23. državno tekmovanje v znanju računalništva (1999)

NALOGE ZA PRVO SKUPINO

1999.1.1

R: 11

Podjetje Import Eskort te je najelo za svetovalca za rešitev njihovega problema letnice 2000. V svojih strojno berljivih bazah imajo zapisane datume transakcij tako, da je znakovni zapis za koledarsko leto dolg dva zloga (byta). Letnica 1987 je na primer predstavljena z znakoma `87`.

Predlagaj postopek, kako lahko veljavnost zapisa datuma podaljšaš vsaj do leta 2030, ali pa **utemelji**, zakaj je njihov problem žal nerešljiv. Pri tem ne smeš porabiti nič več prostora. Naj vas spomnimo, da gre v en zlog (byte) število med 0 in 255 (vključno s tema dvema).

1999.1.2

R: 11

Iz vesolja pričakuješ signal nezemeljske civilizacije. Utemeljeno lahko pričakuješ, da bo sporočilo poslano v jeziku, ki uporablja angleško abecedo 26 malih črk in zadošča naslednjim pravilom:

- Sporočilo je sestavljeno zgolj iz zaporedja znakov `a`, `b`, `c`, `d`, ..., `z`, ki sestavljajo besede, in iz presledkov med njimi.
- Zaporedne črke se vselej razlikujejo (ni podvojenih črk).
- Samoglasniki `a`, `e`, `i`, `o`, `u` se ne smejo stikati.
- Znak `x` je lahko le na koncu besede.

Napiši podprogram Pravo, ki ugotovi, ali sporočilo Sporocilo ustreza tem pogojem.

Klic funkcije naj bo takle:

```
function Pravo(var Sporocilo: array [1..1000000] of char): boolean;
```

1999.1.3

R: 12

V besedilu iščemo vrstice, ki vsebujejo vsaj eno zvezdico. Izpišemo vsako vrstico z zvezdico, poleg nje pa tudi njej sledečo vrstico, ne glede na to, ali tudi sama vsebuje zvezdico ali ne.

Napiši program, ki bo izvedel opisano opravilo. Predpostaviš lahko, da so vse vrstice krajše od 200 znakov.

R: 13 **1999.1.4** V državi Utopiji so ugotovili, da denar kvari ljudi, zato vsako leto izvedejo prerazporeditev bogastva. Vsako leto določijo, kolikšno je največje sprejemljivo premoženje. Za vsakega državljana popišejo, koliko premoženja ima; tistim, katerih premoženje presega največji dovoljeni znesek, vzamejo toliko, da mu ostane le še ta največji dovoljeni znesek. Letos bo največje sprejemljivo premoženje določeno kot 150 % povprečnega premoženja. Tako dobljeni denar razdelijo med najrevnejše državljane in to tako, da ima na koncu čim večje število najrevnejših državljanov enako in čim večje premoženje. Pri tem se vrstni red prebivalcev po bogastvu ne sme spremeniti: če je imel A prej vsaj toliko kot B, ima tudi po prerazporeditvi vsaj toliko kot B. **Opiši postopek**, ki bi to dosegel. Predpostavi, da je mogoče denar drobiti na poljubno majhne enote.

Primer: pet prebivalcev z začetnim premoženjem (30, 50, 120, 240, 260); vsota je 700; povprečje je 140; največje še sprejemljivo premoženje je 210; zadnjima dvema poberemo 30 oz. 50; dobimo 80; damo prvima dvema, končno stanje je (80, 80, 120, 210, 210). Nekatere nesprejemljive prerazporeditve so (90, 70, 120, 210, 210) (ker bi se spremenil vrstni red — prvi na seznamu ima zdaj več kot drugi), (40, 110, 120, 210, 210) (vrstni red se sicer ohrani, vendar je najrevnejši tukaj en sam, pri najboljši prerazporeditvi pa imata najrevnejša dva enako premoženje), (60, 60, 160, 210, 210) (najrevnejša dva imata sicer zdaj enako premoženje, vendar bi lahko dobila več).

NALOGE ZA DRUGO SKUPINO

R: 14 **1999.2.1** Dano je n-mestno število v desetiškem zapisu ($n \geq 2$). Na njegovih števkah lahko izvajaš tri osnovne računske operacije: seštevanje, odštevanje in deljenje. Pri deljenju se količnik zaokroži navzdol; pri seštevanju pa se upošteva le ostanek rezultata po deljenju z 10 (na primer: $6 + 7 = 3$). Deljenje z 0 seveda ni dovoljeno, prav tako pa ni dovoljeno odštevanje, če bi bila razlika negativna. Tako je rezultat vsake takšne operacije spet neko n-mestno število v desetiškem zapisu. Za operanda lahko vzameš dve različni števki ali pa dvakrat isto; rezultat nato zapišeš čez eno od števk, lahko tudi čez kakšno od tistih, ki si ju uporabil kot operanda.

Opiši postopek, ki ugotovi, če je mogoče dano n-mestno število z nekim zaporedjem teh operacij preoblikovati v dano drugo n-mestno število; za primere, ko je to mogoče, tudi opiši ustrezno zaporedje operacij (število teh operacij ni pomembno; nič hudega, če jih je veliko). Če problema ne znaš rešiti v splošnem, poskusi za kakšno podmnožico ciljnih števil (npr. taka, ki imajo po več enakih števk, ali pa taka, kjer je ena od števk enaka 1, ali pa soda, ali pa enaka neki potenci števila 2, ipd.). Splošnejše rešitve dobijo več točk; postopek naj bo čim bolj sistematičen.

Primer: število 4567 hočemo spremeniti v 1234. Primerno zaporedje ope-

racij bi bilo $7 - 4 = 3$ ($\rightarrow$ 4563), $6 - 3 = 3$ ($\rightarrow$ 4533), $5 - 4 = 1$ ($\rightarrow$ 1533), $5 - 3 = 2$ ($\rightarrow$ 1233), $1 + 3 = 4$ ($\rightarrow$ 1234). Drugo primerno zaporedje je tudi $4/4 = 1$ ($\rightarrow$ 1567), $1+1 = 2$ ($\rightarrow$ 1267), $1+2 = 3$ ($\rightarrow$ 1237), $2+2 = 4$ ($\rightarrow$ 1234). Še ena možnost je $6+6 = 2$ ($\rightarrow$ 4267), $2+2 = 4$ ($\rightarrow$ 4264), $6/4 = 1$ ($\rightarrow$ 1264), $4 - 1 = 3$ ($\rightarrow$ 1234).

1999.2.2

R: 15

V matriki $m \times n$ imajo elementi matrike lahko le vrednosti 0 ali 1. **Napiši funkcijo** Najvecji, ki vrne velikost stranice največjega kvadrata, ki povsem leži na poljih z enicami. Notranjost kvadrata mora biti zapolnjena z enicami.

Primer:

0	1	0	1	0	1
0	1	1	1	1	0
0	1	1	1	0	1
0	1	1	1	1	1
1	0	1	0	1	0
0	1	1	1	1	1
1	0	0	1	1	1

Pri zgornji matriki ima največji tak kvadrat stranico dolžine 3.

1999.2.3

R: 17

Matija je dobil službo pri projektu 2MS2MM (dva milijona Slovencev, dva milijona megabitov). Podjetje bo zagotavljalo hitro omrežno povezljivost za vse Slovence, zaradi prikaza porabe pa bo za vsakega Slovenca moralo voditi podatke o prometu. Podatek je shranjen v 32-bitnem števcu in Matija dobi vsakih pet minut tabelo z dvema milijonoma vrednosti, ki jih mora spraviti na disk.

Zahteve so naslednje:

(a) podatki morajo biti zapisani kar najhitreje,

(b) shraniti se mora zadnjih 1000 vrednosti vsakega števca,

(c) če med zapisovanjem zmanjka napajanja, se morajo zavreči vsi podatki iz danega vzorca,

(d) na zahtevo mora Matija vrniti zadnjih 1000 vrednosti zahtevanega števca, skupaj s časi, ko so bile zajete.

Napiši podprogram Shrani(Stevci: **var array** [1..2000000] **of** integer), ki vrednosti števcev shrani, in podprogram Izpisi(Stevec: integer), ki izpiše zadnjih 1000

vrednosti zahtevanega števca, skupaj s časom, ko je bila vsaka vrednost shranjena, v formatu:

$$\begin{array}{l} \mathit{čas}_1\ \mathit{vrednost}_1 \\ \mathit{čas}_2\ \mathit{vrednost}_2 \\ \ldots\ldots\ldots\ldots \end{array}$$

Podprogram Izpisi naj izpis začne pri najstarejši vrednosti in nadaljuje proti najnovejši. Če je shranjenih vrednosti manj kot 1000, naj funkcija izpiše vse shranjene vrednosti. Funkcija Cas: integer ti vrne trenutni čas. Funkcija Skoci(Datoteka; i: integer) postavi točko branja in pisanja v datoteki na zapis, določen z danim celim številom. Funkcija Izplakni(Datoteka) zagotovi, da so po vrnitvi iz podprograma vsi podatki, ki smo jih doslej zapisali v datoteko, tudi v resnici vpisani na disk (torej izprazni vmesne pomnilnike). Če podprogramu Write v nekem klicu podaš 512 ali manj bytov podatkov, lahko predpostaviš, da bodo ti podatki zapisani na disk atomarno (torej če bo sploh kaj od njih zapisanega na disk, bodo zapisani vsi v celoti; če pa bo pred tem prišlo do prekinitve, ne bo zapisano nič od njih).

Opiši tudi strukturo podatkov na disku.

R: 19

1999.2.4

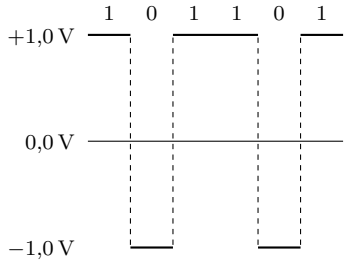
Kadar prek nekega zapisa digitalnih podatkov na magnetnem disku ali traku zapišemo novega, se prejšnja informacija le močno oslabi in jo pretežno prekrije nova, vendar lahko z natančnim branjem pridobimo dovolj informacije, da lahko rekonstruiramo poleg novega tudi prej prisotni zapis. Z dovolj natančnim odčitavanjem lahko rekonstruiramo tudi še starejši (že dvakrat prekriti) zapis ali včasih še več.

Posamezni zaporedni biti so zapisani kot namagnetenje majhnega področja diska v eno od dveh možnih smeri. Pri prehodu čitalne glave prek takega področja se v glavi pojavi električna napetost, ki je pozitivna (približno +1 V), če je na tem mestu zapisani bit 1, in negativna (približno −1 V), če je zapisani bit 0. Zaradi predhodnih zapisov in drugih fizikalnih vzrokov lahko odčitana napetost nekoliko odstopa od idealne vrednosti +1 V ali −1 V.

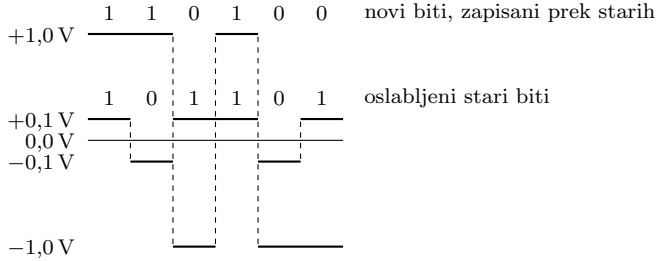
Predpostavimo, da pri pisanju novih podatkov predhodna vrednost na tem mestu oslabi na desetino in tej oslabljeni vrednosti se prišteje nova vrednost. Nova vrednost, ki jo bomo lahko na tem mestu odčitali, je torej $U_{novi} = U_{stari}/10 + U_{bit}$. Primer: če prek enice (+1,0 V) zapišemo ničlo (−1,0 V), bo odčitana napetost na tem mestu $+1{,}0\,\mathrm{V}/10 + (-1{,}0\,\mathrm{V}) = -0{,}9\,\mathrm{V}$. Na novem praznem disku je bilo namagnetenje diska zanemarljivo (blizu 0).

V domači delavnici smo s pomočjo digitalnega osciloskopa, priklopljenega na magnetno glavo, uspeli natančno izmeriti napetosti 256 zaporednih bitov. Te odčitke smo zapisali v datoteko, v vsako vrstico po eno realno število blizu +1 ali −1, ki ustreza enemu bitu. **Napiši program**, ki bo to datoteko prebral in rekonstruiral zadnji, predzadnji in predpredzadnji zapis ter izpisal vsa tri zaporedja bitov.

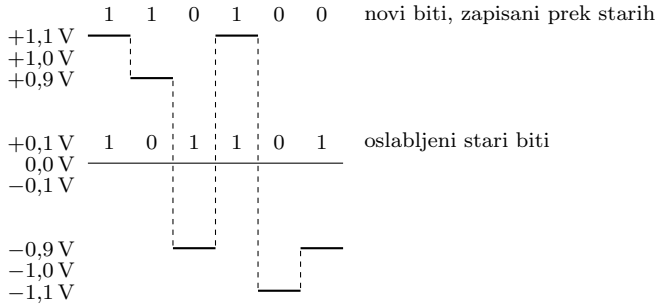
V ilustracijo: najstarejši zaporedni biti 101101 takoj zatem, ko so bili zapisani na prazen disk:



Čez prejšnji zapis zapišemo nove bite 110100. Stari zapis oslabi na desetino in k njemu se prišteje novi zapis. Za lažjo predstavo narišimo v isti diagram obe napetosti:



Rezultat je vsota obeh napetosti iz prejšnje slike:



NALOGE ZA TRETJO SKUPINO

R: 20

1999.3.1

Programer Primož se je znašel v težavnem položaju. Pred leti je napisal podprogram za iskanje v urejeni tabeli, ki ga mora zdaj prilagoditi spremenjenemu problemu.

Podprogram, iz katerega izhaja, PoisciElement (zapisan je spodaj), sprejme štiri parametre: iskani element (ki je kar celo število), tabelo, v kateri išče (tabela mora biti urejena in mora vsebovati vsaj en element) ter indeks prvega ter zadnjega elementa v tabeli. Med tema indeksoma podprogram z bisekcijo poišče element in njegov indeks vrne v parametru Indeks. Rezultat podprograma je v tem primeru true. Če iskanega elementa ni v tabeli, podprogram v parametru Indeks vrne indeks, v katerega mora biti element vstavljen, da bo tabela še vedno urejena. Rezultat je v tem primeru seveda false. Podprogram PoisciElement ima le eno napakico: če je v tabeli več elementov z enako vrednostjo, vrne indeks enega izmed njih, vnaprej pa ne znamo uganiti, ali bo vrnil indeks prvega takega elementa, zadnjega ali katerega od vmesnih. Če mu na primer pošljemo tabelo s petimi enakimi elementi in mejama iskanja 1 in 5, bo podprogram vrnil indeks 3.

Pomagaj Primožu **predelati podprogram** tako, da bo vedno vrnil indeks najbolj levega elementa (najmanjši indeks). V našem izrojenem primeru torej pričakujemo, da bo podprogram vrnil indeks 1. Če iskanega elementa ni v tabeli, se mora predelani podprogram obnašati enako kakor stari. Seveda si želi, da bo novi podprogram približno enako hiter kot stari.

Originalni podprogram, zapisan v pascalu.

```
const N = 1024;
type TabelaT = array [1..N] of integer;

function PoisciElement(Iskani: integer; var Tabela: TabelaT;
                       Prvi, Zadnji: integer; var Indeks: integer): boolean;
var Spodnji, Zgornji, Vmes, Element: integer;
begin
  Spodnji := Prvi; Zgornji := Zadnji;
  while Repeat
    Vmes := (Spodnji + Zgornji) div 2;
    Element := Tabela[Vmes];
    if Iskani = Element then begin
      PoisciElement := true; Indeks := Vmes; exit;
    end
    else if Iskani < Element
      then Zgornji := Vmes - 1
      else Spodnji := Vmes + 1;
  until Spodnji > Zgornji;
  if Iskani > Element
```

```
      then Indeks := Vmes + 1
      else Indeks := Vmes;
   PoisciElement := false;
end; {PoisciElement}
```

Originalni podprogram, zapisan v C++.

```
bool PoisciElement(int iskani, int tabela[], int prvi, int zadnji, int &indeks)
{
  int spodnji = prvi, zgornji = zadnji, vmes, element;
  while (spodnji <= zgornji)
  {
    vmes = (spodnji + zgornji) / 2;
    element = tabela[vmes];
    if (iskani == element) { indeks = vmes; return true; }
    else if (iskani < element) zgornji = vmes - 1;
    else spodnji = vmes + 1;
  }
  if (iskani > element) indeks = vmes + 1; else indeks = vmes;
  return false;
}
```

1999.3.2

R: 21

Véliki vezir ima težave s svojimi ministri. Neprestano sklepajo nove in nove medsebojne zveze, tako da ubogi vezir ne ve več, kje se ga drži glava. Zato se je odločil, da bo izpisal vse možne koalicije.[1] Ker pa v programiranju ni pretirano močan, te prosi za pomoč. Potrebuje enostaven program, ki bo prebral število ministrov in izpisal vse možne koalicije na standardni izhod. Da bo naloga bolj enostavna, naj se ministri imenujejo kar `A`, `B`, `C`, ... Če imamo na primer štiri ministre, naj program izpiše nekaj takega:

```
ABCD
ABC,D
ABD,C
AB,CD
AB,C,D
ACD,B
AC,BD
AC,B,D
AD,BC
A,BCD
A,BC,D
AD,B,C
A,BD,C
A,B,CD
A,B,C,D
```

[1] Tako pravi besedilo naloge iz leta 1999. V resnici naloga sprašuje po vseh možnih razdelitvah ministrov na eno ali več skupin; beseda „koalicija“ običajno pomeni le eno takšno skupino.

Namig: vsaj dve elegantni poti vodita do rešitve — z rekurzijo in z uporabo pomožnih datotek. In nikar se ne ubadajte preveč z obliko izpisa. Da se le vidi, kateri ministri držijo skupaj, pa bo čisto v redu.

R: 24

1999.3.3

Tekmovanje, ki je potekalo po sistemu „vsak proti vsakemu“, je bilo predčasno končano, zato so znani izidi le nekaterih tekem, ne pa vseh. Ker je bilo na koncu kljub vsemu potrebno določiti vrstni red tekmovalcev, so določili naslednje pravilo: tekmovalec A je vsaj tako dober kot tekmovalec B, če je premagal tekmovalca B v medsebojnem dvoboju ali pa je premagal kakšnega tekmovalca C, ki je vsaj tako dober kot tekmovalec B. (Če za tekmovalca A in B velja, da je tekmovalec A vsaj tako dober kot B in obratno, tedaj veljata za enako dobra.)

Sestavi algoritem, ki na podlagi le podmnožice odigranih tekem sestavi tak vrstni red tekmovalcev, v katerem je vsak tekmovalec uvrščen pred vsemi takimi tekmovalci, za katere velja, da je sam vsaj tako dober kot oni, oni pa niso vsaj tako dobri kot on. Bodi pozoren na to, da je lahko veljavnih vrstnih redov tekmovalcev več, algoritem pa naj poišče enega izmed njih.

Primer 1: Če sta na voljo rezultata:

tekmovalec 1 je premagal tekmovalca 2 in
tekmovalec 3 je premagal tekmovalca 4,

tedaj je veljaven katerikoli od naslednjih vrstnih redov:

tekmovalec 1, tekmovalec 2, tekmovalec 3, tekmovalec 4
tekmovalec 1, tekmovalec 3, tekmovalec 2, tekmovalec 4
tekmovalec 1, tekmovalec 3, tekmovalec 4, tekmovalec 2
tekmovalec 3, tekmovalec 4, tekmovalec 1, tekmovalec 2
tekmovalec 3, tekmovalec 1, tekmovalec 4, tekmovalec 2
tekmovalec 3, tekmovalec 1, tekmovalec 2, tekmovalec 4

Pomembno je torej le, da je tekmovalec 1 vedno pred tekmovalcem 2 in tekmovalec 3 vedno pred tekmovalcem 4.

Primer 2: Če so na voljo rezultati:

tekmovalec 1 je premagal tekmovalca 2,
tekmovalec 2 je premagal tekmovalca 3,
tekmovalec 3 je premagal tekmovalca 4,
tekmovalec 4 je premagal tekmovalca 2 in
tekmovalec 5 je premagal tekmovalca 6,

tedaj je veljaven katerikoli vrstni red, v katerem je tekmovalec 5 pred tekmovalcem 6 in tekmovalec 1 pred tekmovalci 2, 3 in 4. Tekmovalci 2, 3 in 4 veljajo za enako dobre. Primerjati jih je mogoče s tekmovalcem 1, ne pa tudi s tekmovalcema 5 in 6.

1999.3.4

R: 27

V podjetju *Pasivna orodja* se ukvarjajo s ponudbo brezplačne storitve elektronske pošte. Ker je ponudba zelo ugodna, imajo zelo veliko prometa. Zaradi tega so prisiljeni deliti računalniške kapacitete na več računalnikov. Na hitri lokalni mreži imajo N enakih računalnikov, ki vsi zaganjajo enak program. Eden izmed njih je glavni računalnik, vsi ostali pa so podporni. Naloga glavnega računalnika, ki edini dobiva zahteve z Interneta, je razdelitev le-teh na podporne računalnike. V primeru, da glavni računalnik odpove, ostanejo podporni računalniki brez dela. Sistemski inženir Matjaž trenutno rešuje ta problem, a ne najde rešitve. Pomagaj mu!

Napiši algoritem, ki zazna izpad glavnega računalnika in sočasno izbere novega. Na mreži isti trenutek ne sme obstajati več kot en glavni računalnik, čas brez glavnega računalnika pa želimo kar se da skrajšati. Vsak računalnik ima svojo enolično številko (npr. številka omrežne kartice).

Uporabi spodaj navedeno okostje programa. Kodo programa vpiši v podprogram, ki je klican enkrat na sekundo. Vsa sporočila, poslana v klicu tega podprograma, zagotovo pridejo do cilja do naslednjega klica podprograma in se pri tem ne izgubljajo.

```
var
  { globalne spremenljivke }

{ zunanje procedure }

{ vrne številko računalnika (je večja od 0), na katerem teče naš program }
function VrniStRacunalnika: integer; external;

{ nastavi stanje računalnika }
procedure PostaviGlavniRac; external;
procedure PostaviPodporniRac; external;

{ pošlje številko Num vsem računalnikom v omrežju (tudi nazaj pošiljatelju) }
procedure PosljiVsem(Num: integer); external;

{ funkcija za sprejemanje podatkov iz mreže }
function Prejmi(var Num: integer): boolean; external;
{ Števila, ki so prišla po mreži do našega računalnika, se zbirajo v vrsti v nekem
  pomožnem pomnilniku. Ta funkcija vzame iz vrste tisto število, ki je najdlje v
  njej, in ga shrani v Num ter vrne true; če pa je bila vrsta prazna, vrne false,
  vrednosti Num pa ne spreminja. }

procedure Zacni;
begin
  { inicializacija, klicana pred prvim klicem podprograma VsakoSekundo }
end; {Zacni}

procedure VsakoSekundo;
begin
  { tu vpišite svojo kodo }
end; {VsakoSekundo}
```

LETO 1999, TEKMOVANJE V POZNAVANJU UNIXA

Pravila

Pri vseh nalogah lahko uporabiš ukaze ukaznih lupin (`csh`, `sh`, `bash`, `ksh`...), skriptnih jezikov (`sed`, `awk`, `perl`...) ali običajnih programov, ki sestavljajo sistem UNIX, priporočeno po standardu POSIX.1. Višjih jezikov (C, pascal, fortran...) ni dovoljeno uporabiti.

Če si v dvomu, ali si uporabil dovoljena sredstva, lahko kadarkoli povprašaš nadzorno komisijo. Odločitev nadzorne komisije je dokončna.

1999.U.1 Besedna analiza

R: 29 Naredi preprosto frekvenčno analizo besedila. Preberi datoteko in na standardni izhod izpiši seznam vseh besed in njihovih frekvenc. Seznam naj bo urejen po vrsti od najmanj frekventnih besed do najbolj frekventnih. Frekvenca pomeni, kolikokrat v datoteki se beseda pojavi. V datoteki ni drugih znakov razen presledkov in črk.

1999.U.2 `vi`

R: 32 Na sistemu z veliko uporabniki se želiš izogniti temu, da bi isto datoteko z urejevalnikom `vi` odprl več kot en uporabnik hkrati. Predlagaj rešitev! Rešitev zapiši v obliki potrebnih ukazov. Komentiraj, kaj so po tvojem mnenju prednosti in slabosti tvojega predloga. Predpostaviti smeš, da vsi uporabniki kličejo urejevalnik takole: `vi datoteka`. Urejevalnik `vi` sam po sebi ne opozori, če je neko datoteko že odprl kdorkoli drug.

1999.U.3 Premešaj

R: 35 V neki datoteki so vrstice urejene po določenem kriteriju. Ta urejenost te moti, zato želiš vrstice psevdonaključno razmešati. Napiši kodo, ki bo to storila. Bodi pozoren na učinkovitost in elegantnost svojega predloga. Psevdonaključno pomeni, da smeš uporabiti generator naključnih števil, ki ti je v tvojem orodju na voljo.

1999.U.4 Številke IP

R: 36 V tekstovni datoteki so na več mestih zapisani številski naslovi IP, ki jih želiš spremeniti v polnovredno ime računalnika (FQDN, angl. fully qualified domain name).

V bazi `/etc/hosts` so po vrsticah navedeni številski naslov in njegovo polnovredno ime:

```
193.2.1.72 nanos.arnes.si
```

V datoteki razen takih zapisov ni ničesar drugega.

Številski naslov IP je lahko oblike: 0.0.0.0–255.255.255.255. Brez škode za splošnost lahko predpostaviš, da v tvoji datoteki vsak zapis oblike 0.0.0.0–999.999.999.999 predstavlja številski naslov IP in da imajo vsi v datoteki zapisani številski naslovi pripadajoča polnovredna imena v bazi. Upoštevaj še, da se naslovi IP ne stikajo z drugimi znaki razen s presledki.

REŠITVE NALOG ZA PRVO SKUPINO

R1999.1.1

N: 1

Da ob prelomu stoletja ne bo zmede, je vsekakor koristno, če lahko letnico predstavimo v celoti, ne pa le njenih zadnjih dveh števk. Vsak byte je dolg 8 bitov in lahko hrani eno od $2^8 = 256$ različnih vrednosti; z dvema bytoma lahko torej predstavimo $256 \times 256 = 65536$ različnih vrednosti, tako da bi lahko, če bi letnice hranili kot 16-bitna dvojiška števila, predstavili datume še za nadaljnjih več deset tisočletij.

Še ena možnost, ki bi tudi delovala kar precej časa, je, da za vsako števko uporabimo 4 bite (saj omogočajo štirje biti 16 različnih vrednosti, mi pa jih potrebujemo le 10) in tako v vsak byte shranimo dve števki. Dva byta tako zadostujeta za štiri števke, torej za vsa leta do vključno 9999.

Če pa hočemo na vsak način ostati pri predstavitvi, ki uporablja le ASCII znake za števke od 0 do 9, bi lahko letnice predstavljali enako kot doslej (torej z dvema števkama, ki povesta desetice in enice), le da bi uvedli dogovor, po katerem števila od `00` do `30` predstavljajo letnice od 2000 do 2030, števila od `31` do `99` pa letnice od 1931 do 1999. Slabost te rešitve je, da ne moremo predstaviti letnic pred 1931 ali po 2030.

R1999.1.2

N: 1

Spodnji podprogram si v spremenljivki `c` zapomni trenutni znak in v `s` še podatek o tem, ali je to samoglasnik; v `pc` in `ps` hrani ta dva podatka za prejšnji znak. Tako ni težko preverjati raznih pogojev. Pogoj, da sme biti `x` le na koncu besede, lahko preverimo pri naslednjem znaku; če je `x` čisto zadnji znak v nizu, je s tem tudi na koncu besede in ga ni treba preverjati še posebej.

```
function Pravo(Sporocilo: var array [1..1000000] of char): boolean;
var i: integer; c, pc: char; s, ps: boolean;
begin
  ps := false; pc := '.'; Pravo := false;
  for i := 1 to 1000000 do begin
    c := Sporocilo[i]; s := c in ['a', 'e', 'i', 'o', 'u'];
    if not (c in ['a'..'z', ' ']) then exit; { nedovoljen znak }
    if c = pc then exit; { podvojen znak }
    if s and ps then exit; { dva samoglasnika zaporedoma }
```

```
      if (pc = 'x') and (c <> ' ') then exit; { x ni na koncu besede }
      pc := c; ps := s;
    end; {for}
    Pravo := true;
end; {Pravo}
```

N: 1

R1999.1.3

Spodnji program bere vrstico za vrstico, pri vsaki pa preveri, če slučajno vsebuje kakšno zvezdico. To si zapomni v spremenljivki Najden, v spremenljivki IzpisiNaslednjo pa si zapomni, če je našel zvezdico v prejšnji in mora zato izpisati tudi trenutno vrstico. Nato vrstico, če je to potrebno, izpiše, vrednost Najden pa prenese v IzpisiNaslednjo, da jo bo imel pri roki ob delu z naslednjo vrstico.

```
program Zvezdice(Input, Output);
const
  VrsticaM = 200;
var
  Vrstica: packed array [1..VrsticaM] of char;
  j: integer;
  Najdena: boolean;           { vrstica vsebuje zvezdico }
  IzpisiNaslednjo: boolean;   { izpisati bo treba tudi naslednjo vrstico }
begin
  IzpisiNaslednjo := false;
  while not Eof(Input) do begin
    ReadLn(Vrstica);
    j := 1; Najdena := false;
    while not Najdena and (j <= VrsticaM) do
      if Vrstica[j] = '*' then Najdena := true else j := j + 1;
    if Najdena or IzpisiNaslednjo then WriteLn(Vrstica);
    IzpisiNaslednjo := Najdena;
  end; {while}
end. {Zvezdice}
```

Kot še mnoge druge naloge, ki se tičejo predvsem dela z nizi ali besedili, lahko tudi to nalogo rešimo veliko krajše, če uporabimo kakšen drug programski jezik, na primer perl:

```
perl -ne '$z = /\*/; print if $z || $pz; $pz = $z'
```

S stikalom -e smo interpreterju perla naročili, da je program naveden kar kot naslednji parameter v ukazni vrstici (zato je v narekovajih), s stikalom -n pa smo mu naročili, naj dani program ponavlja v zanki, po enkrat za vsako vrstico vhodne datoteke. Obe stikali lahko združimo v -ne. Program deluje tako kot zgornja pascalska rešitev: v spremenljivko $z si zapiše, če trenutna vrstica vsebuje zvezdico (to preveri z regularnim izrazom * — poševnica pred zvezdico je potrebna zato, ker ima zvezdica v regularnih izrazih drugače poseben pomen

— dovoli nič ali več ponovitev izraza, ki stoji pred zvezdico); v spremenljivki `$pz` hrani podatek o tem, ali je bila zvezdica v prejšnji vrstici; če je res kaj od tega dvojega, trenutno vrstico izpiše; nato pa vrednost `$z` prenese v `$pz`, da jo bo uporabil pri naslednji vrstici.

R1999.1.4

N: 2

Ves pobrani denar dajmo na kup; najrevnejšemu dajmo toliko, da bo imel potem enako kot drugi najrevnejši. Če to ne gre (torej če denarja na kupu ni dovolj), mu dajmo vse, kar je še ostalo, in končajmo. Nato dajmo prvima dvema najrevnejšima (ki imata zdaj enako) toliko, da bosta imela enako kot tretji najrevnejši. Če to ne gre, dajmo vsakemu pol od tega, kar nam je še ostalo, in končajmo. Nato dajmo trem najrevnejšim toliko, da bodo imeli enako kot četrti; če bi zmanjkalo kupa, dajmo vsakemu tretjino preostanka in končajmo. Tako nadaljujemo, dokler nam kupa ne zmanjka.

Seveda denarja v resnici ni treba deliti takole po kapljicah; lahko tudi z enim pregledom tabele premoženja prebivalcev določimo, koliko bodo imeli po prerazporeditvi najrevnejši ljudje, in nato vsakemu od njih damo toliko denarja, kolikor mu še manjka do te končne vrednosti.

```
const StPreb = ...;
type Tabela = array [1..StPreb] of real;

{ Tabela mora biti urejena naraščajoče.
  Ta podprogram predpostavlja, da smo bogatašem že pobrali toliko,
  za kolikor presegajo največje dovoljeno premoženje, skupna pobrana
  vsota denarja pa je v parametru Kup. }
procedure Razdeli(var Premozenje: Tabela; Kup: real);
var StPrej: integer; { število prejemnikov }
    Vsota: real; { vsota prvotnih premoženj vseh prejemnikov 1..StPrej }
    Novo: real; { novo premoženje najrevnejših po prerazporeditvi }
    i: integer; Zmanjka: boolean;
begin
  StPrej := 0; Vsota := 0; Zmanjka := false;
  while (StPrej < StPreb) and not Zmanjka do
    { Invarianta: kup je dovolj velik, da damo lahko najrevnejšim StPrej − 1
      ljudem toliko, da bodo imeli enako kot oseba StPrej. }
    if Vsota + Kup <= Premozenje[StPrej + 1] * StPrej then Zmanjka := true;
    else begin StPrej := StPrej + 1; Vsota := Vsota + Premozenje[StPrej] end;
  if StPrej > 0 then begin
    Novo := (Vsota + Kup) / StPrej;
    for i := 1 to StPrej do Premozenje[i] := Novo;
  end; {if}
end; {Razdeli}
```

REŠITVE NALOG ZA DRUGO SKUPINO

N: 2

R1999.2.1

Če so vse števke vhodnega števila enake 0, ne moremo iz njega dobiti nobenega drugega števila. Če pa je kakšna števka različna od 0, jo delimo samo s sabo, da dobimo 1; nato jo odštevajmo od vseh drugih, dokler ne dobimo $00\ldots010\ldots0$; prištejmo jo na prvem mestu in odštejmo na starem, da dobimo $1000\ldots00$. Nato jo prištevajmo na vseh mestih razen na prvih dveh, dokler ne dobimo tam želenih končnih vrednosti; dogovorimo se, da bomo tiste števke odslej pustili pri miru in ostal nam je le še problem, ali lahko iz para $(1,0)$ dobimo vse druge pare (a,b). Ker delamo v desetiškem zapisu in je takih parov le sto, se lahko o tem, da lahko res pridemo do vseh, prepričamo tudi ročno. Lahko pa se o tem prepričamo tudi z naslednjim induktivnim razmislekom, ki bi veljal tudi, če bi delali v kakšnem drugem številskem sestavu, ne le v desetiškem.

Vidimo, da lahko iz $(1,0)$ s prištevanjem in odštevanjem dobimo $(1,1)$, $(0,1)$, $(0,0)$. Recimo, da že znamo priti do vseh parov (a,b), kjer sta a in b manjša ali enaka nekemu k (na začetku naj bo $k=1$). Iz $(1,k)$ s prištevanjem pridemo do $(1,k+1)$ in z odštevanjem do $(0,k+1)$; podobno iz $(k,1)$ do $(k+1,1)$ in $(k+1,0)$; iz slednjega s prištevanjem do $(k+1,k+1)$. Za števila oblike $(a,k+1)$, kjer je a med vključno 1 in k: opazimo, da je potem tudi $k+1-a$ med vključno 1 in k, zato do $(a,k+1-a)$ po predpostavki že znamo priti; iz njega pa s prištevanjem tudi do $(a,k+1)$. Podobno pokažemo za $(k+1,a)$. Tako torej vidimo, da lahko pridemo do vsakega ciljnega števila.

Zanimivo bi bilo razmisliti tudi o primeru, ko imamo eno samo števko (torej $n=1$). Če ta ni ničla (iz katere ne moremo dobiti ničesar drugega), lahko z odštevanjem vedno dobimo 0, z deljenjem vedno 1, seštevanje pa je zdaj v bistvu podvajanje. Zato iz 1 dobimo 2, 4, 8, 6 (ker je $16 \bmod 10 = 6$; iz slednje pa pridemo spet v 2, tako da od tu naprej ne dobimo nič novega). Iz a bi dobili $2a$, ki pa je sod, tako da od tu naprej tudi ne dobimo nič novega. Vidimo torej, da lahko iz neničelnega a dobimo števila $a, 0, 1, 2, 4, 6, 8$, ostalih pa nikakor. Ni pa vedno tako hudo, saj lahko na primer v enajstiškem sistemu dobimo vse neničelne števke: $1 \to 2 \to 4 \to 8 \to 16 \equiv 5 \to 10 \to 20 \equiv 9 \to 18 \equiv 7 \to 14 \equiv 3 \to 6 \to 12 \equiv 1$.

Recimo, da delamo v b-iškem sestavu. Pri katerih b se bo dalo iz 1 s podvajanjem priti do vseh ostalih neničelnih števk? Za začetek opazimo, da b ne sme biti sod (razen $b=2$, ko je stvar trivialna): katerokoli števko podvojimo, je rezultat sod, če pa je večji ali enak b, se od pravega rezultata (ki je sod) odšteje b (ki je tudi sod) in dobimo spet sodo vrednost. Zato pri sodem b s podvajanjem ne moremo priti do nobene lihe števke. Omejimo se zdaj na lihe b. Zahteva, da lahko iz 1 s podvajanjem dobimo vse ostale neničelne števke, je pravzaprav enakovredna zahtevi, naj imajo števila $1, 2, 4, 8, \ldots, 2^{b-2}$

same različne ostanke po modulu b (ker jih je $b-1$, bodo to ravno vse števke $1, 2, \ldots, b-2, b-1$; ostanek 0 ni mogoč, saj potence števila 2 ne morejo biti večkratniki b-ja, ko pa je ta vendar lih). Lahko si pomagamo z Eulerjevim izrekom iz teorije števil; ta pravi, da če sta si a in b tuja, je $a^{\phi(b)} \bmod b = 1$. Pri tem je $\phi(b)$ število b-ju tujih števil iz množice $\{1, \ldots, b-1\}$. Ker je b lih, mu je $a = 2$ tuj in lahko uporabimo ta izrek; $\phi(b)$ že po definiciji ne more biti večji od $b-1$, če pa bi bil manjši, bi to pomenilo, da se nam pri podvajanju števk ponovi števka 1 že po $\phi(b) < b-1$ korakih, ne pa šele po $b-1$ korakih. Torej mora biti $\phi(b) = b-1$; toda to pomeni, da so vsa števila od 1 do $b-1$ tuja b-ju; z drugimi besedami, b mora biti praštevilo. Izkaže se, da je ta pogoj potreben, ne pa še tudi zadosten. Primerna so le praštevila oblike $8m+3$ in $8m+5$, pa še to ne vsa. Med števili do 100 so primerna samo naslednja: 2, 3, 5, 11, 13, 19, 29, 37, 53, 59, 61, 67, 83. Emil Artin je postavil znano domnevo, ki med drugim pravi, da je primernih b-jev neskončno mnogo, vendar to še ni čisto dokazano.[2]

R1999.2.2

Najenostavneje bi lahko poiskali največji kvadrat enic tako, da bi se postavili v vsako polje matrike in se vprašali, kolikšen je največji kvadrat z (recimo) spodnjim desnim kotom v tem polju. N: 3

```
const m = ...; n = ...;
type Matrika = array [1..m, 1..n] of integer;

function Najvecji(var a: Matrika): integer;
var i, j, k, s, sm: integer; ok: boolean;
begin
  sm := 0; { sm je stranica največjega doslej najdenega kvadrata }
  for i := 1 to m do for j := 1 to n do begin
    { Poiskali bomo največji kvadrat enic s spodnjim desnim
      kotom na polju a[i, j]. Njegova stranica bo s. }
    s := 0; ok := true;
    while (s < i) and (s < j) and ok do begin
      { Vemo, da obstaja primeren kvadrat s stranico s.
        Ali obstaja tudi tak s stranico s + 1? }
      k := 1; s := s + 1;
      while (k <= s) and ok do begin
        if a[i - s + 1, j - k + 1] = 0 then ok := false;
        if a[i - k + 1, j - s + 1] = 0 then ok := false;
        k := k + 1;
      end; {while k}
    end; {while s}
    { Če se je zanka ustavila, ker smo pri poskusu povečevanja kvadrata odkrili ničlo,
```

[2] Glej MathWorld *s. vv.* "Primitive Root", "Euler's Totient Theorem", "Artin's Conjecture"; *The On-Line Encyclopedia of Integer Sequences*, A001122; in nit z naslovom "2 is a primitive root" v *sci.math* 26.–28. januarja 2001.

```
      je bilo zadnje povečanje s-ja neupravičeno. }
    if not ok then s := s − 1;
    { Če je to nov največji kvadrat, si ga zapomnimo. }
    if s > sm then sm := s;
  end; {for i, j}
  Najvecji := sm;
end; {Najvecji};
```

Vendar pa je ta rešitev precej neučinkovita. Če je matrika polna samih enic, bomo pri vsakem polju (i, j) odkrili kvadrat s stranico $\min\{i, j\}$ in pri tem pregledali vsa polja tega kvadrata. Časovna zahtevnost take rešitve je zato $O(mn(\min\{m, n\})^2)$.

Program si lahko prihrani veliko dela, če si sproti pomaga z rezultati tega, kar je doslej že naredil. Označimo s $s(i, j)$ stranico največjega kvadrata enic z desnim spodnjim kotom v polju (i, j). Ko razmišljamo o kvadratih z desnim spodnjim kotom v polju (i, j), že poznamo $s(i-1, j)$, $s(i, j-1)$ in $s(i-1, j-1)$. No, če je v polju $a(i, j)$ naše vhodne matrike ničla, že tako ali tako vemo, da bo $s(i, j) = 0$. Pa recimo zdaj, da je $a(i, j) = 1$ in je potemtakem $s(i, j) = S \geq 1$. Znotraj tega kvadrata samih enic (s stranico S in desnim spodnjim kotom v polju (i, j)) ležijo tudi kvadrati samih enic s stranico $S-1$ in desnimi spodnjimi koti v poljih $(i-1, j)$, $(i-1, j-1)$ in $(i, j-1)$; torej imamo pogoje: $s(i-1, j) \geq S-1$, $s(i-1, j-1) \geq S-1$ in $s(i, j-1) \geq S-1$. Po drugi strani, čim nek S ustreza tem trem pogojem (poleg tega pa je še $a(i, j) = 1$), takoj sledi, da so v kvadratu s stranico S in desnim spodnjim kotom v polju (i, j) same enice. Tako vidimo, da pri $a(i, j) = 1$ velja $s(i, j) = 1 + \max\{s(i-1, j)s(i-1, j-1), s(i, j-1)\}$. Lepo pri opisanem razmisleku je tudi to, da nam tabele s ne bo treba poznati v celoti, pač pa moramo, ko razmišljamo o $s(i, j)$, poznati le $s(i, j')$ za $j' < j$ ter $s(i-1, j')$ za $j' \geq j-1$, ostalo pa lahko sproti pozabljamo.

```
function Najvecji(a: Matrika): integer;
var s: array [0..n] of integer; i, j, sm, sp, sn: integer;
begin
  { Tabela s se na začetku nanaša na i = 0, kar leži
    zunaj tabele in si zato mislimo tam same ničle. }
  sm := 0; for j := 0 to n do s[j] := 0;
  for i := 1 to m do begin
    sp := 0;
    for j := 1 to n do begin
      { V polju s[j − 1] je zdaj vrednost s(i, j − 1);
        v polju s[j] je vrednost s(i − 1, j);
        vrednost s(i − 1, j − 1) pa imamo začasno v sp.
        Izračunajmo s(i, j) in jo začasno shranimo v sn. }
      if a[i, j] = 0 then sn := 0
      else sn := 1 + Min(s[j], s[j − 1], sp);
```

```
        if sn > sm then sm := sn; { Nov največji kvadrat. }
        { V naslednji iteraciji bomo potrebovali vrednost s(i − 1, j), ki je trenutno
          še v s[j]; zapomnimo si jo v sp, da bomo lahko zdaj v s[j] vpisali
          pravkar izračunano vrednost s(i, j), ki jo trenutno hranimo v sn. }
        sp := s[j]; s[j] := sn;
      end; {for j}
    end; {for i}
    Najvecji := sn;
end; {Najvecji}
```

Zdaj imamo z vsakim poljem (i, j) le konstantno mnogo dela, tako da je časovna zahtevnost celotnega postopka le $O(mn)$. Asimptotično pravzaprav bolje sploh ne bi moglo biti, saj potrebujemo toliko časa že samo za branje matrike a.

Mimogrede, zanimiva različica te naloge bi bila taka, pri kateri bi iskali največji pravokotnik samih enic (največji v tem smislu, da ima največjo ploščino). Glej npr. 4. nalogo na 1. izbirnem tekmovanju univerze v Portu za ACM SWERC, 2. maja 2001 (#836 na `online-judge.uva.es`); in David Vandevoorde, *The maximal rectangle subproblem*, Dr. Dobb's Journal, april 1998.

R1999.2.3

N: 3

Zaradi zahteve, naj se podatki zapišejo kar najhitreje, smo se odločili, da zapišemo vseh dva milijona števcev na disk kar v enem bloku, na začetku bloka pa naj bo še čas, ko so bili ti števci zajeti. Po tisoč pisanjih si bo tako v datoteki sledilo tisoč takih blokov; ob naslednjem (tisočprvem) pisanju bomo morali nekako doseči, da bo računalnik na prvi blok pozabil, saj smo rekli, da hočemo hraniti le zadnjih tisoč vrednosti vsakega števca. Še najlažje je, če takrat z novimi vrednostmi kar povozimo najstarejši blok, torej ravno tiste stare vrednosti, ki bi jih tako ali tako radi zavrgli. Pri izpisovanju vrednosti števcev bomo morali vedeti, kje se začnejo in kje končajo, zato bomo v ta namen na začetku datoteke, pred vsemi podatki o števcih, hranili še dve celi števili, ki povesta, kateri blok je najstarejši in koliko je vseh skupaj.

Zahtevo, naj se ob prekinitvi napajanja med pisanjem zadnja meritev zavrže, lahko upoštevamo tako, da v datoteki v resnici pustimo prostora za 1001 blok, tako da je med najnovejšim in najstarejšim dovolj prostora še za en blok. Najprej zapišimo nove vrednosti v ta dodatni blok, nato pa, če je bilo blokov prej manj kot tisoč, povečajmo podatek o številu blokov (na začetku datoteke); če pa jih je bilo že prej tisoč, povečajmo podatek o indeksu najstarejšega bloka (to nam zagotovi, da bomo tistega najstarejšega zdaj pozabili oz. ga bomo ob naslednjem pisanju povozili z novimi vrednostmi). Tako, če pride do prekinitve napajanja med pisanjem novih vrednosti (ali pa po pisanju teh in pred popravljanjem podatkov na začetku datoteke), nismo ničesar izgubili, saj smo pisali v neuporabljeni pomožni blok. Po pisanju v pomožni blok in

pred popravljanjem glave datoteke kličimo za vsak primer še Izplakni,[3] da ne bi slučajno operacijskemu sistemu kasneje prišlo na misel najprej zapisati na disk naše popravke glave datoteke, nato pa bi zmanjkalo elektrike, še preden bi se na disk zapisali tudi podatki v pomožnem bloku.

```
const N = 2000000; M = 1000;
type StevciT = array [1..N] of integer;

procedure Shrani(var Stevci: StevciT);
var F: file of integer;
    Prvi, Koliko, Kam, i: integer;
begin
  Assign(F, 'shramba'); Reset(F);
  Read(F, Prvi, Koliko);
  Kam := (Prvi + Koliko) mod (M + 1);
  Skoci(F, 2 + Kam * (N + 1));
  Write(F, Cas);
  for i := 1 to N do Write(F, Stevci[i]);
  Izplakni(F);
  if Koliko >= M then begin
    Skoci(F, 0); Write(F, (Prvi + 1) mod (M + 1));
  end else begin
    Skoci(F, 1); Write(F, Koliko + 1);
  end; {if}
  Izplakni(F); Close(F);
end; {Shrani}

procedure Izpisi(Stevec: integer);
var F: file of integer;
    OdKod, Koliko, T, X: integer;
begin
```

[3] Mnogi prevajalniki pascala ponujajo podprogram Flush, ki na prvi pogled daje vtis, kot da dela nekako to, kar bi mi tule želeli od funkcije Izplakni. Vendar pa je Flush običajno zamišljen le za datoteke tipa text, pri katerih standardna knjižnica (enota System) praviloma vzdržuje nek majhen medpomnilnik za odloženo pisanje podatkov. Flush naj bi podatke, ki so še v tem medpomnilniku in čakajo, da bodo zapisani, predal operacijskemu sistemu; vendar pa slednji praviloma tudi sam skrbi za odloženo zapisovanje podatkov na disk, tako da ni nujno, da so naši podatki po klicu Flush res že zapisani na disk.

Implementacija podprograma Izplakni bo v splošnem odvisna od prevajalnika in operacijskega sistema. Spodnja različica je primerna, če je naš prevajalnik FreePascal, operacijski sistem pa kakšen iz družine Windows:

```
uses Dos, Windows; { FileRec je iz Dos, FlushFileBuffers pa iz Windows. }
procedure Izplakni(var f: file);
  begin FlushFileBuffers(FileRec(f).Handle) end;
```

Spremenljivke tipa **file** so tu namreč v resnici strukture tipa FileRec. Polje Handle je številka odprte datoteke, ki jo moramo uporabljati pri klicih funkcij operacijskega sistema. FlushFileBuffers je standardna funkcija iz vmesnika (APIja) Win32.

```
  Assign(F, 'shramba'); Reset(F);
  Read(F, OdKod, Koliko);
  while Koliko > 0 do begin
    Skoci(F, 2 + OdKod * (N + 1));
    Read(F, T);
    Skoci(F, 2 + OdKod * (N + 1) + Stevec);
    Read(F, X);
    WriteLn(T, ' ', X);
    Koliko := Koliko - 1; OdKod := (OdKod + 1) mod (M + 1);
  end; {while}
  Close(F);
end; {Izpisi}

procedure Inicializacija;
var F: file of integer;
begin
  Assign(F, 'shramba'); Rewrite(F);
  Write(F, 0, 0); Close(F);
end; {Inicializacija}
```

R1999.2.4

N: 4

Ker so stare vrednosti desetkratno oslabljene, ne morejo vplivati na predznak meritve; ta nam torej pove, katera je bila zadnja shranjena vrednost, nato pa lahko to vrednost odštejemo in preostanek pomnožimo z deset, pa smo (približno, če zanemarimo morebitne majhne motnje na traku) rekonstruirali stanje pred zadnjim pisanjem na ta del traku.

```
program Smetarjenje(Input, Output);
var Meritev: real;
    j, Starost: integer;
begin
  for j := 1 to 256 do begin
    ReadLn(Meritev);
    Write('Meritev: ', Meritev:6:3, ' ');
    for Starost := 0 to 2 do begin { postopek lahko večkrat ponovimo }
      { ugotovimo nazadnje zapisani bit — ta prevladuje nad ostalimi }
      if Meritev > 0
        then begin Write('1'); Meritev := Meritev - 1 end
        else begin Write('0'); Meritev := Meritev + 1 end;
      { odšteli smo že najbolj svežo vrednost, ojačajmo preostanek }
      Meritev := 10 * Meritev;
      { V praksi se po nekaj korakih približamo pragu motenj in šuma,
        zato starejših podatkov ne moremo več zanesljivo rekonstruirati. }
    end; {for Starost}
    WriteLn;
  end; {for j}
end. {Smetarjenje}
```

REŠITVE NALOG ZA TRETJO SKUPINO

N: 6

R1999.3.1

Predelani podprogram se od prvotnega v bistvu razlikuje le v majhni podrobnosti: namesto da bi iskali podani element, se pretvarjamo, kot da iščemo le malo manjšo vrednost. Če torej najdemo vrednost, ki je manjša od iskane, se odločimo, da je premajhna in iščemo dalje (tako kot v originalnem podprogramu). Če pa najdemo vrednost, ki je večja ali enaka, se naredimo, da je prevelika in prav tako iščemo dalje. Podprogram se zato vedno ustavi takrat, ko spodnja meja iskanja postane večja od zgornje. V težavo zaidemo le, ker mora podprogram vrniti true ali false v odvisnosti od tega, ali je iskana vrednost prisotna v tabeli. Zato uvedemo novo logično spremenljivko Nasel, ki jo postavimo na true, kadar zaznamo element z iskano vrednostjo.

Rešitev v pascalu:

```
function PoisciNajmanjsiElement(Iskani: integer; var Tabela: TTabela;
                                Prvi, Zadnji: integer; var Indeks: integer): boolean;
var
  Spodnji, Zgornji, Vmes: integer;
  Element: integer;
  Nasel: boolean;
begin
  Nasel := false;
  Spodnji := Prvi;
  Zgornji := Zadnji;
  repeat
    Vmes := (Spodnji + Zgornji) div 2;
    Element := Tabela[Vmes];
    if Iskani = Element then Nasel := true;
    if Iskani <= Element
      then Zgornji := Vmes - 1
      else Spodnji := Vmes + 1;
  until Spodnji > Zgornji;
  if Iskani > Element
    then Indeks := Vmes + 1
    else Indeks := Vmes;
  PoisciNajmanjsiElement := Nasel;
end; {PoisciNajmanjsiElement}
```

Rešitev v C++:

```
bool PoisciNajmanjsiElement(int iskani, int tabela[], int prvi, int zadnji, int &indeks)
{
  int spodnji = prvi, zgornji = zadnji, vmes, element;
```

```
  bool nasel = false;
  while (spodnji <= zgornji)
  {
    vmes = (spodnji + zgornji) / 2;
    element = tabela[vmes];
    if (iskani == element) nasel = true;
    if (iskani <= element) zgornji = vmes - 1; else spodnji = vmes + 1;
  }
  if (iskani > element) indeks = vmes + 1; else indeks = vmes;
  return nasel;
}
```

R1999.3.2

N: 7

Do rešitve nas pripelje indukcija: „Kaj se zgodi, če dodamo še enega ministra?“ Z enim ministrom je problem enostavno rešljiv — tvori lahko le eno koalicijo, sam s sabo. Denimo sedaj, da smo rešili problem za N ministrov, in dodajmo še enega. Rešitve problema N ministrov si ogledamo eno za drugo. Pri vsaki je N ministrov razdeljenih v nekaj koalicij (od 1 do N). Novi minister se lahko pridruži katerikoli od koalicij, lahko pa ostane sam. Če je na primer N enak 3 in si ogledujemo razdelitev `A BC`, lahko iz nje nastanejo tri nove razdelitve: `AD BC`, `A BCD` in `A BC D`. To razmišljanje nas neposredno pripelje do dveh rešitev — rekurzivne in z uporabo datotek. Obe uporabljata indukcijo in posamično dodajanje ministrov.

Rekurzivna rešitev:

```
program MinistriA;

const N = 11; { Največ ministrov }

  function IzpisiKoalicije(Stevilo: integer): integer;
  type Koal = set of 1..N;
       Iter = record
                Koliko: integer;
                Koalicije: array [1..N] of Koal;
              end;
  var a: Iter; Stevec: integer;

    { Izpiše vsa taka razbitja ministrov na koalicije, ki jih je moč dobiti,
      če v razbitje a dodamo vse ministre od Naslednji do Stevilo. }
    procedure RekurzivnoIzpisi(var a: Iter; Naslednji: integer);

      procedure Izpisi(var a: Iter);

        procedure IzpisiEno(k: Koal);
        var i: integer;
        begin
          for i := 1 to Stevilo do
            if i in k then Write(Chr(i + Ord('A') - 1));
```

```
      end; {IzpisiEno}

    var i: integer;
    begin {Izpisi}
      for i := 1 to a.Koliko do begin
        if i > 1 then Write(',');
        IzpisiEno(a.Koalicije[i]);
      end; {for}
      WriteLn; Stevec := Stevec + 1;
    end; {Izpisi}

  var i: integer;
  begin {RekurzivnoIzpisi}
    a.Koalicije[a.Koliko + 1] := [];
    for i := 1 to a.Koliko + 1 do begin
      { Poskusimo dodati naslednjega ministra v koalicijo i.
        i = a.Koliko + 1 pomeni, da tvori sam svojo (novo) koalicijo. }
      a.Koalicije[i] := a.Koalicije[i] + [Naslednji];
      if i = a.Koliko + 1 then a.Koliko := a.Koliko + 1;
      { Če je še kaj ministrov, izvedimo rekurzivni klic; sicer izpišimo razbitje. }
      if Naslednji = Stevilo
        then Izpisi(a)
        else RekurzivnoIzpisi(a, Naslednji + 1);
      { Izbrišimo ministra iz i-te koalicije, da ga bomo lahko dali še v }
      a.Koalicije[i] := a.Koalicije[i] - [Naslednji];      { kakšno drugo. }
    end; {for}
    { V zadnji iteraciji smo ustvarili novo koalicijo, pa jo zdaj pobrišimo. }
    a.Koliko := a.Koliko - 1;
  end; {RekurzivnoIzpisi}

begin {IzpisiKoalicije}
  Stevec := 0; a.Koliko := 0;
  RekurzivnoIzpisi(a, 1);
  IzpisiKoalicije := Stevec;
end; {IzpisiKoalicije}

var Stevilo: integer;
    StMin: integer;
begin {MinistriA}
  Write('Vnesi število ministrov (od 1 do ', N, '): ');
  ReadLn(StMin);
  if (StMin < 1) or (StMin > N) then WriteLn('Od 1 do ', N, ', sem rekel!')
  else begin
    Stevilo := IzpisiKoalicije(StMin);
    WriteLn('Vseh možnih koalicij je ', Stevilo, '.');
  end;
end. {MinistriA}
```

Oglejmo si še rešitev z uporabo datotek. Za razliko od prejšnjega programa, ki koalicije izpiše na zaslon, jih ta inačica pusti kar v datoteki, kar je verjetno bolj uporabno.

```
program MinistriB;

const N = 11; { Največ ministrov }

  procedure IzracunajKoalicije(Ministrov: integer);
  var f: array [1..2] of text;
      fi, fo: integer;
      i, j: integer;
      s, t: string;
      ch: char;
      Stevec: longint;
  begin
    Assign(f[1], '1'); Rewrite(f[1]); WriteLn(f[1], 'A'); Reset(f[1]); fi := 1;
    Assign(f[2], '2'); Rewrite(f[2]); fo := 2;
    for i := 2 to Ministrov do begin
      { V datoteki fi so vsa razbitja prvih i − 1 ministrov na koalicije.
        V datoteko fo bomo izpisali vsa razbitja prvih i ministrov. }
      Stevec := 0; ch := Chr(i + Ord('A') − 1);
      while not Eof(f[fi]) do begin
        { Preberimo naslednje možno razbitje prvih i − 1 ministrov. Presledek, ki ga
          bomo dodali na konec niza, bo deloval kot stražar za spodnji stavek if. }
        ReadLn(f[fi], s); s := s + ' ';
        { Poskusimo dodati novega ministra v vsako od koalicij. }
        for j := 1 to Length(s) do begin
          if s[j] = ' ' then begin
            t := s; Insert(ch, t, j);
            WriteLn(f[fo], Copy(t, 1, Length(t) − 1));
            Stevec := Stevec + 1;
          end; {if}
        end; {for j}
        WriteLn(f[fo], s, ch);    { Lahko pa ima novi minister sam svojo koalicijo. }
        Stevec := Stevec + 1;
      end; {while}
      if i = Ministrov then WriteLn(f[fo], Stevec, ' koalicij');
      { Zamenjamo vhodno in izhodno datoteko. }
      Rewrite(f[fi]); fi := 3 − fi;
      Reset(f[fo]); fo := 3 − fo;
    end; {for i}
    Close(f[1]); Close(f[2]); Erase(f[fo]);
    WriteLn('Koalicije so shranjene v datoteki ', fi);
  end; {IzracunajKoalicije}

var StMin: integer;
```

```
begin
  Write('Vnesi število ministrov (od 1 do ', N, '): ');
  ReadLn(StMin);
  if (StMin < 1) or (StMin > N)
    then WriteLn('Od 1 do ', N, ' sem rekel!')
    else IzracunajKoalicije(StMin);
end. {MinistriB}
```

Pa še rešitev v pythonu:

```
def koalicije(n):
  if n == 0: yield []; return
  c = chr(ord('A') + n - 1)
  for p in koalicije(n - 1):
    yield p + [c]
    for i in range(len(p)):
      yield p[:i] + [p[i] + c] + p[i + 1:]

# Primer uporabe:
import sys
for p in koalicije(int(sys.stdin.readline())):
  print p
```

Mimogrede, števila, ki nam povedo, na koliko načinov se lahko n ministrov razdeli v k nepraznih skupin, se imenujejo Stirlingova števila druge vrste in jih označujejo na različne načine, npr. $S(n,k)$, $s_n^{(k)}$ in $\left\{{n \atop k}\right\}$. Zgoraj opisani razmislek nam je pokazal, da bi jih lahko računali po formulah $\left\{{n \atop k}\right\} = \left\{{n-1 \atop k-1}\right\} + k\left\{{n-1 \atop k}\right\}$ za $n > 0$ in $\left\{{0 \atop 0}\right\} = 1$, $\left\{{0 \atop k}\right\} = 0$ za $k \neq 0$. Vsote $B_n = \sum_{k=1}^{n} \left\{{n \atop k}\right\}$, ki nam povedo skupno število vseh razbitij n ministrov, pa se imenujejo Bellova števila.[4] Vrednosti B_n za $n = 1, \ldots, 11$ so: 1, 2, 5, 15, 52, 203, 877, 4 140, 21 147, 115 975, 678 570.

N: 8

R1999.3.3

Edine omejitve v zvezi z medsebojnim vrstnim redom tekmovalcev v naši razvrstitvi nastopijo v primerih, ko je a vsaj tako dober kot b, slednji pa ni vsaj tako dober kot a; takrat imamo omejitev, da mora biti a v razvrstitvi pred b.

Rekli smo, da sta dva tekmovalca enako dobra natanko tedaj, ko je vsak od njiju vsaj tako dober kot drugi. Očitno je ta lastnost tranzitivna: če je a enako dober kot b, slednji pa enako dober kot c, sta tudi a in c enako dobra. Torej je smiselno govoriti o celih skupinah enako dobrih tekmovalcev (te skupine naj bodo tudi „maksimalne“ — v tem smislu, da če je v skupini nek tekmovalec a, so v njej tudi *vsi* ostali tekmovalci, ki so enako dobri kot a). V taki skupini

[4] Glej MathWorld *s. vv.* “Stirling Number of the Second Kind”, “Bell Number”; *The On-Line Encyclopedia of Integer Sequences*, A000110.

je vsak enako dober kot vsi ostali in je zato vseeno, v kakšnem medsebojnem vrstnem redu zapišemo tekmovalce iz take skupine.

Definirajmo zdaj, da je neka skupina A vsaj tako dobra kot neka druga skupina B natanko tedaj, ko je vsaj eden od tekmovalcev iz A vsaj tako dober kot vsaj eden od tekmovalcev iz B. Pri skupinah se ne more zgoditi, da bi bili dve različni skupini enako dobri, kajti to bi pomenilo, da je neki $a \in A$ vsaj tako dober kot neki $b \in B$, neki $b' \in B$ pa vsaj tako dober kot neki $a' \in A$; ker sta a in a' oba iz A, sta oba enako dobra; torej je a' vsaj tako dober kot a in zato tudi vsaj tako dober kot b; in podobno sta b in b' enako dobra, ker sta oba iz B; zato pa je b vsaj tako dober kot b', in ker je slednji vsaj tako dober kot a', je tudi b vsaj tako dober kot a'. Iz tega sledi, da sta si b in a' enako dobra, zaradi prej ugotovljene tranzitivnosti pa tudi, da so si vsi tekmovalci iz A in vsi iz B enako dobri. Toda to je nemogoče, ker potem A in B ne bi mogli biti dve različni skupini enako dobrih tekmovalcev.

Zagotovo obstaja neka skupina, za katero velja, da ni nobena vsaj tako dobra kot ta. Kajti če bi za vsako obstajala neka druga vsaj tako dobra, bi recimo bila A_2 vsaj tako dobra kot A_1, pa A_3 vsaj tako dobra kot A_2 in tako naprej; prej ali slej bi se nam skupine začele ponavljati in bi to pomenilo, da je neka A_i vsaj tako dobra kot A_j, ta pa vsaj tako dobra kot A_i; toda to bi pomenilo, da sta enako dobri, kar pa smo že v prejšnjem odstavku spoznali za nemogoče.

Naj bo torej A neka skupina, kateri ni nobena druga vsaj tako dobra. Torej ni nobenih omejitev, ki bi zahtevale, da mora biti kateri iz tekmovalcev te skupine v vrstnem redu za nekim tekmovalcem kakšne druge skupine. Zato lahko kar takoj postavimo na začetek vrstnega reda vse tekmovalce te skupine (njihov medsebojni vrstni red je lahko poljuben, saj so si vsi enako dobri). V nadaljevanju našega razmišljanja nam tekmovalci iz skupine A ne bodo povzročali nikakršnih težav več — kot smo pravkar videli, omejitev, da bi moral biti kdo drug pred njimi, sploh ni; lahko pa obstajajo omejitve, da mora biti kdo drug za njimi, kar pa bo gotovo izpolnjeno, saj smo jih postavili na začetek vrstnega reda. Torej se lahko v nadaljevanju obnašamo, kot da tekmovalcev iz skupine A sploh nikoli ni bilo.

Zdaj zagotovo obstaja neka skupina B, kateri ni nobena druga vsaj tako dobra (lahko da je za B to veljalo še prej, lahko pa je bila prej A vsaj tako dobra kot B, ampak zdaj smo A pač v mislih zavrgli). Enak razmislek kot prej tudi zdaj pokaže, da lahko B takoj postavimo v vrstni red, saj nobena od preostalih skupin ne more zahtevati, da bi bila v njem pred tekmovalci skupine B. Zdaj lahko tudi na B pozabimo. Potem poiščemo naslednjo primerno skupino (tako, ki ji ni nobena vsaj tako dobra) in tako nadaljujemo, dokler ne obdelamo vseh skupin.

Naš algoritem je torej tak:

1 Poišči skupine enako dobrih tekmovalcev.

2 Ponavljaj, dokler je ostalo še kaj skupin:
3 Naj bo A poljubna taka skupina, za katero velja, da ni nobena druga vsaj tako dobra kot A.
4 Izpiši vse tekmovalce skupine A v poljubnem vrstnem redu.
5 Zbriši skupino A.

Pravzaprav bi se spodobilo, če bi te korake opisali malo podrobneje. Posvetimo se točki (1), s katero je še največ dela; o ostalih lahko bralec razmisli sam. Skupine enako dobrih tekmovalcev lahko poiščemo takole. Na začetku za vsakega tekmovalca a vemo, kateri so ga premagali v neposrednih tekmah; naj bo $P(a)$ množica vseh teh. Označimo s $P^\star(a)$ množico vseh, ki so vsaj tako dobri kot a. Na začetku lahko postavimo $P^\star(a) := P(a)$, nato pa za vsakega tekmovalca b, ki ga dodamo v $P^\star(a)$, dodamo v $P^\star(a)$ še vse tekmovalce iz $P(b)$, kajti če je b vsaj tako dober kot a, tisti iz $P(b)$ pa so b-ja nekoč premagali, so tudi oni vsaj tako dobri kot a. Ko tako za vsakega tekmovalca a poznamo $P^\star(a)$, gremo lahko za vsakega a in za vse $b \in P^\star(a)$ pogledat, če je slučajno tudi $a \in P^\star(b)$, in če je res tako, vemo, da sta a in b enako dobra.

Algoritem *SkupineEnakoDobrih:*
Vhod: množica tekmovalcev T
za vsakega $a \in T$ še množica $P(a)$ tistih, ki so ga premagali.
Izhod: *ŠtSkupin* pove, koliko skupin je;
skupina[a] pove, v kateri skupini je tekmovalec a.

```
 1  for each a ∈ T do
 2      skupina[a] := 0; P*(a) := {}; Q := {a};
 3      while Q ≠ {} do
 4          naj bo b poljuben element Q;
 5          Q := Q − {b};
 6          for each c ∈ P(b) do if c ∉ P*(a) then
 7              P*(a) := P*(a) ∪ {c}; Q := Q ∪ {c};
 8  ŠtSkupin := 0
 9  for each a ∈ T do if skupina[a] = 0 then
10      ŠtSkupin := ŠtSkupin + 1; skupina[a] := ŠtSkupin;
11      for each b ∈ P*(a) do
12          if a ∈ P*(b) then skupina[b] := ŠtSkupin;
```

V teoriji grafov bi rekli, da smo iz rezultatov tekmovanja naredili usmerjen graf, v katerem vsakega tekmovalca predstavlja neka točka, usmerjena povezava od a do b pa je prisotna natanko tedaj, ko je a premagal b-ja. To, da je a vsaj tako dober kot b, pomeni, da je točka b v grafu dosegljiva iz točke a; skupinam enako dobrih tekmovalcev ustrezajo krepko povezane komponente grafa. Potem naredimo nov graf, v katerem je po ena točka za vsako krepko povezano komponento in povezava od A do B natanko tedaj, če obstaja v

prvotnem grafu kakšna povezava od kakšnega $a \in A$ do kakšnega $b \in B$. Ta novi graf je zanesljivo acikličen in če ga topološko uredimo, dobimo vrstni red, v katerem je treba izpisati povezane komponente prvotnega grafa.

Gornji algoritem za odkrivanje skupin enako dobrih tekmovalcev (torej: za iskanje krepko povezanih komponent) ni najboljši možni, vendar je zelo preprost. Običajni algoritem za iskanje krepko povezanih komponent je učinkovitejši, vendar je bolj zapleten in ga tu ne bomo opisovali. Najde se ga v mnogih knjigah o algoritmih (npr. Cormen *et al.*, *Introduction to Algorithms*, razdelek 23.5 v prvi izdaji, 22.5 v drugi).

R1999.3.4

N: 9

Glavni računalnik naj pošlje vsako sekundo svojo številko vsem računalnikom v omrežju. Ostali računalniki poslušajo in če v neki sekundi ne prejmejo iz omrežja nobene številke, si to razlagajo kot izpad glavnega računalnika. V tem primeru vsak računalnik, ki to opazi, razpošlje vsem svojo številko; za glavnega obvelja tisti, ki je imel najmanjšo številko. Vsak podporni računalnik poskuša najprej prebrati prispele podatke iz omrežja; če glavni računalnik obstaja in deluje normalno, bo prišla le njegova številka in je stvar opravljena; če je glavni računalnik izpadel in je že več pomožnih to opazilo ter poslalo svoje številke, lahko določimo za glavnega tistega z najmanjšo številko; sicer pa pošljemo svojo številko v omrežje in bomo v naslednji sekundi preverili, ali smo postali glavni računalnik ali ne (kajti možno je, da je v istem času poslal svojo številko še kdo drug in mogoče bo on postal glavni, ne pa mi).

Upoštevati moramo, da lahko sporočilo potuje do drugih računalnikov skoraj celo sekundo. Poleg tega ne bi bilo realistično pričakovati, da se podprogram VsakoSekundo kliče na vseh računalnikih naenkrat. Recimo, da pošlje računalnik 1 sporočilo računalniku 2 ob času t in da sporočilo pride do računalnika 2 ob času $t + 1 - \varepsilon$; in recimo, da se na računalniku 2 izvede VsakoSekundo ob času $t + 1 - 2\varepsilon$, ko sporočilo do njega še ni prišlo; računalnik 2 bo torej sporočilo videl šele ob času $t + 2 - 2\varepsilon$, skoraj dve sekundi po tistem, ko je bilo odposlano.

Če torej ob nekem klicu VsakoSekundo opazimo, da od strežnika v zadnji sekundi nismo dobili nobenega sporočila, to še ne pomeni, da je strežnik izpadel iz omrežja. Čisto mogoče je, da je z njim vse v redu, le da smo njegovo predzadnje sporočilo prevzeli že ob prejšnjem klicu VsakoSekundo, njegovo zadnje pa se je malo zakasnilo in nas še ni doseglo, zato ob trenutnem klicu VsakoSekundo pač ne vidimo od glavnega računalnika nobenega sporočila. Zato lahko o izpadu glavnega računalnika zanesljivo govorimo šele, če v dveh zaporednih klicih VsakoSekundo opazimo, da nas ne čaka nobeno sporočilo. Računalnik naj se tudi ne razglasi za glavnega, dokler ni zmagal pri glasovanju v dveh zaporednih sekundah — po prvi sekundi namreč še obstaja možnost, da je poslal svojo številko še kak računalnik z manjšo številko od naše, ki še ni videl našega sporočila, mi pa še ne njegovega. Zato, če bi se naš računalnik takoj razglasil

za glavnega, bi se lahko zgodilo, da bi se tisti drugi kasneje tudi, pa bi imeli potem v sistemu dva glavna računalnika, ki bi vpila drug mimo drugega.

```
var SemGlavni: boolean; { Trenutno stanje računalnika. }
    MinStPrejSek,       { Najmanjša številka, ki smo jo prejeli v prejšnji sekundi. }
    StSek: integer;     { Zacni postavi StSek na 0, VsakoSekundo jo poveča za 1. }

procedure Zacni;
begin
  SemGlavni := false;
  MinStPrejSek := -1;
  StSek := 0;
  PostaviPodporniRac;
end; {Zacni}

procedure VsakoSekundo;
var St, MinSt, Glavni: integer;
begin
  StSek := StSek + 1;
  if SemGlavni then begin
    PosljiVsem(VrniStRacunalnika);
  end
  else begin
    { Katera je najmanjša številka, prejeta v zadnji sekundi? }
    MinSt := -1;
    while Prejmi(St) do if (MinSt = -1) or (St < MinSt) then MinSt := St;
    { Katera je najmanjša številka, prejeta v zadnjih dveh sekundah? }
    Glavni := MinStPrejSek;
    if (Glavni = -1) or ((MinSt > 0) and (MinSt < Glavni)) then Glavni := MinSt;
    { Poglejmo zdaj, kako je z glasovanjem. }
    if Glavni = -1 then begin
      { Glavnega računalnika ni; predlagajmo sebe za glavnega. Če smo se ravnokar
        zbudili (StSek = 1), tega raje ne storimo, saj se lahko po naključju
        zgodi, da v tisti ≤ 1 sekundi, kolikor dolgo smo budni, ne dobimo
        nobenega sporočila, čeprav v omrežju v resnici obstaja glavni računalnik.
        V tem primeru bi delali zgolj zmedo, če bi zdaj začeli vpiti svojo številko. }
      if StSek >= 2 then PosljiVsem(VrniStRacunalnika);
    end
    else begin
      { Glasovanje je v teku, glavni bo tisti z najmanjšo številko. Če smo to mi,
        se spodobi, da pošljemo takoj svojo številko, drugače bomo to storili šele
        čez eno sekundo in se bodo drugi medtem še pregovarjali. Preden pa se
        dokončno razglasimo za glavnega, bi radi, da bi bila naša številka najmanjša
        dve sekundi zaporedoma. To je potrebno zaradi možnosti, da se naša
        sporočila na poti zadržijo eno sekundo in smo lahko zato šele po dveh
```

```
        sekundah prepričani, da ni še kje kak drug računalnik, ki za našo zmago
        ne ve in še kar vpije svojo številko. }
      if VrniStRacunalnika = Glavni then begin
        if MinStPrejSek = Glavni then
          begin SemGlavni := true; PostaviGlavniRac; end;
        PosljiVsem(VrniStRacunalnika);
      end else begin
        SemGlavni := false;
        PostaviPodporniRac;
      end; {if}
    end; {if}
    MinStPrejSek := MinSt;
  end; {if}
end; {VsakoSekundo}
```

Morebitna slabost te rešitve je, da se ne trudi preprečiti istočasnega obstoja več glavnih računalnikov. Do tega bi lahko prišlo, če bi bilo nekaj povezav v omrežju prekinjenih in bi omrežje zaradi tega razpadlo na več nepovezanih delov. Računalniki v tistih delih omrežja, iz katerih se dosedanjega glavnega računalnika ne vidi, bi mislili, da je glavni računalnik izpadel, tako da bi zdaj vsak del omrežja izglasoval svoj glavni računalnik. To je težko preprečiti, saj računalniki, ki glavnega ne vidijo več, ne vedo, ali je ta prenehal delovati ali pa je prišlo le do prekinitve kakšne omrežne povezave. Lahko pa preprečimo vsaj to, da po ponovni vzpostavitvi povezav ostane v omrežju več glavnih računalnikov, ki vpijejo drug mimo drugega. Začetek podprograma VsakoSekundo lahko popravimo tako, da tudi glavni računalnik posluša, kaj pošiljajo drugi računalniki, in če sliši nižjo številko od svoje, razglasi sebe za podpornega:

```
  if SemGlavni then begin
    MinSt := −1;
    while Prejmi(St) do if (MinSt = −1) or (St < MinSt) then MinSt := St;
    if (MinSt > −1) and (MinSt < VrniStRacunalnika) then
      { Eden od drugih glavnih računalnikov ima manjšo številko
        kot mi, zato postanimo podporni računalnik. }
      begin SemGlavni := false; PostaviPodporniRac end
    else { Naš računalnik lahko ostane glavni. }
      PosljiVsem(VrniStRacunalnika);
  end
```

REŠITVE NALOG PRVEGA TEKMOVANJA IZ UNIXA

R1999.U.1

N: 10

Pomagali si bomo s programi `tr`, `sort` in `uniq`, ki jih bomo povezali s cevmi (kar pomeni, da ukazna lupina ob poganjanju teh programov poskrbi za to, da se standardni izhod enega

programa spelje na standardni vhod naslednjega in tako naprej). Rešitev je lahko takšna:

```
cat datoteka.txt | tr " " "\n" | sort | uniq -c | sort -n
```

Program `cat` samo bere vhodno datoteko in jo piše na svoj standardni izhod. Cev bo poskrbela, da pridejo ti podatki na standardni vhod programa `tr`, ki smo mu naročili, naj vse presledke spremeni v znake za konec vrstice. Tako pride vsaka beseda v samostojno vrstico; dobljeno besedilo pošljemo programu `sort`, ki uredi vrstice po abecedi. Zdaj torej pridejo vse pojavitve posamezne besede skupaj. Program `uniq` bere svoj vhod in če je več zaporednih vrstic enakih, izpiše od takšne skupine le eno vrstico; s stikalom `-c` smo zahtevali, naj izpiše še število vrstic v skupini. Zdaj torej dobimo v vsaki vrstici niz oblike „`123 bla`“, ki nam pove, da se je beseda `bla` v vhodni datoteki pojavila 123-krat. Posledica tistega prvega urejanja je tudi to, da so besede tu navedene po abecedi; ker pa bi jih radi uredili po frekvenci, pokličimo `sort` še enkrat. Tokrat mu s stikalom `-n` povemo, naj ignorira morebitne presledke na začetku vrstice (ki jih je mogoče izpisal `uniq`) in nato začetek vrstice gleda kot število, ne kot niz (drugače bi namreč lahko ugotovil, da je na primer `10` manjše od `2`, ker bi tadva niza primerjal znak po znak in videl, da je `1` leksikografsko pred `2`).

Opisana rešitev se zanaša na dejstvo, da so v vhodni datoteki le črke in presledki (kot zagotavlja besedilo naloge). Če bi bilo v datoteki še kaj drugega, na primer ločila, bi bilo razbijanje na besede malo bolj zapleteno; za prvo silo bi lahko na primer vse ne-črke spremenili v presledke:

```
sed "s/[^A-Za-z]/ /g"
```

Tu smo uporabili program `sed`; ukaz `s/`*vzorec*$_1$`/`*vzorec*$_2$`/`*zastavice* zamenja pojavitve vzorca 1 z vzorcem 2; zastavica `g` zahteva zamenjavo vseh pojavitev (namesto samo prve). Regularni izraz `[^A-Za-z]` se ujame z vsakim znakom, ki ni ena od črk *A, . . . , Z* ali *a, . . . , z*.

Naša rešitev je zaenkrat tudi občutljiva na razlike med velikimi in malimi črkami; tako sta na primer *Bla* in *bla* zanjo dve povsem različni besedi. Če bi hoteli pred obdelavo spremeniti velike črke v male, bi si spet lahko pomagali s programom `tr`:

```
tr [:upper:] [:lower:]
```

`tr` v splošnem vzame kot parametra dva niza, nato pa bere standardni vhod in vsako pojavitev kakšne črke prvega niza zamenja z istoležno črko drugega niza; rezultat izpisuje na standardni izhod. Parameter [:upper:] je enakovreden nizu „`ABC...XYZ`“, podobno pa [:lower:] nizu „`abc...xyz`“.

Še ena morebitna slabost naše rešitve bi se pojavila pri delu z dolgimi besedili. Recimo, da imamo besedilo z n besedami, od katerih je k različnih.

Naša rešitev bi za urejanje (prvi klic programa `sort`) porabila $O(n)$ dodatnega pomnilnika (ali prostora na disku) in, če uporablja `sort` katerega od splošnonamenskih postopkov za urejanje, tudi $O(n \log n)$ časa. Pri dovolj velikih n bi torej lahko postal učinkovitejši naslednji postopek: besedilo berimo vrstico po vrstico in ga sproti režimo na besede, te pa shranjujmo v razpršeni tabeli, kjer ob vsaki besedi tudi piše, kolikokrat se pojavlja. V razpršeni tabeli je torej vsaka različna beseda omenjena le enkrat, zato je poraba pomnilnika le $O(k)$. Urejanje potrebujemo zdaj le na koncu, da uredimo besede po frekvenci; ker imamo takrat vsako besedo v seznamu omenjeno le enkrat, porabimo za to le $O(k \log k)$ časa. Če prištejemo še čas, potreben za branje vhodnega besedila, imamo skupno zahtevnost $O(n + k \log k)$. Lepo pri tem je, da je k (število različnih besed) v praksi precej manjši od n (števila vseh besed), saj se mnoge besede pojavljajo po večkrat (in to nekatere zelo velikokrat). Znani Heapsov zakon ugotavlja, da je k približno enak $c \cdot n^d$ za neki konstanti c in d (ki sta odvisni od jezika in vrste besedil, s katerimi delamo); glavno je, da je d običajno precej manjši od 1. Za poskus smo vzeli zbirko 806 791 Reutersovih člankov in opazovali, kako se povečuje k, če gledamo vse več člankov (in s tem povečujemo n); izkazalo se je, da je $k \approx 39 \cdot n^{0,48}$ (cela zbirka ima približno 182 milijonov besed, od tega 380 tisoč različnih). Tukaj lahko torej rečemo, da je $k = O(\sqrt{n})$.

Sledi primer rešitve z razpršeno tabelo:

```
import sys
frekvence = {}
for vrstica in sys.stdin:
  for beseda in vrstica.split():
    try: frekvence[beseda] += 1
    except: frekvence[beseda] = 1
seznam = [(frekvence[beseda], beseda) for beseda in frekvence]
seznam.sort()
for (frekvenca, beseda) in seznam: print "%s %d" % (beseda, frekvenca)
```

Za razbijanje vrstice na besede (pri presledkih) smo uporabili pythonovo funkcijo `split`. Pri dostopu do razpršene tabele moramo posebej obravnavati primer, ko na neko besedo naletimo prvič; takrat je v razpršeni tabeli še ni. Stavek `frekvence[beseda] += 1` sproži takrat izjemo (*exception*), ker bi moral najprej prebrati iz razpršene tabele vrednost, ki pripada tej besedi, da bi jo nato lahko povečal za 1. To izjemo prestrežemo (stavek **try**...**except**) in v tem primeru preprosto vpišemo besedo v razpršeno tabelo z začetno frekvenco 1 (ker smo pravkar videli njeno prvo pojavitev). Namesto te rešitve s **try**...**except** bi lahko tudi eksplicitno preverili, če je beseda že v razpršeni tabeli:

```
if beseda in frekvence: frekvence[beseda] += 1
else: frekvence[beseda] = 1
```

Ali pa bi uporabili metodo `get`, ki ji lahko povemo, kakšno vrednost naj vrne, če besede še ni v tabeli:

```
frekvence[beseda] = frekvence.get(beseda, 0) + 1
```

Vendar se izkaže, da je zadnja različica približno 15 % počasnejša.

Za primerjavo smo pognali obe rešitvi, torej tisto s `sort` in `uniq` ter tisto z razpršeno tabelo, na nekaj dolgih angleških besedilih. Izkaže se, da je rešitev z razpršeno tabelo hitrejša šele pri besedilih, dolgih več deset milijonov znakov, vendar je to verjetno deloma tudi posledica neučinkovitosti pythonovega interpreterja.

		Št. besed		Čas izvajanja	
Besedilo	Dolžina	vseh	različnih	`sort` + `uniq`	razp. tabela
Gibbonova *Zgodovina*	9,0 MB	1,5 M	55 K	4,3 s	5,9 s
Dickensova dela	18,9 MB	3,6 M	39 K	10,1 s	9,9 s
Reutersovi članki 1/8	145 MB	22,9 M	160 K	81 s	59 s
Reutersovi članki	1147 MB	183 M	381 K	1129 s	462 s

Besedila so ista, kot smo jih že uporabili pri poskusih v rešitvi naloge 1989.2.3.

N: 10

R1999.U.2

Pri tej nalogi moramo poznati ali iznajti pojem zaklepanja. Ko en uporabnik dela z datoteko, mora biti za druge nekako „zaklenjena“, tako da ne bodo mogli do nje (oz. bodo lahko vsaj opazili, da bi bilo bolje, če je ne bi odpirali).

Rešitev z nevsiljenim zaklepanjem. Izvršilno datoteko `vi` preimenujmo v `vi_old` in jo zamenjajmo z našo skripto:

```
#!/usr/bin/perl
use Fcntl ':flock';   # definicija konstant LOCK_*

$preimenovan_vi = "vi_old";
$ime_datoteke = $ARGV[0];
$ime_lock_datoteke = $ime_datoteke . ".lock";

open(FH, ">$ime_lock_datoteke");
$return = flock(FH, LOCK_EX | LOCK_NB);

if (! $return) {
  print "Čakam, da se datoteka \"$ime_datoteke\"odklene.\n"
  print "Za prekinitev pritisnite CTRL+C.\n";
  flock(FH, LOCK_EX);
}

system($preimenovan_vi . " " . $ime_datoteke);
flock(FH, LOCK_UN);
close(FH);
```

Skripta uporablja sistemski klic flock, ki zagotavlja nevsiljeno (advisory) zaklepanje datotek (torej lahko nek drug proces takšno datoteko še vseeno odpre, npr. s funkcijo open, četudi jo je nek drug proces ekskluzivno zaklenil s flock). Pred klicem urejevalnika besedila `vi` se tako ustvari datoteka-ključavnica, ki nakazuje, da je datoteka, ki jo urejamo, zaklenjena. Klicu flock podamo zastavici LOCK_EX, ki zahteva izključni dostop do ključavnice, in LOCK_NB, ki mu pove, naj ne čaka, da se bo ključavnica sprostila, če je jo je zasegel že kdo drug. Zato lahko iz vrednosti, ki jo flock vrne, ugotovimo, če je uspel ključavnico zaseči ali ne; če je ni, uporabnika obvestimo, da bo treba čakati, in pokličemo flock še enkrat, tokrat brez LOCK_NB.

Zaklepno datoteko moramo odpreti v pisalnem načinu (predznak > ob odpiranju datoteke), kar datoteko tudi ustvari, če še ne obstaja. Ta način sicer ob odpiranju tudi poreže datoteko na dolžino 0 bytov (za razliko od >>), vendar nas to ne bo motilo, ker vanjo tako ali tako ne bomo ničesar pisali; glavno je, da ostane to še vseeno ista datoteka (ker če bi > obstoječo datoteko na primer pobrisal in ustvaril novo, bi bilo to za zaklepanje seveda neuporabno: ko bi nek program datoteko zaklenil, drugi tega ne bi opazili, ker sploh ne bi gledali iste datoteke).

Omeniti velja, da rešitev deluje le pod sistemi, ki imajo implementiran sistemski klic flock (linux in večina drugih unixov). Prav tako rešitev ne deluje na oddaljeno priklopljenih datotečnih sistemih NFS (in sorodnih); tam je potrebno datoteko zakleniti preko sistemskega klica fcntl.

Po svoje bi bilo lepo, če bi naš program na koncu tudi pobrisal zaklepno datoteko, da se nam ne bi take datoteke kopičile na disku, vendar pa bi utegnile biti s tem težave. „Črni scenarij“ bi bil takšen: program A odpre datoteko, jo zaklene in požene `vi`; program B jo odpre in čaka; program A se vrne iz `vi` in datoteko odklene, zapre in pobriše; B jo zaklene in požene `vi`; program C jo odpre, vidi, da je prosta, in jo tudi zaklene in požene `vi`. Kaj se je zgodilo? Ko je A pobrisal datoteko, je operacijski sistem še ni zares pobrisal, ker jo je imel B še odprto, vendar pa jo je „skril“ pred drugimi: ko poskuša C odpreti datoteko s tem imenom, dobi v resnici že novo datoteko, ne pa tiste, ki jo ima B še odprto (in zaklenjeno).

Rešitev s pomočjo dostopnih pravic. Za zaseganje zaklepne datoteke lahko namesto funkcije flock uporabimo tudi običajne unixove dostopne pravice do datoteke. Zaklepno datoteko poskusimo ustvariti tako, da bomo imeli bralni in pisalni dostop do nje samo mi, drugi uporabniki pa ne. Če datoteko ureja že kdo drug, nam to ne bo uspelo, saj ima tisti drugi uporabnik že ekskluzivni dostop do zaklepne datoteke; tako bomo vsaj vedeli, pri čem smo. Drugače pa bomo zaklepno datoteko uspešno ustvarili in tako tudi vedeli, da lahko poženemo `vi`.

Pomembno je, da ne gremo najprej preverjat, če datoteka že obstaja, in jo šele nato poskusimo ustvariti; če bi počeli tako, bi nas lahko med našim pre-

verjanjem in ustvarjanjem datoteke prehitel kdo drug, ki bi ravno tako opazil, da še ne obstaja, in jo nato ustvaril, še preden bi jo ustvarili mi. Namesto tega bomo datoteko kar takoj poskusili ustvariti, nato pa bomo samo pogledali, če se je to posrečilo ali ne.

```
#!/bin/bash
if ( umask u=rw,g=,o= ; touch $1.lock > /dev/null 2>&1 )
then
  vi_old $1
  rm -f $1.lock
else
  echo "Datoteka $1 je trenutno zaklenjena."
fi
```

Za ustvarjanje zaklepne datoteke smo si pomagali s programom `touch`, ki ustvari prazno datoteko, če ta prej še ni obstajala, sicer pa ji le postavi čas zadnje spremembe na trenutni čas. Pred tem smo z lupininim vgrajenim ukazom `umask` zahtevali, naj se datoteke ustvarja tako, da jih bomo lahko mi (`u`, *user*) brali in pisali (`rw`), drugi iz naše skupine (`g`, *group*) in ostali uporabniki (`o`, *others*) pa je ne bodo smeli niti brati niti pisati. S tem zagotovimo, da če nam bo datoteko uspelo ustvariti, je drugi uporabniki (razen administratorja) ne bodo mogli povoziti.

Nastavitve, ki jih podamo prek klica `umask`, bi v splošnem veljale vse do naslednje spremembe, ne le za prvi naslednji ukaz. Ker pa hočemo, da bi bila ta sprememba pri našem programu le začasna (da bi vplivala samo na program `touch`), smo `umask` in `touch` ovili v oklepaje in jima s tem dodelili ločeno podlupino, iz katere se sprememba `umask` ne vidi navzven.

Izpis programa `touch` nas ne zanima, zato smo njegov standardni tok za napake preusmerili na običajni standardni izhod (`2>&1`), tega pa na `/dev/null`. Vrednost, ki jo vrne `touch` in z njo pove, če se je zaklepne datoteke uspel „dotakniti“ ali ne, pa bomo uporabili kot pogoj v stavku **if**.

Ko se vrnemo iz programa `vi`, moramo zaklepno datoteko pobrisati, kajti dokler obstaja, je datoteka z vidika vseh ostalih uporabnikov zaklenjena. Programu `rm` s stikalom `-f` naročimo, naj nas nič ne sprašuje in naj se tudi ne zmeni za to, če bi mogoče zaklepna datoteka ne obstajala.

Lepo pri tej rešitvi je, da je preprostejša od prve in da zaklepne datoteke na koncu pobriše za sabo. Za razliko od prve pa nas ta rešitev ne varuje pred tem, da bi isti uporabnik odprl neko datoteko iz več različnih procesov. Še posebej nerodno pri tem je, da bi prvi od teh procesov, ko bi se vrnil iz programa `vi`, zaklepno datoteko pobrisal in bi bila potem z vidika ostalih uporabnikov datoteka odklenjena, čeprav je v resnici mogoče še odprta v drugih procesih prvega uporabnika.

Mimogrede omenimo še, da za razliko od prvotne različice programa `vi`

nekatere novejše različice, na primer zelo razširjeni `vim` („`vi` improved“), že same po sebi skrbijo tudi za zaklepne datoteke.

R1999.U.3

N: 10

Če je datoteka dovolj majhna, jo lahko kar celo preberemo v pomnilnik, premešamo njene vrstice in jih izpišemo v novem vrstnem redu. Primer rešitve v pythonu:

```
import sys, random
vrstice = sys.stdin.readlines()
random.shuffle(vrstice)
for s in vrstice: sys.stdout.write(s)
```

Pravzaprav bi morali paziti še na možnost, da se zadnja vrstica vhodne datoteke mogoče ne konča z znakom za konec vrstice. Ker ta vrstica po mešanju najbrž ne bo več zadnja, bi ji morali znak za konec vrstice dodati, da se ne bo v izhodni datoteki sprijela z naslednjo.

Za mešanje vrstic smo zgoraj uporabili pythonovo funkcijo shuffle iz modula random. Oglejmo si še, kako bi lahko premešali vrstice brez takšne funkcije:

```
import sys, random
vrstice = sys.stdin.readlines()
n = len(vrstice)
while n > 0:
  # Izpisati bo treba še vrstice[0], ..., vrstice[n − 1] (v naključnem vrstnem redu).
  i = random.randrange(n)    # Naključno število iz množice {0, 1, ..., n − 1}.
  sys.stdout.write(vrstice[i])
  vrstice[i] = vrstice[n − 1]; n = n − 1
```

V vsaki iteraciji glavne zanke si naključno izberemo eno od n vrstic, ki jih doslej še nismo izpisali. Izbrano vrstico izpišemo, nato pa na njeno mesto v tabeli vrstica postavimo zadnjo še neizpisano vrstico, torej niz vrstica[n − 1]. Potem lahko n zmanjšamo za 1 in postopek se nadaljuje. Z indukcijo se lahko hitro prepričamo, da ima pri takšnem postopku res vsaka permutacija naše tabele vrstic enake možnosti, da bo izbrana (če le randrange(n) res vrača vsa števila od 0 do n − 1 z enako verjetnostjo).

Slabo pri tem postopku je, da mora prebrati vse vrstice v pomnilnik, torej ga ne bi mogli uporabiti na zelo velikih datotekah. Tu bi bilo pametno narediti novo kopijo datoteke, pri kateri bi na začetek vsake vrstice vrinili neko primerno veliko psevdonaključno število. Vrstice te datoteke potem uredimo, v urejeni datoteki porežimo števila, ki smo jih prej vrinili na začetek vsake vrstice, pa nam ostanejo vrstice prvotne datoteke v premešanem vrstnem redu. Za urejanje bi morali uporabiti kakšnega od algoritmov zunanjega (eksternega) urejanja, ki si pomagajo s pomožnimi datotekami in zato ne porabijo veliko pomnilnika; prepustimo to kar standardnemu programu `sort`. Vse opisane korake lahko opravimo kar iz ukazne lupine:

```
awk '{ print rand(), $0 }' urejena.txt | sort -n | cut -d " " -f 2-
```

Program `awk` bere vhodno datoteko (`urejena.txt`) vrstico za vrstico in na vsaki vrstici izvede stavek, ki smo mu ga podali v zavitih oklepajih. Stavek **print** v `awk`u izpiše svoje argumente, vmes po en presledek, na koncu pa prazno vrstico. Funkcija rand, ki je že vgrajena v `awk`, vrne naključno število med 0 in 1; v spremenljivki `$0` pa nam `awk` hrani vsebino trenutne vrstice.

Za urejanje uporabimo program sort, ki mu s stikalom `-n` naročimo, naj začetke vrstic pri urejanju gleda kot števila (iz enakih razlogov kot pri 1. nalogi, glej str. 30). Na koncu bi radi tista naključna števila z začetkov vrstic pobrisali, kar lahko naredimo s programom `cut`. Ta naj razreže vsako vrstico pri presledkih (`-d " "`) in izpiše vse kose od drugega naprej (`-f 2-`); zavržemo torej le prvi kos, ki vsebuje naključno število, ki smo ga dodali pred urejanjem.

N: 10

R1999.U.4

Najprej preberimo datoteko `/etc/hosts` in si pripravimo razpršeno tabelo, ki bo preslikovala številske naslove v imena. Potem lahko beremo vhodno datoteko (spodnji program bere kar standardni vhod) in v njej iščemo pojavitve številskih naslovov IP ter jih zamenjujemo z imeni računalnikov. Pri iskanju naslovov IP si lahko pomagamo z regularnim izrazom, saj dobro poznamo zgradbo takega naslova: sestavljen je iz štirih skupin števk, vsaka ima največ tri števke, med skupinami pa so pike; poleg tega nam naloga zagotavlja še, da se naslovi IP v vhodni datoteki ne stikajo z drugimi znaki razen s presledki.

```
#!/usr/bin/perl
open(HOSTS_FILE, "/etc/hosts");
while ($line = <HOSTS_FILE>) {
  chomp $line;
  ($ip, $fqdn) = split(/ +/, $line);
  $hostbyip {$ip} = $fqdn;
}
close(HOSTS_FILE);

while (<>) {
  s/\b(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})\b/$hostbyip{$1}/g;
  print;
}
```

Pri vsaki vrstici datoteke `/etc/hosts` najprej s perlovo funkcijo chomp odrežemo znak za konec vrstice, nato pa jo s split razcepimo pri presledku (ali presledkih). Tako dobimo dva kosa, številski naslov IP in pripadajoče ime, ki ju nato vpišemo v tabelo hostbyip.

Za iskanje in zamenjevanje številskih naslovov z imeni uporabljamo operator `s`/*vzorec*$_1$/*vzorec*$_2$/*zastavice*, ki zamenja pojavitve vzorca 1 z vzorcem 2; zastavica `g` zahteva zamenjavo vseh pojavitev, ne pa le prve. Pri regularnem

izrazu, ki naj bi odkrival številske naslove IP, upoštevajmo naslednje: pred piko je treba postaviti poševnico \, ker drugače pika sama po sebi deluje kot metaznak, ki se ujame s poljubnim znakom pregledovanega niza; metaznak `\d` se ujame s katero koli števko; metaznak `\b` se ujame z nizom dolžine 0, vendar le na robovih besed. Tisti del regularnega izraza, ki se bo ujel s številskim naslovom, postavimo v oklepaje; tako se bomo lahko na ta del vhodnega niza sklicevali še iz zamenjavnega vzorca (s spremenljivko `$1`). Zamenjavni vzorec `$hostbyip{$1}` se torej razširi v niz, ki v razpršeni tabeli `hostbyip` pripada ključu `$1`; ker je slednji ravno številski naslov IP, se bo zamenjavni vzorec razširil v pripadajoče ime računalnika.