100000] of longint;
    predbeholsa: array [1..100000] of boolean;

begin
  ReadLn(n);
  for i := 1 to n do ReadLn(studenti[i]);
  for i := 1 to n do predbeholsa[i] := false;
  minimum := n;
  for i := n downto 1 do begin
    x := studenti[i];
    if x > minimum then predbeholsa[x] := true;
    if x < minimum then minimum := x;
  end;
  for i := 1 to n do begin
    if predbeholsa[i] then WriteLn(i);
  end;
end.
```

3. Zamorenie špiónmi

opravoval Ivan
(max. 10 bodov)

Algoritmus na riešenie bol jednoduchý, úvaha za ním bola trochu zložitejšia. Skoro všetci riešitelia úspešne vyriešili túto úlohu, alebo aspoň vymysleli dobrý algoritmus a mali tro-

Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

chu problémy s triedením – napríklad sa rozhodli predpokladať, že vstup bude utriedený. Zdôvodňovanie správnosti algoritmu niektorým nešlo práve najlepšie, ale prax to zlepší.

Jeden základný vzťah hodný povšimnutia je, že jednotka, ktorá sa raz do základne dostala, môže sa tam vždy znovu dostať. Ak jednotka bola poslaná späť pre príjazd inej jednotky, tak ju tam môžeme dostať znovu tak, ako sme ju tam dostali minule. Navyše na to, aby sme na základňu dostali nejakú jednotku, nepotrebujeme presúvať väčšie jednotky.

Znamená to, že nám pri počítaní stačí vedieť množinu jednotiek, ktorú tam vieme dostať a postupne do nej pridávať jednotky (pokiaľ sa dá). Dokonca nám stačí pamätať si počet špiónov v tejto množine a z toho už vieme, akú najväčšiu jednotku tam môžeme pridať (súčet jednotiek v množine a maximálneho denného prírastku d).

Teraz by už malo byť vidieť, že vhodným algoritmom môže byť utriediť jednotky podľa veľkosti, skúšať ich od najmenšej pridať, pokiaľ sa to dá a vypísať, koľko sa ich podarilo pridať. Tento algoritmus mala väčšina riešiteľov. Jeho časová zložitosť je kvôli triedeniu $O(n \log n)$ a pamäťová $O(n)$.

Listing programu:

```
program spioni;
var A: array of integer;
    B: array of integer;
    n, i, d: integer;

procedure mergesort(u, v: integer);  { merge sort: utriedi A[u], ..., A[v - 1] }
var i, j, k, m: integer;

    procedure copy(var p: integer);  { prekopíruje jeden prvok do pomocného poľa }
    begin
        B[k] := A[p];
        Inc(k);
        Inc(p);
    end;

begin
    if v - u < 2 then  { menej ako 2 prvky => vždy utriedené }
        Exit;
    m := (u + v) div 2;  { stred triedenej časti }
    mergesort(u, m);  { rekurzívne utriedime obe polovice }
    mergesort(m, v);

    i := u;  { ukazovateľ do prvej polovice, }
    j := m;  { do druhej polovice, }
    k := u;  { do pomocného poľa }
    while (i < m) and (j < v) do
        if A[i] < A[j] then  { najprv menší prvok }
            copy(i)
        else
            copy(j);
    while (i < m) do  { keď sa už minuli prvky z niektorej polovice, }
        copy(i);        { kopírujeme tie zvyšné }
    while (j < v) do
        copy(j);

    for i := u to v - 1 do  { nakoniec prekopírujeme prvky z pomocného poľa späť }
        A[i] := B[i];
end;

begin
    Read(n);
```


Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

```
    SetLength(A, n);
    for i := 0 to n - 1 do
        Read(A[i]);
    Read(d);

    SetLength(B, n);
    mergesort(0, n);

    for i := 0 to n - 1 do
        if A[i] <= d then
            d := d + A[i]  { môžeme presunúť aj túto jednotku }
        else begin
            WriteLn(i);  { podarilo sa presunúť len i jednotiek }
            Exit;
        end;
    WriteLn(n);  { podarilo sa presunúť všetky }
end.
```

opravoval Mio
(max. 15 bodov)

4. Zbytočné opevnenie

Úvodom treba spomenúť to, na čo ste všetci prišli – že údaj o pozícii veže bol zbytočný (nebolo ho však treba zabudnúť načítať).

Väčšina z vás odovzdala riešenie podobné tomuto: Utriedime si pole intervalov podľa ich začiatku. Potom toto pole prechádzame zľava a kontrolujeme, či aktuálny interval nie je obsiahnutý v nejakom predchádzajúcom. Kontrola prebieha tak, že si pamätáme koniec intervalu, ktorý siaha najviac doprava a kontrolujeme, či koniec aktuálneho intervalu neleží vľavo od neho.

Ešte tu ostáva jeden problém (na ktorom stroskotalo mnoho z vás) a to prípad, ak dva intervaly začínajú na tom istom mieste. Vtedy ich potrebujeme usporiadať tak, aby sme najskôr spracovali ten interval, ktorý končí neskôr.

Časová zložitosť riešenia – raz tu triedim a inak nič, čo by trvalo dlhšie – čiže $O(n \log n)$. Pamäťová zložitosť – $O(n)$ pamätať si treba informácie o vežiach.

Plne funkčné riešenie so zložitosťou $O(n \log n)$ dostalo max. 15 bodov. Ak ste odovzdali plne funkčné riešenie so zložitosťou $O(n^2)$, mohli ste získať 10 bodov. Ak ste neošetrili hraničné prípady, dostali ste sankciu -3 body. -1 bod bol za každú chýbajúcu zložitosť. $+1$ bonusový bod pri kvadratickom riešení bol, ak túto zložitosť spôsobovalo pomalé triedenie. Ďalšie body boli strhávané za iné chyby, prípadne ak bolo riešenie nefunkčné.

To je všetko, užite si vzorák.

Listing programu:

```
const MAXN = 10000;
var n, i, s, naj: integer;
    A, B, P: array [1..MAXN] of integer;

begin
  ReadLn(n);
  for i := 1 to n do
    ReadLn(A[i], s, B[i]);

  { Zotriedime intervaly vzostupne podľa začiatku, v prípade rovnosti zostupne
    podľa konca. Kód mergesort-u je v predchádzajúcej úlohe. }

  naj := 0;
```


Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

```
  for i := 1 to n do begin
    if B[i] <= naj then
      WriteLn(P[i])  { pôvodné číslo intervalu, ktorý je i-ty po zotriedení }
    else
      naj := B[i];
  end;
end.
```

5. Zbohatlíkom odzajtra

Vzorák PPershing, opravovalo sa samo
(max. 20 bodov)

Hneď na úvod začnem tým, že úloha dopadla o dosť horšie ako som očakával. Väčšina z vás robila kadejaké podivné heuristiky, i keď riešenie je v skutku jednoduché. Omnoho väčší problém bol pozháňať a poparsovať zdrojové údaje. Samotná stránka poskytuje možnosť downloadu, pri troche snahy zistíme aj linku.

Takže postupne – najskôr sa pozrieme na fázu „zháňania“ vstupov. V našom prípade sťahovania kurzového lístka. Priamo samotná stránka má download link na aktuálny výber a pri troche snahy sa dostaneme k nasledujúcej URL:

```
http://www.oanda.com/currency/historical-rates/download?quote_currency=#FROM#&price=bid
&end_date=2011-1-1&start_date=2009-12-31&period=daily&display=absolute&rate=0&data_range=y2
&view=table&base_currency_0=#TO#&download=csv
```

kde `#FROM#` a `#TO#` sú zamenené príslušnými menami. Dané možné kombinácie všetkých mien (ktorých je $12 \cdot 11 = 132$) môžeme buď posťahovať ručne, alebo pomocou nejakého skriptu. Dostaneme údaje v nasledujúcom CSV formáte:

```
"Daily BID rates @ +/- 0%"
"www.oanda.com/currency/historical-rates/"
""
""
"End Date","EUR/CZK","","","",""
"2011-01-01","25.0608",,,,
"2010-12-31","25.2353",,,,
"2010-12-30","25.2918",,,,
"2010-12-29","25.3268",,,,
...
```

Evidentne takýto formát je už pomerne použiteľný a dá sa pomerne jednoducho čítať (a pre tých lenivejších sa dá dokonca použiť CSV knižnica). Každopádne, už máme všetky potrebné vstupné údaje, hurá konečne do riešenia úlohy.

V prvom rade sa zamyslime, ako najlepšie môže PPershing investovať svoje peniaze. Možno sa to na prvý pohľad nezdá, ale vždy sa mu oplatí mať všetky peniaze v jednej mene. Prečo? Predpokladajme, že by PPershing investoval do rôznych mien. Potom ale v priebehu času (najneskôr na konci) musí svoje investície aj tak zameniť za spoločnú menu. Pozrime sa teraz na obidve „cesty“ finančným svetom. Ak jedna mala vyšší percentuálny výnos ako tá druhá, investovaním všetkého do tej lepšej by si PPershing prilepšil. V prípade, že majú rovnaký percentuálny výnos, je to jedno a teda riešenie používajúce iba jednu cestu je rovnako dobré. Záverom našich myšlienok je, že PPershingovi sa oplatí mať furt celé svoje imanie v jednej mene[1].

[1] Nie je to úplne pravda. Dané tvrdenie platí len v ideálnom matematickom svete. V našom finančnom svete zaokrúhľovania nadol môžu ovplyvniť výsledok. Napríklad sa nám môže oplatiť veľmi drobnú časť sumy (pár centov) nezameniť, lebo by sa zbytočne zaokrúhlila na nulu pri prevode. Ak sa časom nazbiera niekoľko takýchto zbytkov, môžeme ich prevádzať napríklad na CZK (kde je strata zaokrúhlením menšia) a následne na konci prevedieme naspäť na eurá. Takúto šancu predbehnúť vzorák však nikto nevyužil ;-)

Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

Ako teda na tom môže byť PPershing v deň X? Označme najväčší dosiahnuteľný počet peňazí meny M v deň X ako $cash[X, M]$. Evidentne, riešením celej úlohy je $cash$[1.1.2011,EUR]. No a najlepšie riešenie pre $cash[X, M]$ môže vzniknúť dvoma spôsobmi:

- predchádzajúci deň sme nič nerobili, teda $cash[X, M] = cash[X - 1, M]$
- predchádzajúci deň sme previedli nejaké peniaze z meny M' (a mali sme peniaze iba v tej mene) na menu M, v tom prípade $cash[X, M] = preved(X - 1, M', M, cash[X - 1, M'])$ kde funkcia $preved$ vypočíta peniaze po prevode (správnym kurzom a s bankovým poplatkom, ak treba).

Výsledkom je väčšia z týchto hodnôt. Všimnime si teda, že keď vieme spočítať $cash[X - 1, ...]$, vieme spočítať aj $cash[X, ...]$. Postupne teda vypočítame tabuľku rozmerov $dni \times mien$, vyplnenie jednej položky trvá čas $O(mien)$. Spolu máme teda pamäťovú zložitosť $O(dni \cdot mien)$ a časovú $O(dni \cdot mien^2)$.

Úžasné, máme výsledok. Teda až na jednu drobnosť – nás nezaujímalo len to, koľko PPershing vie zarobiť, ale hlavne ako. Toto je však už iba drobná zmena do nášho programu. Zakaždým si v tabuľke nezapamätáme len $cash[X, M]$ ale aj $prev[X, M]$ ktoré bude hovoriť, z ktorej predchádzajúcej meny sme najlepší výsledok dosiahli. No a teraz nám stačí odzadu postupne prejsť po $prev$ a vypísať (v opačnom poradí) všetky potrebné finančné operácie.

Listing programu:

```
#!/usr/bin/python
import data; # postahovane data
import datumy; # cislo dna -> datum
import copy;
import sys;

###
 # preved sumu z jednej meny na druhu
 #
 # @param int hodnota hodnota v centoch (hodnota * 100)
 # @param int kurz hodnota kurzu * 10000 (kurz 1.0 = 10000)
 #
 # @returns int vysledna suma, v centoch
 ##
def preved(hodnota, kurz, interbank):
  res =  (hodnota * kurz * (1000-interbank)) / 10000 / 1000;
  return res

###
 # Vrati formatovanu hodnotu v centoch
 ##
def currencyToString(value):
  return "%d.%02d" % (value / 100, value % 100)

currencies = data.currencies;
DAYS = len(datumy.datumy); # pocet dni

# nacitaj vstup
vstup = sys.stdin.read().split(' ')[2]; # vstup je "interbank fee 2.5%"
interbank = int(vstup[0])*10 + int(vstup[2]);

# nastav pociatocne hodnoty v dynamike
initial = {}
for c in currencies:
  initial[c] = (0, "XXX");
initial['EUR'] = (1000000, "XXX");
#tabulka najlepsich vysledkov
```


Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

```
best = [];
best.append(initial);

current = initial;
for day in range(1, DAYS):
  new = {} # nove hodnoty
  for c in currencies:
    new[c] = (current[c][0], "---");
  # skus vylepsit nove hodnoty
  for c_to in currencies:
    for c_from in currencies:
      x = preved(current[c_from][0],
                 int(data.data[c_from][c_to][day-1]),
                 interbank)
      if (x > new[c_to][0]):
        new[c_to]=(x, c_from)
  best.append(new)
  current = new

# skoncili sme, ideme odzadu pozbierat najlepsiu cestu
events = [];
aktualna_mena = 'EUR';
for day in range(DAYS - 1, 0, -1):
  predchadzajuca_mena = best[day][aktualna_mena][1]
  if (predchadzajuca_mena <> '---'):
    events.append((datumy.datumy[day-1],
                   best[day-1][predchadzajuca_mena][0],
                   predchadzajuca_mena,
                   aktualna_mena))
    aktualna_mena = predchadzajuca_mena;

# a mozeme vypisat od zaciatku
events.reverse()
for event in events:
  print "day %s" % event[0]
  print "transfer %s %s %s" % (currencyToString(event[1]), event[2], event[3])
```

6. O výrobnej linke

opravoval Imp
(max. 20 bodov)

Tento príklad mal veľmi málo riešiteľov – triapolkrát menej, než predchádzajúci, či nasledujúci. Možno robilo problém predstaviť si, čo znamená postupovať nejakou stratégiou s ohľadom na to, že ju používa aj ten druhý. Skutočne, táto stratégia občas viedla k neintuitívnym a paradoxným rozhodnutiam, na ktoré niektorí riešitelia v riešení aj poukázali. Navyše, niektorí z vás upozornili aj na to, že stratégia nebola popísaná celkom jednoznačne.

Možno toto všetko viedlo k skutočnosti, že úlohu riešilo málo ľudí – a na druhej strane, každý, kto ju riešil, ju vyriešil správne a s optimálnou zložitosťou. Body sa preto strhávali iba za malé nedostatky a nikto nedostal menej ako 18 bodov.

Ako sa teda táto úloha mala riešiť? Hneď sformulovať všeobecnú stratégiu môže byť ťažké, tak si najprv rozoberme malé prípady. Ak je na páse iba jediný balíček, oplatí sa vziať si ho. V opačnom prípade by sme predsa nezískali nič. Čo ak sú tam dva balíčky? Môžeme prvý nechať tak a vziať si len ten druhý. Druhá možnosť je, že si vezmeme prvý balíček, čím sa na ťah dostane druhý hráč a ten si bude chcieť vziať druhý balíček. To teda znamená, že si môžeme vybrať len jeden z nich – a zrejme si chceme vybrať ten väčší.

Ako nám pomôžu tieto malé prípady? Tým, že aj keď rozoberáme všeobecný prípad, tieto malé prípady raz nastanú. Aj pre veľké N raz dôjde k tomu, že $N-2$ balíčkov už po

Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

linke prejde a zostanú tam posledné dva. Hráč, ktorý bude vtedy práve na ťahu, sa bude rozhodovať na základe vyššie uvedenej úvahy. Ba čo viac, všimnime si dve alternatívy, ktoré mohli nastať v prípade s dvoma balíčkami. Akokoľvek sme sa už rozhodli ohľadom prvého balíčka, previedli sme riešenie na prípad s jedným balíčkom. Rozdiel bol v tom, že v prvom prípade (ak sme prvý balíček nechali tak), sme ešte stále na ťahu, v druhom prípade (ak sme si ho vzali), je na ťahu druhý hráč.

Ak sme takto vedeli previesť riešenie pre $N = 2$ na riešenie $N = 1$, mohli by sme to aj zovšeobecniť. Riešením je teda akási indukcia, kde $N = 1$ je báza a s každým indukčným krokom zvyšujeme počet balíčkov, pre ktorý úlohu vieme riešiť. Hlavnou pointou tohto príkladu však je, že je to indukcia od konca radu – nepozeráme sa na prvý balíček, prvé dva, atď. ale na posledný balíček, posledné dva… lebo to sú tie prípady, ktoré raz nastanú.

Načítajme teda hodnoty všetkých balíčkov do poľa P – budeme to potrebovať, lebo ich chceme prechádzať v opačnom poradí. Toto nám hneď vynucuje pamäťovú zložitosť $O(N)$, ktorú si už nezhoršíme. Pomôže nám to však dosiahnuť aj časovú zložitosť $O(N)$, ktorá je zrejme optimálna, pretože tak či tak potrebujeme prečítať celý vstup. Pre jednoduchosť použime aj dve pomocné polia, ktoré budú tiež dĺžky N. Neskôr ukážeme, že tieto dve pomocné polia vlastne nepotrebujeme (a plný počet dostali tí riešitelia, ktorí si aj toto uvedomili).

V týchto pomocných poliach (označme si ich A, B) si pre každý index i spočítame nasledovné: ak máme práve na linke už len balíčky i až n, koľko najviac praliniek ešte môže získať hráč, ktorý je práve na ťahu ($A[i]$) a koľko ten druhý ($B[i]$)? Riešením potom bude $A[1]$, čo znamená „koľko praliniek ešte môže získať hráč, ktorý je na ťahu, keď sa hra začína?“, čo znamená presne „koľko praliniek môže získať Kleofáš?“.

Tieto polia vyplníme dynamicky odzadu. Zjavne platí $A[n] = P[n]$ a $B[n] = 0$, pretože ak zostáva jediný balíček, tak ten, kto je na ťahu, si ho vezme a druhému sa už nič neujde. Zamyslime sa teraz nad hodnotami $A[i]$, $B[i]$ pre $i < n$. Ak sa rozhodneme si i-ty balíček nevziať, dostaneme $A[i] = A[i + 1]$, $B[i] = B[i + 1]$, pretože takto nikto nič nezíska, hráč na ťahu zostáva, iba sa linka posunie ďalej. Ak si ho však vezmeme, dostávame $A[i] = P[i] + B[i + 1]$, $B[i] = A[i + 1]$. To znamená, že získame hodnotu i-teho balíčka, po linke sa prisunie $(i+1)$-vý balíček, ale na ťah sa dostane druhý hráč. Ten sa teraz bude snažiť získať maximum, teda $A[i+1]$, a my ešte získame $B[i+1]$, teda „maximum pre hráča, ktorý nie je na ťahu“. Keďže každý chce získať čo najviac praliniek, z týchto dvoch možností (vziať, či nevziať? To je otázka!) si vždy vyberieme tú, ktorá nám maximalizuje $A[i]$.

Tu dochádza k na začiatku spomínanej nejednoznačnosti. Čo treba robiť, ak oboma možnosťami dosiahnem rovnakú hodnotu $A[i]$? Zadanie to totiž nešpecifikuje. Uvažujme napríklad balíčky s hodnotami: 3 2 1 2. Ak si skúsite odsimulovať vyššie uvedený postup, zistíte, že pri druhom balíčku obe možnosti prinesú rovnakú hodnotu $A[2]$, ale inú hodnotu $B[2]$. To by nás teoreticky nemalo zaujímať, my sa snažíme maximalizovať vlastný zisk (hodnoty v poli A) a zisk druhého (v poli B) nás nezaujíma. Malo by teda byť jedno, ktorú možnosť si zvolíme. Avšak, prvý balíček sa jednoznačne oplatí zobrať a to spôsobí, že sa hráči na ťahu vymenia. Hodnota, ktorá nám teda vyšla v $B[2]$ tak ovplyvní hodnotu $A[1]$, ktorá nás už ale zaujíma, pretože je riešením. Na základe rozhodnutia druhého hráča tu prvý môže získať až 5 praliniek, alebo len 3. Tí z vás, ktorí nejednoznačnosť odhalili, zvolili riešenie „vezmem najľavejší balíček“, respektíve „budem milosrdný“ (čo opäť znamená vezmem najľavejší, lebo milosrdný som, keď tomu druhému dám väčší priestor na výber). To v našom riešení, keďže postupujeme odzadu, znamená „ak nie je jednoznačné, či mám alebo nemám vziať, tak vezmem“, pretože toto zabezpečí, že vezmem čím skorší balíček.

Ako bolo spomenuté vyššie, dá sa to implementovať ešte troška šikovnejšie – stačí si všimnúť, že pri výpočte $A[i]$ a $B[i]$ sa pozerám iba na $A[i + 1]$ a $B[i + 1]$, a teda pozície $i + 2$ a vyššie už nikdy nebudem potrebovať, preto si nemusím naraz pamätať celé polia, ale vždy iba poslednú vyrátanú hodnotu v A, B. Samozrejme, stále si treba pamätať celé pole

Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

P, teda tento trik pamäťovú zložitosť nezlepší. Napriek tomu som pri opravovaní považoval za dobré oceniť tých, ktorí si uvedomili, že pamäťová zložitosť je $O(N)$ kvôli poľu P a nie kvôli poliam A, B, bodom naviac.

Zamyslenie na záver: čo je na tejto stratégii také paradoxné? Vezmime si napríklad postupnosť balíčkov s hodnotami 1 1 1 1 1 1 2. Kým rozumní hráči by sa pekne striedali, aby mali čo najviac praliniek, v tomto prípade si Kleofáš vezme rovno balíček s 2 pralinkami a Zachariášovi nič neostane. Keby to totiž nespravil a zobral by si balíček s 1 pralinkou, riskoval by, že sa tak zachová Zachariáš a on sa už nedostane na ťah.

Listing programu:

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

int main() {
    int N;
    scanf("%d", &N);
    vector<int> P(N);
    for (int i = 0; i < N; i++)
        scanf("%d", &P[i]);

    int A = P[N - 1], B = 0;
    for (int i = N - 2; i >= 0; i--)
        if (P[i] + B >= A) {
            int C = A;
            A = P[i] + B;
            B = C;
        }

    printf("%d\n", A);
}
```

7. Obyvateľstvo pod kontrolou

vzorák písal Bob
(max. 20 bodov)

Tento príklad sa tešil neobvyklej priazni riešiteľov. Teda, neobvyklej vzhľadom na jeho vysoké číslo, ročné obdobie a prebiehajúce MS v hokeji. Veď minulý rok som sa takto pri podobnej príležitosti sťažoval, že prišli iba štyri riešenia, a teraz? Rovných 22! Priznávam, ten výkričník na konci predchádzajúcej vety slúži iba na zdôraznenie mojej radosti a nie ako faktoriál, ale aj tak...

Z pohľadu do výsledkovky sa dá ešte usúdiť, že prišli kadejaké riešenia, niektoré za 20 bodov, niektoré aj za menej. To by malo byť motiváciou pre tých, ktorí neposlali nič – aj keď ste nevymysleli optimálne riešenie, stále máte šancu získať nenulový počet bodov.

Fajn, povinný úvod máme za sebou, pozrime sa konečne na vzorák. Mapa kráľovstva je vlastne strom s koreňom vo vrchole 1 (hrad). Navyše máme zadané poradie, v akom budú úradníci vysielaní do svojich miest (teda najprv príde úradník číslo 1 do mesta c_1, potom druhý do mesta c_2, atď.). Na toto sa môžeme (ak sa to bude hodiť) pozerať aj z druhej strany: máme pre každé mesto zadané číslo úradníka, ktorý sa v ňom usadí.

Asi najjednoduchším riešením je postupne simulovať cestu každého úradníka z hradu do prideleného mesta. V pomocnom poli si budeme pre každý vrchol pamätať, či sa v ňom už usadil úradník. Na nájdenie cesty do c_i spustíme z vrcholu 1 prehľadávanie do hĺbky (depth-first search, DFS). V momente, keď prehľadávaním navštívime vrchol c_i, začneme sa vracať v rekurzii späť a cestou spočítame mestá s úradníkmi.

Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

Keďže takých prehľadávaní spustíme n a pri každom prejdeme prinajhoršom celý graf, časová zložitosť je $O(n^2)$. Tento odhad je tesný, lebo napríklad na grafe v tvare hviezdy (všetky ostatné mestá sú k hradu priamo pripojené hranou) toto riešenie naozaj potrebuje rádovo n^2 krokov.

Ľahko vidno zjavný nedostatok nášho riešenia: kvôli hľadaniu cesty z vrcholu 1 do vrcholu c_i musíme prejsť v najhoršom prípade celým grafom, aj keď je táto cesta oveľa kratšia. Pre každý vrchol x v strome existuje práve jedna cesta vedúca z neho do koreňa. Označme $f(x)$ vrchol, ktorý na tejto ceste nasleduje za x. Najprv si jedným DFS-kom spusteným z koreňa predrátame hodnoty $f(x)$. Následne budeme vedieť prejsť cestou z x do koreňa v čase lineárnom od jej dĺžky (začneme v x, prejdeme do $f(x)$, potom do $f(f(x))$, ..., až kým nedôjdeme do 1).

Žiaľ, časová zložitosť sa nezmení; tentoraz je najhorším prípadom graf v tvare jednej dlhej čiary. Ak ale zanedbáme čas potrebný na predpočítanie hodnôt $f(x)$, toto riešenie nie je na žiadnom vstupe pomalšie ako to predchádzajúce, ba práve naopak. Navyše sa mu darilo získavať o čosi viac bodov.

Šikovnému čitateľovi už musí byť jasné, čo nasleduje. Samozrejme, že zložitosť našich riešení zostáva na nelichotivej úrovni $O(n^2)$, keď simulujeme cestovanie každého úradníka osobitne. Ak si chceme prilepšiť, musíme to rátať nejako naraz. Nejako.

V našom úplne vzorovom riešení preto použijeme len jedno DFS z koreňa. Keď také prehľadávanie narazí na vrchol x, na zásobníku má vlastne uloženú celú cestu z vrcholu 1 až do x. Nás zaujíma, koľko je na tejto ceste takých vrcholov, v ktorých sa usadí úradník s menším číslom ako má ten v meste x (to sú presne mestá, v ktorých musí úradník smerujúci do x nechať bonboniéru).

Potrebujeme teda dátovú štruktúru, v ktorej si budeme počas DFS pamätať množinu čísel úradníkov usadených v mestách aktuálne rozpracovaných prehľadávaním. Zišla by sa nám podpora rýchleho vkladania a mazania prvkov, navyše požadujeme možnosť zistiť počet prvkov menších ako nejaké t.

Mohli by sme využiť niektorý z vyvažovaných binárnych stromov, ktoré všetky potrebné operácie vykonávajú v čase $O(\log n)$. Takéto stromčeky sa ale kódia dosť... nepohodlne. Zvyčajne si vystačíme s kontajnerom `set`, v ktorom nám tvorcovia STL implementovali červeno-čierny strom; `set` ale nepodporuje operácie „vráť k-ty najmenší prvok“, ani „koľký v poradí je tento prvok?“.

Takže nakoniec sa obrátime na o čosi menší kaliber: súčtový intervalový strom. Množinu čísel budeme mať reprezentovanú ako pole dĺžky n. Na každej pozícii budeme mať uloženú hodnotu 0 alebo 1 podľa toho, či sa dané číslo nachádza v množine. Počet čísel menších ako t zistíme ako súčet intervalu $[1, t-1]$.

Aha, takže vlastne aj intervalový strom je zbytočne veľké kladivo na tento problém. Bude nám stačiť obyčajný súčtový fínsky strom. Podrobnosti k jeho implementácii nájdete na adrese `http://www.ksp.sk/ksp2.0/wiki/Kucharka/FinskyStrom`. Snáď len spomenúť, že táto štruktúra potrebuje na každú z požadovaných operácií čas $O(\log n)$, takže celková časová zložitosť vzorového riešenia je $O(n \log n)$.

Listing programu:

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

int n;
vector<vector<int> > G;   // graf
vector<int> T, F, R;   // časy príchodov úradníkov, fínsky strom, pole výsledkov

int lastone(int x) {   // posledná jednotka binárneho zápisu x
```


Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

```
    return x & (x ^ (x - 1));
}

int sum(int x) {  // súčet prvých x prvkov
    int res = 0;
    for (int i = x; i > 0; i -= lastone(i))
        res += F[i];
    return res;
}

void update(int x, int c) {  // zväčší hodnotu x-tého prvku o c
    for (int i = x; i <= n; i += lastone(i))
        F[i] += c;
}

void DFS(int x, int p) {  // prehľadávanie z x, dostali sme sa sem z p
    R[T[x] - 1] = sum(T[x]);  // počet menších časov
    update(T[x], 1);  // zaznačíme si, že úradník príde v čase T[x]
    for (int i = 0; i < (int) G[x].size(); ++i)
        if (G[x][i] != p)  // nechceme sa vracať do otcovského vrcholu
            DFS(G[x][i], x);
    update(T[x], -1);  // a na koniec si tohto úradníka odznačíme
}

int main() {
    cin >> n;
    G.resize(n);
    for (int i = 0; i < n - 1; ++i) {
        int a, b;
        cin >> a >> b;
        --a, --b;
        G[a].push_back(b);
        G[b].push_back(a);
    }
    T.resize(n);
    for (int i = 1; i <= n; ++i) {
        int c;
        scanf("%d", &c);
        T[c - 1] = i;
    }

    F.resize(n + 1, 0);
    R.resize(n);
    DFS(0, -1);

    for (int i = 0; i < n; ++i)
        printf("%d\n", R[i]);
}
```

8. O férovej súťaži

opravoval Piťo
(max. 25 bodov)

Na úvod trochu terminológie. Graf je dvojica (V, E), kde V je množina vrcholov a E je množina hrán, kde hrana je dvojica vrcholov. Orientovaný graf má hranu definovanú ako usporiadanú dvojicu (môžeme si ju znázorniť šípkou). Hovoríme, že orientovaný graf je silne súvislý, keď medzi každou dvojicou vrcholov grafu existuje cesta s ohľadom na orientáciu hrán, a slabo súvislý, ak orientáciu neberieme do úvahy (šípky budú len čiary). Silno súvislý

Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

komponent grafu je potom najväčší (v zmysle „nezväčšiteľný“, nie najväčší zo všetkých) podgraf, ktorý je silne súvislý.

V tomto príklade ste mali za úlohu zistiť, koľko súťažiacich môže pri vhodnom poradí zápasov naisto vyhrať. Predstavme si hráčov ako vrcholy grafu a zápasy, kde súťažiaci A určite porazí súťažiaceho B, ako hranu z A do B (opačne než sa ponúka zo zadania). Pri takejto reprezentácii, ak vedie hrana z A do B a z B do C, ale napríklad nevedie z A do C, vieme poradím hier (B, C), (A, B) dosiahnuť, aby hráč A pokračoval v turnaji, no hráči C aj B už boli vyradení, teda akoby hráč A nepriamo porazil hráča C.

Vo všeobecnosti, ak v našom grafe existuje cesta od hráča X ku hráčovi Y, tak ak budú hráči na tejto ceste zápasiť v poradí od Y ku X, tak v konečnom dôsledku z nich zostane len X. Našou úlohou bude spočítať od koľkých súťažiacich existuje v tomto grafe cesta ku všetkým ostatným. V tomto momente sa naskytuje jednoduché a aj pomalé riešenie. Stačí pre každý vrchol celý graf prehľadať (či už do šírky alebo hĺbky) a zistiť, či sa z neho vieme všade dostať. Jeho časová zložitosť by bola $O(NM)$, kde N je počet vrcholov a M počet hrán.

Takýto algoritmus toho však robí veľa navyše. Môžeme si napríklad všimnúť skupinu vrcholov, kde medzi každými dvomi existuje cesta – silne súvislý komponent (ďalej len SSK). Takouto skupinou je napríklad graf v druhom vzorovom vstupe, s vrcholmi A, B, C a hranami (A, B), (B, C), (C, A). Pre ne platí, že každý vie poraziť zvyšných v skupine, no čo je dôležitejšie, stačí, aby sme vedeli zariadiť výhru turnaja jednému z nich, automaticky to vieme aj zvyšným v jeho skupine. Keby sme vedeli nájsť všetky SSK, mohli by sme nahradiť všetky vrcholy vrcholmi reprezentujúcimi len daný SSK a použiť náš predošlý algoritmus.

Čo si však môžeme všimnúť teraz je, že takto vzniknutý komponentový graf je acyklický. Ak by v ňom existoval cyklus, z každého vrcholu cyklu by bolo možné sa dostať do ostatných, čo by tiež platilo aj pre pôvodné vrcholy týchto SSK a teda by to neboli najväčšie silne súvislé podgrafy (dali by sa spojiť do jedného). Ďalej v takomto grafe existuje aspoň jeden vrchol, do ktorého nevedie žiadna hrana (z neho môže). Ak je takých viac, potom medzi týmito vrcholmi neexistuje cesta, zo žiadneho sa nedá dostať do zvyšných. (A teda výsledkom je nula. Táto situácia tiež nastane, ak pôvodný graf nebol ani slabo súvislý.) Ak je práve jeden, potom sa z neho musí dať dostať do zvyšných a hľadaný počet súťažiacich je počet vrcholov v pôvodnom grafe, ktoré boli ním nahradené, čiže počet vrcholov tohto SSK.

Ako teda na hľadanie SSK? Jednou z možností je Tarjanov algoritmus. Náš pôvodný graf budeme prehľadávať do hĺbky. Pri prehľadávaní do hĺbky si budujeme DFS strom, ktorý tvoria hrany v ňom smerujúce nadol vedúce k novoobjavovaným vrcholom. Hrany vedúce od spracovávaného vrcholu k už objavenému nazveme spätné. Tie vedú v DFS strome nahor. Pri prehľadávaní si budeme v premennej držať aktuálny čas, číslo ktoré vždy s objavením nového vrcholu zväčšíme. V každom vrchole si ho označíme (čas objavenia) a vo funkcii budeme vracať vrchol s najmenším časom aký sme videli. Prechádzať po hranách budeme len ak budú viesť do vrcholov, ktoré ešte nepatria do žiadneho SSK. Vracaný čas bude zvlášť zaujímavý v prípadoch, keď počas prehľadávania narazíme na nejakú spätnú hranu. Keď sa nám počas vracania stane, že vracaný čas bude zhodný s časom objavenia vrcholu v ktorom sme, vieme, že všetky vrcholy, ktoré sme z tohto vrcholu navštívili, sa sem vedeli dostať a zároveň sa odtiaľto nedá dostať nikam inam, takže spolu tvoria SSK. Určiť tieto vrcholy môžeme tak, že si budeme objavené vrcholy hádzať do zásobníka a keď sa dostaneme do tejto situácie, všetky vrcholy od vrchu až po ten kde sme patria do jedného SSK.

Takže zhrnutie. Vyrobíme silne súvislé komponenty, spočítame hrany vedúce do jednotlivých komponentov a spočítame komponenty bez vchádzajúcich hrán. Ak je takých viac, výsledok je 0, ak je len jeden, potom je výsledkom počet vrcholov daného silne súvislého komponentu.

Tarjanov algoritmus je len prehľadávanie do hĺbky so zásobníkom na vrcholy. Každou hranou pôjdeme nanajvýš raz, každý vrchol navštívime a dáme do zásobníka práve raz. Pri

Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

počítaní vchádzajúcich hrán prejdeme každú raz a potom ešte raz každý komponent. Celková časová a aj pamäťová zložitosť je $O(N + M)$.

Listing programu:

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

int a, b, N, cas = 0;      // počet vrcholov a súčasný čas v DFS
vector<vector<int> > G;   // náš graf, na G[i] sú tí, ktorých i porazí
vector<int> casy;         // časy objavenia vrcholov v DFS
stack<int> V;             // zásobník vrcholov pre Tarjanov algoritmus

vector<int> SSK;          // ID SSK pre všetky vrcholy
vector<int> SSK_v;        // počet vrcholov v danom SSK
int SSK_p = 0;            // počet SSK

int DFS(int v) {          // DFS s Tarjanovým algoritmom
    casy[v] = cas++;
    V.push(v);

    // zistíme najskôr objavený vrchol dosiahnuteľný z tohto vrcholu
    int minc = casy[v];
    for(int i = 0; i < G[v].size(); ++i) {
        int w = G[v][i];
        if(SSK[w] == -1) { // vrcholy už priradené SSK ignorujeme
            if(casy[w] == 0) minc = min(minc, DFS(w));
            else minc = min(minc, casy[w]);
        }
    }

    // ak je to tento, všetky vrcholy v DFS strome pod ním s ním tvoria SSK
    if(minc == casy[v]) {
        SSK_v.push_back(0);
        while(SSK[v] == -1) {
            SSK[V.top()] = SSK_p;
            ++SSK_v[SSK_p];
            V.pop();
        }
        ++SSK_p;
    }

    return minc;
}

int main() {
    cin >> N;

    G.resize(N);
    SSK.resize(N, -1);
    casy.resize(N, 0);

    for(int i = 0; i < N; ++i) {
        cin >> a;
        for(int j = 0; j < a; ++j) {
            cin >> b;
            G[b-1].push_back(i);
        }
```


Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

```
    }

    // spustíme DFS
    for(int i = 0; i < N; ++i) if(casy[i] == 0) DFS(i);

    // spočítame hrany vchádzajúce do jednotlivých SSK zvonka
    // hrany berieme z pôvodného grafu, takže nás zaujímajú
    // len tie čo vedú medzi rôznymi SSK
    vector<int> SSK_dnu(SSK_p, 0);
    for(int i = 0; i < N; ++i) {
        for(int j = 0; j < G[i].size(); ++j) {
            int w = G[i][j];
            if(SSK[i] != SSK[w]) ++SSK_dnu[SSK[w]];
        }
    }

    // spočítame do koľko SSK nevchádzajú hrany
    int kolko = 0, ktory;
    for(int i = 0; i < SSK_p; ++i) {
        if(SSK_dnu[i] == 0) {
            ++kolko;
            ktory = i;
        }
    }

    if(kolko == 1) cout << SSK_v[ktory] << endl;
    else cout << 0 << endl;
}
```

1::9. Tichý lov

opravoval Usamec
(max. 25 bodov)

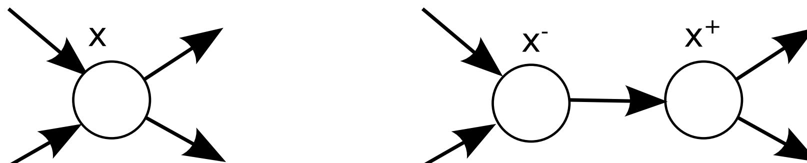
Riešenie začneme podobne ako ôsmy príklad minulej série. Takže opäť z nášho grafu vymažeme hrany, po ktorých nejde žiadna najkratšia cesta, a následne z neho spravíme orientovaný graf (smer hrany vyjadruje smer najkratšej cesty).

V tomto grafe chceme nájsť najmenšiu množinu **vrcholov** iných ako A, B, po ktorých odstránení v grafe nebude existovať cesta z A do B. Pre človeka, ktorý má trochu hlbšie znalosti grafových algoritmov, je táto úloha pomerne typická. My si teraz ukážeme, ako ju riešiť.

Redukcia z odstraňovania vrcholov na odstraňovanie hrán

Rovno si povedzme, že úloha, kde by sme hľadali čo najmenšiu množinu **hrán**, po ktorých odstranení nebude existovať cesta z A do B, je jednoduchšia.

Preto prvá vec, ktorú urobíme, bude prevod z jednej úlohy na druhú. Každý vrchol x rozdelíme na dva vrcholy x^+ a x^- nasledovne: Do x^- budú vstupovať všetky hrany, ktoré pôvodne išli do x; z x^+ budú vychádzať všetky hrany, ktoré pôvodne išli z x. A vyrobíme novú hranu, ktorá vedie z x^- do x^+.



Keď v takomto prerobenom grafe nájdeme najmenšiu množinu hrán, ktorá oddelí A^+ od B^-, tak z nej množinu rozdeľovacích vrcholov zostrojíme jednoducho: pre každú hranu vezmeme jeden susedný vrchol z pôvodného grafu (pre hrany z x^+ do y^- je jedno či vezmeme

Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

x alebo y, pre hranu z x^- do x^+ samozrejme vezmeme x). Takto spolu s vrcholmi zrušíme v grafe príslušné hrany a máme rozpojený graf. Za domácu úlohu si dokážte, že vzniknutá množina vrcholov bude fakt najmenšia.

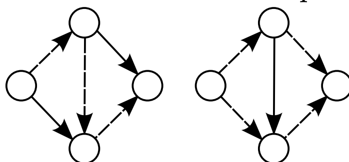
Minimálny rez a maximálny počet disjunktných ciest

To, čo vlastne hľadáme, sa ináč volá minimálny rez grafu. Rez grafu je rozdelenie grafu na dve množiny S, T, pričom vrchol A je v množine S a vrchol B je v množine T. Veľkosťou rezu nazývame počet hrán vedúcich z S do T. Našou úlohou je nájsť rez s najmenšou veľkosťou – minimálny rez. Zjavne keď hrany vedúce z S do T z grafu odstránime, tak cesta vedúca z A do B existovať nebude.

Disjunktnými cestami budeme volať také dve cesty, ktoré nemajú spoločnú hranu. Zjavne medzi A, B nie je viac disjunktných ciest ako je veľkosť minimálneho rezu. Dá sa dokázať aj to, že minimálny rez nie je väčší ako počet disjunktných ciest. Takže naša úloha sa zvrháva na hľadanie maximálneho počtu disjunktných ciest medzi A^+ a B^-.

Zlepšujúce cesty

Otázka znie: ako hľadať disjunktné cesty? Môžeme skúsiť greedy spôsob (pridávať nové cesty, kým sa nedá pokračovať), ale obrázok nás ľahko presvedčí, že to nie je dobrý prístup.



Predstavme si, že máme nájdenú cestu C ako na obrázku vľavo (vyznačená prerušovanými čiarami) a hľadáme novú cestu D. To, čo by sme potrebovali, je zrušiť jednu hranu cesty C, prepojiť začiatok cesty D s pokračovaním cesty C a nájsť ceste C nové pokračovanie. Výsledok bude ako na obrázku vpravo. Takže to, čo potrebujeme, je dovoliť pri hľadaní novej cesty vracať sa po už nájdených cestách. Zlepšujúcou cestou nazveme cestu, ktorá ide iba v smere hrán – pokiaľ nimi žiadna cesta nevedie – a proti smeru hrán – pokiaľ tam nejaká cesta vedie. Použitie zlepšujúcej cesty z A do B (po hranách, ktorými sme sa vracali, cesta viesť nebude a po hranách, ktorými sme išli vpred, budú viesť nové časti ciest) nám zjavne zvýši počet objavených disjunktných ciest o jedna.

Teraz si ešte rýchlo dokážeme tvrdenie z predchádzajúcej časti. Chceme dokázať, že minimálny rez nie je väčší ako maximálny počet disjunktných ciest. Zoberme graf, kde máme maximálny počet disjunktných ciest. Označme množinou S všetky vrcholy, kam sa vieme dostať zlepšujúcou cestou z A (do B sa dostať nevieme, lebo ináč by sme vedeli zvýšiť počet disjunktných ciest). V množine T budú zvyšné vrcholy. Zjavne všetky hrany vedúce z S do T sú obsadené nejakou cestou, takže veľkosť rezu je rovná počtu disjunktných ciest. Všimnite si, že tento dôkaz nám dáva priamy návod na nájdenie rezu.

Zhrnutie

Tak a úlohu už máme vyriešenú. Zhrňme si všetky kroky.

- Najprv – spôsobom rovnakým ako v ôsmej úlohe – zostrojíme orientovaný graf, na ktorom sú všetky najkratšie cesty vedúce z A do B. Toto vieme v čase $O((N+M)\log N)$.
- Každý vrchol rozdelíme na dva vrcholy a hrany poprepájame. Toto vieme v čase $O(N+M)$.
- Budeme postupne v grafe hľadať zlepšujúce cesty, kým sa to dá. Jedno hľadanie zlepšujúcej cesty trvá $O(N+M)$. Veľkosť rezu nebude väčšia ako $O(N)$, takže celkový čas je $O(N(N+M))$.
- Nakoniec ešte nájdeme množinu vrcholov, kam sa zo začiatku vieme dostať zlepšujúcou cestou. Hrany medzi touto množinou a zvyškom tvoria rez.
- Preložíme hrany na pôvodné vrcholy.

Takže máme celkový čas $O(N(N+M))$ a pamäťové nároky $O(N+M)$.

Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

Poznámky

Existuje trochu všeobecnejšia úloha ako tá, ktorú sme teraz riešili. Predstavte si, že by sme na hrany dali váhy. A chceli sme nájsť rez s čo najmenšou váhou hrán. Potom by sme hľadali maximálny tok zo začiatku ku koncu, čiže každá hrana by mala obmedzenie na kapacitu, ktorá ňou tečie (resp. počet ciest, ktoré tadiaľ môžu viesť) a vo vrcholoch sa žiadny tok hromadiť nemôže (čo tam pritečie, to tam odtečie). Riešenie takejto úlohy je veľmi podobné. Budeme tiež hľadať zlepšujúce cesty (tentokrát na prechod spätne nám stačí, že tam niečo tečie) a tlačiť nimi toho čo najviac naraz. Dá sa ukázať, že pokiaľ hľadáme najkratšie zlepšujúce cesty, tak časová zložitosť riešenia neprekročí $O(NM^2)$. Tento prístup môžeme využiť aj pri riešení našej úlohy, na hranách z x^- do x^+ dáme kapacitu 1 a na zvyšných dáme kapacitu nekonečno. Pripomínam, že odhad zložitosť bude stále $O(NM)$, keďže riešenie je špeciálny prípad maximálneho toku.

Listing programu:

```
#include <cstdio>
#include <vector>
#include <queue>
#include <algorithm>
#include <tr1/unordered_map>
using namespace std;
using namespace std::tr1;

#define INF 2000000000
int N, M;

struct Vrchol {
  vector<pair<int, int> > next;   // susedia - vrchol, cena
  int d[2];                        // najlepsia najdena vzdialenost
                                   // 0 - od startu, 1 - od ciela
  Vrchol() {
    d[0] = d[1] = INF;
  }
};

vector<Vrchol> v;

// Nechceme kopirovat kod, napiseme si proceduru
void dijkstra(int start, int index) {
  // Vo fronte mame dvojicu (vzdialenost, index)
  priority_queue<pair<int, int>, vector<pair<int, int> >, greater<pair<int, int> > > fr;
  fr.push(make_pair(0, start));
  while (!fr.empty()) {
    int dist = fr.top().first;
    int x = fr.top().second;
    fr.pop();
    if (v[x].d[index] <= dist) continue; // Skontrolujeme, ci sme vrchol uz neuzavreli
    v[x].d[index] = dist;
    for (int i = 0; i < v[x].next.size(); i++) {
      fr.push(make_pair(dist + v[x].next[i].second, v[x].next[i].first));
    }
  }
}

struct Vrchol2 {   // Vrchol v prerobenom grafe
  unordered_map<int, int> next;  // Kluc je susedny vrchol,
  unordered_map<int, int> prev;  // Hodnota vyjadruje ci tadial vedie cesta
  int prev_bfs;
```


Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

```
};

vector<Vrchol2> v2;

int main() {
  scanf("%d %d", &N, &M);
  v.resize(N);
  for (int i = 0; i < M; i++) {
    int a, b, c;
    scanf("%d %d %d", &a, &b, &c);
    a--; b--;
    v[a].next.push_back(make_pair(b, c));
    v[b].next.push_back(make_pair(a, c));
  }
  int start, end;
  scanf("%d %d", &start, &end);
  start--; end--;
  dijkstra(start, 0);
  dijkstra(end, 1);

  // 2X je X-, 2X+1 je X+
  v2.resize(2*N);
  for (int i = 0; i < N; i++) {
    v2[2*i].next[2*i+1] = 0;
    v2[2*i+1].prev[2*i] = 0;
    for (int j = 0; j < v[i].next.size(); j++) {
      if (v[i].d[0] + v[i].next[j].second + v[v[i].next[j].first].d[1] ==
          v[end].d[0]) {
        v2[2*i+1].next[2*v[i].next[j].first] = 0;
        v2[2*v[i].next[j].first].prev[2*i+1] = 0;
      }
    }
  }

  while(true) {
    for (int i = 0; i < 2*N; i++) {
      v2[i].prev_bfs = -1;
    }

    // BFSkom najdeme zlepsujucu cestu
    queue<int> fr;
    fr.push(2*start+1);
    v2[2*start+1].prev_bfs = -2;
    while (!fr.empty()) {
      int x = fr.front(); fr.pop();
      for(unordered_map<int, int>::iterator it = v2[x].next.begin();
          it != v2[x].next.end(); ++it) {
        if (it->second == 1)
          continue;
        if (v2[it->first].prev_bfs != -1)
          continue;
        v2[it->first].prev_bfs = x;
        fr.push(it->first);
      }
      for(unordered_map<int, int>::iterator it = v2[x].prev.begin();
          it != v2[x].prev.end(); ++it) {
        if (it->second == 0)
          continue;
        if (v2[it->first].prev_bfs != -1)
          continue;
```


Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

```
      v2[it->first].prev_bfs = x;
      fr.push(it->first);
    }
  }
  // Skontrolujeme, ci sme nasli cestu
  if (v2[2*end].prev_bfs == -1)
    break;

  // Prejdeme po najdenej ceste spat a prepiseme co treba
  int cur = 2*end;
  while (cur != 2*start+1) {
    int back = v2[cur].prev_bfs;
    if (v2[cur].prev.count(back)) {
      v2[cur].prev[back] = 1;
      v2[back].next[cur] = 1;
    }
    if (v2[cur].next.count(back)) {
      v2[cur].next[back] = 0;
      v2[back].prev[cur] = 0;
    }
    cur = back;
  }
}
// Najdeme rezove hrany
for (int i = 0; i < 2*N; i++) {
  if (v2[i].prev_bfs == -1) continue;
  for (unordered_map<int,int>::iterator it = v2[i].next.begin();
       it != v2[i].next.end(); ++it) {
    if (v2[it->first].prev_bfs == -1) {
      // Vrchol it->first/2 zablokujeme
      // Vybereme si vzdialenejsi vrchol preto, aby sme nezablokovali
      // hned start
      printf("%d\n", it->first/2+1);
    }
  }
}
}
```

1::10. Totálna fúzia

nebolo treba opravovať
(max. 25 bodov)

Túto úlohu nevyriešil nikto. Teda, my jej riešenie poznáme a pretože si myslíme, že je celkom zaujímavé, neprezradíme ho hneď teraz. Takže sa ešte stále môžete snažiť vymyslieť ho.

2::9. Temná párty

opravoval Usamec
(max. 25 bodov)

Pri riešení takejto úlohy skoro nič nepokazíme, keď si najprv zadaných ľudí utriedime podľa jednej vlastnosti.

Keď sa teraz pozrieme na hodnoty druhej vlastnosti, tak sme dobrých účastníkov vybrali práve vtedy, keď tieto hodnoty sú rastúce. Takže chceme hľadať najdlhšiu rastúcu podpostupnosť. Tento problém sa už v tomto ročníku vyskytol a jeho riešenie si môžete prečítať vo vzoráku ôsmeho príkladu z druhej série.

Ešte sú tu dva malé zádrhy. Potrebujeme vyriešiť ľudí, ktorí majú rovnakú hodnotu prvej vlastnosti. Na to, aby sme dvoch týchto ľudí nevybrali, tak budeme triediť vzostupne podľa

Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

prvej vlastnosti a zostupne podľa druhej. Takto ich nemôžeme vybrať, lebo by sme porušili rastúcosť druhej vlastnosti.

Druhý problém je výpis konkrétnych vybraných prvkov. Ten vyriešime tak, že v každom bode si okrem najlepšej aktuálnej dĺžky budeme pamätať aj najbližšie menšie miesto, ktoré máme vybrať, aby sme dosiahli najlepšiu dĺžku.

Výsledné riešenie má časovú zložitosť $O(N \lg N)$ a pamäťovú $O(N)$.

Listing programu:

```
#include <algorithm>
#include <cstdio>
#include <map>
#include <vector>
using namespace std;

#define INF 2000000100

int main() {
  int N;
  vector<pair<pair<int, int>,int> > data; // najprv parametre potom povodne
                                            // miesto, kvoli triedeniu
  scanf("%d", &N);
  data.resize(N);
  for (int i = 0; i < N; i++) {
    scanf("%d %d", &data[i].first.first, &data[i].first.second);
    data[i].first.second *= -1;  // Prevratime druhu suradnicu, aby sme mohli
                                 // spravne triedit
    data[i].second = i+1;
  }
  data.push_back(make_pair(make_pair(INF, -INF),-1));  // Sentinel
  sort(data.begin(), data.end());

  // Hladanie najdlhsej rastucej podpostupnosti
  vector<int> odkial; // Pre spatnu rekonstrukciu
  odkial.resize(data.size());
  map<int, int> lis;  // Hodnota -> kde sa nachadza
  lis[0] = -1;  // Sentinel
  for (int i = 0; i < data.size(); i++) {
    map<int, int>::iterator it2 = lis.lower_bound(-data[i].first.second);
    map<int, int>::iterator it = it2;
    it2--; // Mensi prvok
    odkial[i] = it2->second;
    if (it != lis.end())
      lis.erase(it);
    lis[-data[i].first.second] = i;
  }

  // Vypisanie vystupu
  vector<int> output;
  int cur = odkial[N];
  while (cur != -1) {
    output.push_back(data[cur].second);
    cur = odkial[cur];
  }
  printf("%d\n", output.size());
  for (int i = output.size()-1; i > 0; i--)
    printf("%d ", output[i]);
  printf("%d\n", output[0]);
}
```


Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

vzorák písal Zemčo
(max. 25 bodov)

2::10. Trápenie stromami

S trápením stromami sa trápil len jeden riešiteľ. Škoda, že vás nebolo požehnanejšie, pretože príklad to bol veľmi príjemný, pekný a relatívne nie ťažký (má všetky predpoklady ideálneho životného partnera). Jediné, čo mohlo odradiť, je netriviálna matematika v pozadí. Aby ste sa nesťažovali, vedzte, že tento príklad je poľského pôvodu a (ne)riešili ste jeho zľahčenú verziu :-).

Na úvod vzoráku si poďme situáciu trošku zjednodušiť a uvažovať počítanie výslednej hodnoty priamo a nie ako zvyšok po delení 1 000 000 009. Toto číslo označme M, nech šetríme atramentom a našimi lesmi, aj keď tá zbytočná poznámka o šetrení minula viac lesa ako všetky výskyty čísla M v ďalšom texte. Pozrieme sa na úlohu čisto matematicky a nebudeme uvažovať obmedzenie našich pozemských výpočtových zariadení pamätať si ľubovoľne veľké čísla. V tomto vzoráku predpokladáme, že botanické pojmy ako strom, podstrom, list a koreň sú čitateľovi známe. Obdobne by bolo dobré, keby čitateľ vedel, čo je to binomický koeficient $\binom{n}{k}$, že kombinatoricky je to počet možností, ako z n-prvkovej množiny vybrať k prvkov a že číselne (algebraicky) je to rovné $\frac{n!}{k!(n-k)!}$.

Máme danú štruktúru stromu. Celý algoritmus bude prebiehať prehľadávaním tohto stromu do hĺbky a postupným vyhodnocovaním *zdola hore*. Každý podstrom si môžeme predstaviť ako samostatný strom, ktorý pozostáva z nejakého počtu vrcholov p. Preto môžeme uvažovať podúlohu o počte možností rozdelenia čísel $1, 2, \dots, p$ v tomto podstrome. Ak máme podstrom s koreňom vo vrchole v, potom tento počet rozdelení označme $P(v)$. Pre strom pozostávajúci z jediného vrcholu je ľahko vidno, že odpoveď je 1.

Vo všeobecnej situácii máme daný podstrom s koreňom u a k synmi $v_1, v_2, \dots, v_k$, ktorí sú koreňmi podstromov s $t_1, \dots, t_k$ vrcholmi. Uvažujme, že rekurzívne pracujúce prehľadávanie do hĺbky už spočítalo počet kombinácii v podstromoch a teda vieme hodnoty $P(v_1), P(v_2), \dots, P(v_k)$ a rovnako vieme aj hodnoty $t_1, t_2, \dots, t_k$. Prvá vec, ktorú chceme zistiť, je počet vrcholov v podstrome s koreňom u. To nie je ale nič iné ako súčet hodnôt t_i a ešte plus jedna, pretože potrebujeme započítať aj samotný vrchol u. Označme tento počet vrcholov T.

Teraz predpokladajme, že v celom tomto strome budeme umiestňovať hodnoty $1, 2, \dots, T$. Je zjavné, že číslo 1 bude musieť byť použité v koreni, inak by nemohla byť dodržaná vlastnosť haldy. Ostali nám čísla $2, \dots, T$, tieto budeme rozmiestňovať medzi podstromy synov.

Majme situáciu, v ktorej má vrchol u práve dvoch synov v_1,v_2 s počtami vrcholov t_1 a t_2 a rozmiestňujeme hodnoty $2, \dots, t_1 + t_2 + 1$. Uvažujme ľubovoľné rozdelenie týchto hodnôt na dve množiny S_1 a S_2, pričom v S_1 je t_1 čísel a sú určené do podstromu pod v_1 a v S_2 je zvyšných t_2 čísel a sú určené do podstromu pod v_2. Koľko máme možností na ich rozmiestnenie v dvoch podstromoch pod synmi?

Už predtým sme vypočítali, že možností, ako rozdeliť čísla $1, \dots, t_1$ v podstrome pod v_1 je práve $P(v_1)$. Predstavme si ľubovoľné korektné rozmiestnenie čísel z S_1 do tohto podstromu. Prečíslujme teraz tieto čísla na čísla z rozsahu $1, \dots, t_1$ tak, aby sme zachovali relatívne usporiadanie. To ide, keďže máme v oboch prípadoch t_1 rôznych čísel. Toto prečíslovanie je dokonca jednoznačné. Napríklad, čísla 1,7,5 by sme prečíslovali na 1,3,2. Zjavne takto dostaneme nejaké rozmiestnenie čísel $1, \dots, t_1$. Toto rozmiestnenie bude korektné, pretože ak bol každý otec menší ako jeho synovia v pôvodnom usporiadaní, bude to platiť aj po prečíslovaní. No a podobná úvaha sa dá spraviť aj opačne: každému korektnému označeniu vrcholov číslami $1, \dots, t_1$ sa dá priradiť jednoznačné označenie číslami z S_1 tak, aby sa zachovali relatívne vzťahy medzi hodnotami vo vrcholoch, čo znova zaručuje korektnosť nového číslovania.

Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

Záver: dostávame vzťah *jedna ku jednej* a preto je počet možných korektných rozmiestnení čísel z S_1 v podstrome pod v_1 práve $P(v_1)$.

A koľko je rozmiestnení pre celý podstrom pod u? Rozmiestňovanie pod dvoma synmi je nezávislé. Keď nejak rozmiestnime hodnoty z S_1 pre prvého syna v prvom podstrome, potom v druhom podstrome môže byť ľubovoľné korektné rozmiestnenie čísel z S_2. Preto je to celé súčin počtu možností, koľkými rozdeliť čísla medzi množiny S_1 a S_2 krát počet možností na rozmiestnenie v prvom podstrome krát počet možností na rozmiestnenie v druhom podstrome:

$$P(u) = \binom{t_1 + t_2}{t_1} \cdot P(v_1) \cdot P(v_2)$$

Takto sme započítali správny počet korektných rozmiestnení, pretože každé korektné rozdelenie zodpovedá práve jednému z rozdelení na dve podmnožiny S_1 a S_2 a rozmiestnení týchto hodnôt nejakým jedným korektným spôsobom spomedzi počtu $P(v_1)$, respektíve $P(v_2)$. Naopak, každé takéto rozdelenie a priradenie zodpovedá jednému korektnému rozdeleniu v celom podstrome pod u.

A pre všeobecnú situáciu s ľubovoľným počtom synov je to podobné. Ak máme k synov, potom je to počet možností, ako rozdeliť všetky čísla podstromu medzi k množín daných veľkostí krát počet rozmiestnení vybraných čísel v rámci jednotlivých podstromov. Počet rozdelení je rovný nasledovnej hodnote:

$$P(u) = \frac{(T-1)!}{t_1! \cdot t_2! \cdot \ldots \cdot t_k!} \cdot P(v_1) \cdot P(v_2) \ldots P(v_k)$$

Úvahy o správnosti tohto vzťahu sú obdobné ako tie v modelovom prípade pre dvoch synov pod u. Možná kombinatorická interpretácia je, že vezmeme všetkých $T-1$ čísel a tie zoradíme všetkými možnosťami. Potom vezmeme v tomto usporiadaní prvých t_1 čísel a priradíme prvej hromádke. Potom ďalších t_2 čísel, tie priradíme druhej hromádke, a tak ďalej. A prečo delíme? Pretože nám nezáleží na poradí pridávania do hromádky, ale len na množine čísel v nej. Preto je jedno, či usporiadanie začína 2,1,4 alebo 4,2,1, ak prvé tri čísla vyberáme do prvej hromádky. Preto budú niektoré usporiadania z nášho pohľadu rovnaké a potrebujeme zo všetkých rovnakých započítať práve jedno. No keďže t_1 prvkov môžeme usporiadať $t_1!$ spôsobmi, potom treba príslušnú hodnotu vydeliť práve $t_1!$ (a analogicky aj ostatnými faktoriálmi).

Odstavec pomimo. Môžeme sa na to kombinatoricky dívať aj inak: máme $T-1$ hodnôt a vyberáme z nich t_1 do prvej hromady. Potom ostalo $T-1-t_1$ a vyberáme t_2 do druhej hromady a tak ďalej. Dostávame výraz:

$$\binom{T-1}{t_1} \cdot \binom{T-1-t_1}{t_2} \cdot \binom{T-1-t_1-t_2}{t_3} \ldots \binom{T-1-t_1-\ldots-t_{k-1}}{t_k} \cdot P(v_1) \ldots P(v_k)$$

Ako ľahké cvičenie sa presvedčte, že táto hodnota je v skutočnosti presne tá, ktorú sme vymysleli vyššie. Presvedčíte sa o tom tak, že si rozpíšete binomické koeficienty pomocou faktoriálov a výraz upravíte.

Vymysleli sme vzťah na výpočet počtu rozdelenia čísel $1, \ldots, T$ v podstrome s T vrcholmi. Je ľahko vidno, že sa nám ho bude dariť v konkrétnom vrchole vypočítať v čase úmernom počtu synov: rekurzívne sa zavoláme na synov a hodnoty, ktoré sa nám budú vracať (počet vrcholov v podstrome pod synom v_i a počet kombinácii $P(v_i)$) budeme používať na výpočet výslednej hodnoty. Samozrejme, že potrebujeme poznať faktoriály a preto si ich predrátame. Keďže ale nepotrebujeme faktoriál čísla väčšieho ako N, stačí nám čas $O(N)$. Viac sa o tom dozviete v ďalšom odstavci.

Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

Ako to vypočítať ako zvyšok po delení M?

Operácii *zvyšok po delení* M hovoríme múdro a kratšie *modulo* M. Ako sa neskôr presvedčíme, požiadavka o odpovedi modulo M nám šetrí potrebu pamätať si ľubovoľne veľké čísla, čo je vo všeobecnosti veľká oštara. Označme operáciu modulo pomocou symbolu percenta %. Napríklad, $11 \% 4 = 3$. Keď budeme v tejto časti vzoráku uvádzať nejaké rovnice, spravidla tým myslíme rovnosť v modulárnej aritmetike. Napríklad, ak sa pohybujeme v priestore modulo 4, potom $11 = 3$, ale $11 \neq 2$. Práca s číslami modulo M má niektoré dosť príjemné vlastnosti. Keď chceme poznať súčin dvoch obrovských čísel modulo M, stačí nám poznať hodnotu týchto čísel modulo M, tie vynásobiť (a odpoveď prípadne ešte znova zmodulovať). Matematickejšie:

$$(p_1 \cdot p_2) \% M = ((p_1 \% M) \cdot (p_2 \% M)) \% M$$

Napríklad, ak chceme poznať výsledok $(45127 \cdot 31243) \% 5$, tak nepotrebujeme v skutočnosti poznať celé (relatívne) veľké čísla, ale keďže $45127 \% 5 = 2$ a $31243 \% 5 = 3$, potom $(45127 \cdot 31243) \% 5 = (2 \cdot 3) \% 5 = 6 \% 5 = 1$. Kto chce, nech si uvedený fakt na dva riadky dokáže. Teda tento fakt ukazuje, že je jedno, či si pamätáme celé čísla alebo len ich zvyšky po delení M.

Ako už naznačuje vzťah vyššie, okrem násobenia potrebujeme aj delenie modulo M. Čo je to vlastne delenie modulo M? Jednému by mohla byť odpoveď na túto otázku veľmi ťažko predstaviteľná. My vlastne pracujeme s celými číslami $0, 1, \ldots M-1$, tak ako je možné vypočítať napríklad $3/5$? Nuž, na vysvetlenie použijeme množinu čísel, ktoré sa objavujú pri operáciách modulo 7. Uveďme najprv tabuľku s výsledkami operácie násobenia na tejto množine:

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6
2	0	2	4	6	1	3	5
3	0	3	6	2	5	1	4
4	0	4	1	5	2	6	3
5	0	5	3	1	6	4	2
6	0	6	5	4	3	2	1

Príslušné políčko sme dostali ako výsledok operácie násobenia dvoch čísel a potom vypočítali zvyšok po delení siedmimi. Potreba zistiť x v rovnici $\frac{a}{b} = x$ je rovnaká ako potreba zistiť x v rovnici $a = x \cdot b$ (možno nie je úplne zjavné, prečo a ako toto môžeme a chceme urobiť, avšak nebudeme zachádzať do prílišných detailov). Čiže ekvivalentná formulácia je: čím mám vynásobiť b, aby som dostal a (modulo M, samozrejme)? Napríklad, nahliadnuc do tabuľky vyššie, vidíme, že $\frac{5}{3} = 4$. Totiž, ak chceme trojku vynásobiť x, aby bol výsledok 5, potom je vhodné x rovné 4 (skutočný výsledok je 12, zvyšok po delení 7 je práve 5). Existuje samozrejme veľa takých čísel, avšak tabuľka prezrádza, že v rozmedzí 0 až 7 je také práve jedno. Toto, mimochodom, platí pre každé možné a a b okrem nuly.

Máme teda rovnicu $a = x \cdot b$. Predstavme si, že k obom stranám prinásobíme nenulový prvok, ktorý označíme ako b^{-1} a pre ktorý bude platiť $b \cdot b^{-1} = 1$. Z pravej strany rovnice $a \cdot b^{-1} = x \cdot b \cdot b^{-1}$ odstránime $b \cdot b^{-1}$ (pretože jedna krát x je rovnaké x aj v modulárnom násobení) a dostávame $a \cdot b^{-1} = x$. To znie super, pretože sme konečne osamostatnili hľadané x a zároveň už pracujeme len s operáciou násobenia.

Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

Tajomné b^{-1} budeme nazývať *inverzný prvok* k prvku b. Vo všeobecnosti ich môže byť veľa (alebo žiaden), avšak pohľad do tabuľky prezrádza, že ak sa pohybujeme v aritmetike modulo 7, ku každému číslu okrem 0 existuje práve jeden. V tabuľke nižšie si môžete pozrieť inverzné prvky pri počítaní modulo 7. Tento fakt platí pre všetky prvočísla, dôkaz si na tomto mieste odpustíme. Aj naše M zo zadania je prvočíslo! Takže už to začína niekam smerovať: ak budeme chcieť deliť číslom b, stačí miesto toho vynásobiť jeho (existujúcim a unikátnym) inverzným prvkom.

Hodnota	0	1	2	3	4	5	6
Inverzný prvok	-	1	4	5	2	3	6

Zostáva otázka, ako efektívne zistiť inverzný prvok k danému číslu. Jeden spôsob je len jednoducho číslo skúšať násobiť 1, 2, ... až kým niekedy nevyjde jedna – daný súčiniteľ by bol hľadaný inverzný prvok. Toto nás ale stojí čas $O(M)$, čo je v našom prípade neprípustné.

Nastúpiť teda musí sofistikovanejšia teória, ktorú nebudeme rozoberať do hĺbky. Jeden zo spôsobov je využiť Malú Fermatovu vetu. Nech p je prvočíslo, a je celé číslo medzi 0 a $p-1$ vrátane. Potom

$$a^p \% p = a$$

Ak sa pozrieme na uvedenú rovnicu a vykrátime obe strany číslom a (premyslite si, že môžeme, aj keď sme v aritmetike modulo p), potom dostávame upravenú verziu

$$(a^{p-2} \cdot a) \% p = 1$$

Takže vidno, že k ľubovoľnému číslu a je inverzný prvok jeho vlastná mocnina na $p-2$ (umocnená modulárne). No a umocniť hodnotu na $p-2$ vieme v čase $O(\log p)$ (kto vie ako, môže tento odstavec preskočiť). Dajme si znova príklad: nech chceme vypočítať číslo b^{53}. To môžeme zapísať ako $b^{32} \cdot b^{16} \cdot b^4 \cdot b^1$. Teda, je to súčin niektorých mocnín b, pričom hodnoty exponentov sú mocniny dvojky. Takže vieme začať s b^1 a postupným umocňovaním na druhú dostávať $b^2, b^4, b^8, \ldots$. Samozrejme, priebežne treba číslo aj modulovať, aby sa nám zmestilo do premennej. Popri tom si môžeme priebežne počítať výsledok a občas k nemu prinásobiť nejakú tú narastajúcu mocninu b. Odpoveď na otázku, kedy to máme prinásobovať, nám dáva binárny zápis želaného exponentu. Vieme, že 53 v binárnom zápise je `110101`. Idúc odzadu tohto zápisu (od najmenej významnej cifry) vidíme, že b^1 prinásobiť máme a b^2 nie, pretože príslušná cifra binárneho zápisu 53 je nula, a takýmto spôsobom pokračovať až do konca. No a počet krokov tohto umocňovania je úmerný dĺžke zápisu exponentu v binárnej sústave, čo je logaritmus so základom dva (zaokrúhlený nahor na celé číslo).

Zhrnutie celého algoritmu: prehľadávaním do hĺbky zisťujeme výsledok najprv pre podstromy pod synmi konkrétneho vrcholu. Potom na základe týchto výsledkov počítame hodnotu pre daný vrchol podľa uvedeného vzorca. Toto číslo vyhodnocujeme osobitne ako čitateľ a menovateľ a keď spočítame tieto hodnoty, jednoducho čitateľa vynásobíme inverzným prvkom pre vypočítaný menovateľ. V jednom vrchole trávime čas úmerný počtu hrán idúcich z neho, takže pre každú hranu vykonávame konštantný počet operácií. Hrán je presne $N-1$, spolu je to teda čas $O(N)$. Ďalších $O(N)$ potrebujeme na predpočítanie faktoriálov modulo M. Okrem toho v každom vrchole hľadáme inverzný prvok, dokopy nás to teda stojí čas $O(N \log M)$. Kým $O(M)$ bolo neprípustné, hodnota $\log M$ je v našom prípade menej ako 32, čo je celkom kamarátska konštanta. Keďže si musíme pamätať celý graf, výsledná pamäť je lineárna.

Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

Listing programu:

```
#include <vector>
#include <cstdio>
using namespace std;
typedef pair<int,long long> PILL;
#define M 1000000009
#define MAXN 1000000
int N;
//faktorialy modulo M
long long F[MAXN];
//pre kazdy vrchol zoznam synov
vector<vector<int> > G;

//vypocita exponent modulo M, v logaritmickom case
long long modexp(long long co, int nakolku){
  long long vratim = 1;
  long long exp = co;
  while(nakolku > 0){
    if (nakolku % 2){
      vratim *= exp;
      vratim %= M;
    }
    nakolku /= 2;
    exp *= exp;
    exp %= M;
  }
  return vratim;
}

//vypocet inverzneho prvku modulo M
long long inverzny(long long prvok){
  return modexp(prvok, M - 2);
}

//spocitame velkost podstromu a pocet moznosti pre koren v
PILL rataj(int v){
  long long citatel = 1;
  long long menovatel = 1;
  //sucet = pocet vrcholov v tomto podstrome
  int sucet = 0;
  for(int i=0;i<G[v].size();i++){
    //p.first - pocet vrcholov v podstrome, p.second - pocet moznosti
    PILL p = rataj( G[v][i] );
    citatel *= p.second;
    citatel %= M;
    sucet += p.first;
    menovatel *= F[p.first];
    menovatel %= M;
  }
  citatel *= F[sucet];
  citatel %= M;
  return  make_pair( sucet + 1 , ( citatel*inverzny(menovatel) ) % M);
}

int main(){
  scanf("%d ",&N);
  G.resize(N+1,vector<int>());
  F[0] = 1;
  for(int i=1;i<=N;i++) F[i] = (F[i-1]*i) % M;
```


Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09

```
  for(int i=2;i<=N;i++){
    int x;
    scanf("%d ",&x);
    G[x].push_back(i);
  }
  printf("%lld\n",rataj(1).second);
}
```


Táto práca bola podporovaná Agentúrou na podporu výskumu a vývoja na základe zmluvy č. LPP-0103-09