



**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

1. Zápis predmetov

opravovala Táňa
(max. 10 bodov)

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. Povie nám len to, že jeden študent z radu odišiel. To sa väčšine podarilo, takže Vaše riešenia sa líšili len v detailoch. Pozrime sa na to ideálne riešenie.

Najprv načítame počet udalostí a potom už len reagujeme na každú osobitne – keď príde 0, rad sa skrúti o jedna a keď príde kladné číslo, rad sa predĺži o jedného študenta. Teraz si už len musíme sledovať, kedy je najdlhší. Na to budeme mať zvlášť premennú – keď bude rad dlhší ako naše maximum, tak toto maximum prepíšeme.

Dôležité bolo zamyslieť sa, kedy treba kontrolovať maximum, či je stále najväčšie. Keď sa nad tým zamyslíme, prideme na to, že maximum môže ohroziť len rad, ktorý sa práve predĺžil, teda stačí kontrola po každom jeho predĺžení alebo tesne pred jeho skrútením. Tým sa nám znížil celkový počet vykonaných príkazov. Dalo sa ešte uvažovať ďalej: keď máme údaje za celý deň – teda začíname nulovým radom, aj ním končíme – tak je nám jedno, kedy porovnávame rad s maximom, lebo počet pričítaných a odčítaných jednotiek je rovnaký. Ak by sme však mali údaje za nejaký časový úsek a končili by sme na nenulovej dĺžke radu, je výhodnejšie porovnávať tesne pred skrútením radu, lebo rad zväčšíme viackrát ako zmenšíme, teda táto časť podmienky bude volaná menejkrát. Takéto detaily som už nehodnotila.

Časová zložitosť riešenia je lineárna, $O(n)$, lebo musím spracovať každý vstup a pritom vykonám 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 zložitou, jej dobrým odhadom a taktiež najmenším počtom vykonávaných príkazov – kontrola maxima sa nevykonala po každej zmene dĺžky radu, ale len v jednom prípade – pri zväčšení alebo zmenšení. Jeden bod som strhávala za kontrolu maxima po každej udalosti a zlý alebo chýbajúci odhad pamätevej 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:

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

```
int main()
{
```

<http://www.ksp.sk/ksp2.0>

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

```

int N,dlzka=0,max=0; // na začiatku máme rad dĺžky 0, takže aj maximum bude 0
cin >> N;
for(int i=0;i<N;i++)
{
    int vlasy;
    cin >> vlasy;
    if (vlasy>0) {dlzka++;} // ak dostaneme kladné číslo, rad sa predĺži o 1
    else {
        if (dlzka>max) max=dlzka; // ak dostanem 0, pozriem, či je rad dlhší ako max
        dlzka--; // potom ho skrátim
    }
}
if (dlzka>max) {cout << dlzka << endl;} // ak sa rad neskrátil, max mohlo zostať 0,
else {cout << max << endl;} // preto ho skontrolujem
return 0;
}

```

2. Zastavme nezbedníkov!

vzorák písal Tomi
(max. 10 bodov)

Nezbedník sa predbehol vtedy, ak je niekde neskôr po ňom menšie číslo. Takže stačí ísť vo vstupnom poli odzadu a vždy si pamätať, aké najmenšie číslo sme zatiaľ videli. Keď uvidíme číslo x , ale už sme niekde za ním videli aj číslo menšie ako x , tak sa x predbehol. Nakoniec vypíšeme tie x , čo sa predbehli, vo vzostupnom poradí.

Toto riešenie zbehne v čase $O(N)$ a má pamäť $O(N)$.

Listing programu:

```

program Nezbednici;
var n, i, 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
        if studenti[i] > minimum then predbeholsa[studenti[i]] := true;
        if studenti[i] < minimum then minimum := studenti[i];
    end;
    for i := 1 to n do begin
        if predbeholsa[i] then WriteLn(i);
    end;
end.

```

Chýba súbor prikl3.tex!!!

3. Zbytočné opevnenie

opravoval Mio
(max. 15 bodov)

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

Väčšina z Vás odovzdala veľmi podobné riešenie, ktoré spočívalo v tom, že ste si utriedili pole podľa začiatku, a následne ste veže prechádzali postupne a kontrolovali, či náhodou aktuálna veža nie je v nejakej predchádzajúcej. Kontrola prebehla tak, že ste si pamätali koniec intervalu, ktorý siahal najďalej a kontrolovali, či náhodou koniec aktuálnej veže neleží pred ním. Aby to teda bolo zaujímavejšie, tak ukážem iné riešenie tohto príkladu(hoci s horšou konštantou).

Základnou myšlienkou je toto: ak pred nejakou vežou začína viac veží ako pred ňou končí, tak sa nutne musí nachádzať vnútri intervalu nejakej veže, pretože existuje aspoň 1 veža, ktorej interval začal pred ňou a ešte neskončil. Takže stačí zistiť koľko veží pred ňou začína a koľko končí.

Ale to je celkom jednoduché. Počet veží pred ňou(resp. za ňou) je rovný jej pozícii v utriedenom poli podľa začiatku(resp. konca).

Ešte tu ostáva jeden problém(na ktorom stroskotalo mnoho z vás) a to prípad ak sú zhodné začiatky, alebo konce intervalov (alebo oboje). Bolo totiž potrebné zabezpečiť, aby v utriedenom poli boli pri rovnakých začiatkoch skôr tie, ktoré mali koniec neskôr. Pri rovnakých koncoch končili skôr tie s väčším začiatkom. A ak boli intervaly zhodné, tak ich bolo treba nejakým spôsobom do seba porovnávať(napríklad, tak, že kto má menšie číslo veže začína skôr a končí neskôr).

Časová zložitosť riešenia - 2krát tu triedim a inak nič čo by trvalo dlhšie - čiže $O(n \log n)$. Pamäťová zložitosť - $O(n)$ pamätať si treba informácie o vežiach.

Ako som hodnotil: Hodil som si mincou a potom som hodnotil nasledovne:

Plne funkčné riešenie so zložitou $O(n \log n)$ dostalo max 15 bodov. Ak ste odovzdali plne funkčné riešenie so zložitou $O(n^2)$ 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 spôsoboval pomalý sort. Ďalšie body boli strhávané za iné chyby, prípadne ak bolo riešenie nefunkčné.

To je všetko, užite si vzorák - myslím že je komentovaný dostatočne aby mu aj paskalista nemal problém rozumieť.

Listing programu:

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

struct veza //ekvivalent pascalovskeho record
{
    int zaciatok, koniec, cislo;
};

bool cmpzac(veza v1, veza v2) //porovnavatko podla zaciatku veze
{
    if(v1.zaciatok==v2.zaciatok) //rovnake zaciatky
    {
        if(v1.koniec==v2.koniec) //rovnake oboje
            return v1.cislo<v2.cislo; //veza s vacsim cislom je vnutri veze s mensim cislom
        return v1.koniec>v2.koniec; //vacsi konec zacina skor
    }
    return v1.zaciatok < v2.zaciatok;
}

bool cmpkon(veza v1, veza v2) //porovnavatko podla konca veze
{
    if(v1.koniec==v2.koniec) //rovnake konce
    {
```

```

    if(v1.zaciatok==v2.zaciatok) //rovnake oboje
        return v1.cislo>v2.cislo; //veza s vacsim cislom je vnuti veze s mensim cislom
    return v1.zaciatok>v2.zaciatok; //vacsi zaciatok konci skor
}
return v1.koniec < v2.koniec;
}

int main()
{
    int n, p;
    vector<veza> veze1, veze2; // 2 zoznamy vezi
    vector<int> pozicie; //zoznam pozicii - kde je ktora veza v poli veze 2
    cin >> n; //nacitanie poctu vezi
    veze1.resize(n); veze2.resize(n); pozicie.resize(n);
    for(int i = 0;i<n;i++) //forcyklus od 0 do n-1
    {
        veze1[i].cislo = i;
        cin >> veze1[i].zaciatok >> p >> veze1[i].koniec; //nacitanie veze
        veze2[i] = veze1[i]; //skopirovanie veze do druhého zoznamu
    }
    sort(veze1.begin(),veze1.end(),cmpzac); //usortim podla zaciatku
    sort(veze2.begin(),veze2.end(),cmpkon); //usortim podla konca

    for(int i = 0;i<n;i++) //naplnim pozicie
        pozicie[veze2[i].cislo] = i;

    bool buralsom = false;
    for(int i = 0;i<n;i++)
    {
        if(i>pozicie[veze1[i].cislo])
        {
            cout << veze1[i].cislo + 1 << endl; //vypisem vezu
            buralsom = true; //poznacim si ci som uz bural nejake veze
        }
    }

    if(!buralsom)
        cout << "Nemozno zburat ziadnu vezu." << endl;
}

```

4. Zbohatlíkom odzajtra

Vzorák PPeršing, 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 kurzovov
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%"
```

<http://www.ksp.sk/ksp2.0>

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

```
"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 prílepil. 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

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, s bankovým poplatkom ak je 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 reverznom poradí) všetky potrebné finančné operácie.

Listing programu:

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

###
# preved sumu z jednej meny na druhu
```

¹Nie 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 oplatíť 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 ;-)

```

#
# @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

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

currencies = data.currencies;
DAYS = len(datумы.datумы); # počet 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
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

# skončili 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((datумы.datумы[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])
```

5. O výrobnéj 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 zložitou. 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 linke prejde 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 spätná indukcia - nepozerala 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 náročnosť $O(N)$, ktorú si už nezhoršíme. Pomôže nám to však dosiahnuť aj časovú náročnosť $O(N)$, ktorá je zrejme optimálna, pretože tak či tak potrebujeme prečítať celý vstup. Pre jednoduchosť použijeme 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 pralíniek 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 pralíniek ešte môže získať hráč, ktorý je na ťahu, keď sa hra začína?“, čo znamená presne „koľko pralíniek 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 pralínok, 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 skúsíte 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 pralínok, 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 P , teda tento trik pamäťovú náročnosť nezlepší. Napriek tomu som pri opravovaní považoval za dobré oceniť tých, ktorí si uvedomili, že pamäťová náročnosť je $O(N)$ kvôli poľu P a nie kvôli poliam A , B , bodom navyše.

Zamyslenie na záver: čo je na tejto stratégii také paradoxné? Vezmime si napríklad postupnosť balíčkov s hodnotami 1111112. Kým rozumní hráči by sa pekne striedali, aby mali čo najviac pralínok, v tomto prípade si Kleofáš vezme rovno balíček s 2 pralínkami a Zachariášovi nič neostane. Keby to totiž nespravil a zobral by si balíček s 1 pralínkou, riskoval by, že sa tak zachová Zachariáš a on sa už nedostane na ťah.

Listing programu:

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

vector<int> P;

int main() {
    int N;
    scanf("%d", &N);
    P.resize(N);

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

    int A = P[N - 1], B = 0;

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

```

        B = C;
    }

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

    return 0;
}

```

vzorák písal Bob
(max. 20 bodov)

6. Obyvateľstvo pod kontrolou

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) pozeráť 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 vracáť v rekurzii späť a cestou spočítame mestá s úradníkmi.

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.

Lahko 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 dlhšej 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).

Potrebuje 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 obrátíme 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
    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 (vector<int>::iterator i = G[x].begin(); i != G[x].end(); ++i)
        if (*i != p) // nechceme sa vracat do otcovského vrcholu
            DFS(*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]);
}

```

Chýba súbor prikl8.tex!!!

2::9. Tichý lov

opravoval Usamec
(max. 25 bodov)

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

V tomto grafe chceme nájsť najmenšiu množinu vrcholov iných ako A, B , po ktorých odstránení graf nebude existovať cesta z A do B . Pre človeka, ktorý ma 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 odstránení nebude existovať cesta z A do B .

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^+ .

TODO: Obrázok

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

TODO: obrázok

Predstavme si, že máme nájdenú cestu C ako na obrázku vľavo 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 vracat' sa po už nájdených cestách. Zlepšujúcou cestou nazveme cestu, ktorá 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 rýchlo ešte dokážeme tvrdenie z predchádzajúcej časti. Chceme dokázať, že minimálny rez nie je väčší maximálny počet disjunktných ciest. Zoberme graf, kde máme maximálny počet disjunktných ciest. Označme množinou S všetky vrcholy, kde 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 $O(N)$.

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.

Preložime hrany na pôvodné vrcholy.

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

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 mali 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 (tento krát na prechod spatne 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 špeciálny prípad maximálneho toku.

Chýba súbor prikl10-1.tex!!!

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 súradnice, 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 v riešení ôsmomého príkladu z druhej série.

Ešte sú tu dva malé zádrhle. Potrebujeme vyriešiť ľudí, ktorý majú rovnakú hodnotu prvej vlastnosti. Na to aby sme dvoch týchto ľudí nevybrali, tak budeme triediť vzostupne podľa 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]);
}

```

7. Trápenie stromami

vzorák písal Zemčo
(max. 25 bodov)

S trápením stromami sa trápil len jeden riešiteľ. Škoda, že vás nebolo požehnamejšie, 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 nestažovali, vedzte, že tento príklad je poľského pôvodu a (ne)riešili ste jeho zľahčenú verziu :-).

Na úvod vzoráku si poďme situáciu trochu zjednodušiť a uvažovať počítanie výslednej hodnoty priamo a nie ako zvyšok po delení 1000000009. 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, nehovoriac o ďalšom zbytočnom pokračovaní tejto vety... 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.

Máme danú štruktúru stromu. Celý algoritmus bude prebiehať prehľadávaním tohto stromu 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 su koreňmi podstromov s t_1, \dots, t_k vrcholmi. Uvažujme, že rekurzívne pracujúce prehľadávanie do hĺbky už spočítalo počet kombinácií 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.

Myšlienka je, že keď vyberieme niektoré z týchto čísel a povieme, že tieto použijeme na označenie vrcholov v niektorom konkrétnom podstrome s koreňom v v_i pre nejaké i , tak možnosť, ako usporiadať tieto čísla v rámci tohto podstromu je práve $P(v_i)$. TODO

Predstavme si, že strom s vrcholom

Toto pozorovanie sa dá rozšíriť o ďalších synov nasledovnou úvahou.

Keby si niekto tento vzťah algebraicky rozpísal (to znamená miesto binomických koeficientov napísal zlomok s faktoriálmi) a začal si to zjednodušovať, príde na to, že sa to rovná číslu:

Toto je presne počet spôsobov, ako rozdeliť T-1 čísel medzi podstromy s príslušnými počtami vrcholov. TODO

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, $11 \% 4 = 3$. 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. Tento teda 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 $0, 1, \dots, M-1$, tak ako je možné vypočítať napríklad $3/5$? Nuž, na vysvetlenie použijeme množinu čísel, ktoré sa objavujú pri operáciách modulo 7. Uvedme 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 $a = x \cdot b$. Čiže ekvivalentná formulácia je: čím mam vynásobiť b , aby som dostal a (samozrejme, v modulárnej aritmetike)? Napríklad, nahliadnúc 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 oboj 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.

Tajomné b^{-1} budeme nazývať *inverzný prvok* k prvku b . Vo všeobecnosti ich môže byť veľa (alebo žiadnen), avšak pohľad do tabuľky prezrádza, že ak sa pohybuje 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 odpuštíme. 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

Otázka zostáva, ako efektívne zistiť inverzný prvok k danému číslu. Jeden spôsob je len jednoducho dané b 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, n je číslo medzi 0 a $p - 1$. Potom

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

Ak sa pozrieme na uvedenú rovnicu a vykrátime obe strany číslom a , 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 umocniť hodnotu na $p - 2$ vieme v čase $O(\log P)$. 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, \dots . Samozrejme, priebežne treba číslo aj modulovať, aby sa nám zmestil 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 od zadu tohto zápisu (najmenej významnej cifry), vidíme, že b^1 prinásobiť máme a b^2 nie, pretože prísušná cifra 53 je nula, a takýmto spôsobom pokračovať až dokonca. 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 tieho 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 $O(N)$. 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

pripade menej ako 32, čo je celkom kamarátska konštanta. Keďže si musíme pamätať celý graf, výsledná pamäť je lineárna.

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