28. državno tekmovanje v znanju računalništva (2004)

NALOGE ZA PRVO SKUPINO

Svoje odgovore dobro utemelji. Če pišeš izvorno kodo programa ali podprograma, jo obvezno tudi komentiraj in v nekaj stavkih z besedami opiši, na kakšni ideji temelji tvoja rešitev.

2004.1.1 SMS

Rešitev: str. 17

Pri pisanju sporočil na prenosnem telefonu moramo vsakič, ko dve sosednji črki pripadata isti tipki prenosnega telefona, malce počakati. Če želimo na primer natipkati besedo „bacil“, moramo pritisniti tipke

2 2 (za *b*) 2 (za *a*) 2 2 2 (za *c*) 4 4 4 (za *i*) 5 5 5 (za *l*),

kar pomeni, da moramo dvakrat malce počakati — preden natipkamo „a“ in preden natipkamo „c“.

Napiši program, ki za prebrani stavek izračuna, kolikokrat bomo pri pisanju sporočila na prenosnem telefonu morali počakati. Predpostavi, da stavek vsebuje le male črke angleške abecede in presledke.

Razporeditev črk po tipkah:

1	2	3	4	5	6	7	8	9
(presledek)	abc	def	ghi	jkl	mno	pqrs	tuv	wxyz

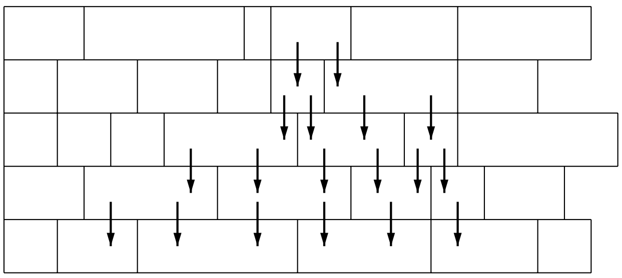
2004.1.2 Ploščice

Rešitev: str. 17

Po tleh smo v nekaj vrstic položili ploščice, ki so vse enako visoke, vendar različno široke. Ploščice so zložene stikoma (ena ob drugi) tako, da v posamezni vrstici ni „lukenj“ in da so levi robovi vrstic lepo poravnani.

Po ploščicah se smemo premikati na naslednji način: začnemo v eni od ploščic prve vrstice; nato pa v vsakem koraku stopimo s trenutne ploščice na eno od tistih, ki so v naslednji vrstici tik pot trenutno ploščico in imajo z njo skupen nek del stranice (samo oglišče ni dovolj!). S premikanjem končamo, ko pridemo v zadnjo vrstico. (Glej primer na spodnji sliki.)

Primer razporeditve ploščic v pet vrstic. Puščice kažejo, kako se lahko premikamo med ploščicami, če začnemo pri četrti ploščici prve vrstice. V zadnji (peti) vrstici so dosegljive ploščice od vključno druge do vključno pete.



Napiši program, ki ugotovi, katera je najbolj leva in katera najbolj desna ploščica v zadnji vrstici, ki ju lahko pri opisanem načinu premikanja dosežemo. Predpostavi, da so ti na voljo naslednji podprogrami:

- **function** StVrstic: integer; **external**; — Vrne število vrstic.
- **function** StPloscic(StVrstice: integer): integer; **external**;
  Vrne število ploščic v dani vrstici (večje ali enako 1). Vrstice so oštevilčene od 1 do StVrstic.
- **function** SirinaPloscice(StVrstice, StPloscice: integer): integer; **external**;
  Vrne širino dane ploščice (v centimetrih). Ploščice so v vsaki vrstici oštevilčene od leve proti desni (skrajno leva ploščica ima številko 1, skrajno desna pa ima številko StPloscic(StVrstice)).
- **function** ZacetnaPloscica: integer; **external**;
  Številka ploščice (v prvi vrstici), v kateri se začne naše gibanje. To je število med vključno 1 in vključno StPloscic(1).

Predpostaviš lahko, da so ploščice razporejene tako, da bo v zadnji vrstici dosegljiva vsaj ena ploščica. Skupna širina vseh ploščic ne bo presegla največje vrednosti, ki jo še lahko hrani tip integer oz. **int**.

Še deklaracije v C/C++:

```
extern int StVrstic();
extern int StPloscic(int StVrstice);
extern int SirinaPloscice(int StVrstice, int StPloscice);
extern int ZacetnaPloscica();
```

2004.1.3 Ruleta

Rešitev: str. 18

Ruleta je igra na srečo, pri kateri se žreba številke od 0 do 36 (vržemo kroglico in pogledamo, pri kateri številki se ustavi). Igralci pred žrebanjem stavijo in če njihove stave zadenejo, se jim stavljeni znesek večkratno povrne.

Na voljo imaš naslednje funkcije, ki ti sporočajo podatke o stavah:

- **function** StStav: integer; **external**; — Vrne število stav.
- **function** AliZadene(Stava, Stevilka: integer): boolean; **external**;
  Ali dana stava zadene, če je izžrebana številka Stevilka? Stave so oštevilčene od 1 do StStav.
- **function** VisinaStave(Stava: integer): integer; **external**;
  Vrne znesek, ki ga je igralec stavil pri tej stavi.
- **function** Kolicnik(Stava: integer): integer; **external**;
  Vrne količnik, ki pove, kolikokratno se stavljeni znesek povrne, če stava zadene. V praksi so ti količniki odvisni od tega, kolikšna je verjetnost zadetka. (Na primer: če je stava taka, da zadene le pri eni številki, je ta količnik 36; če zadene, igralnica vrne igralcu znesek, ki ga je ta stavil, in mu za povrhu plača še 35-krat tolikšno vsoto. Če pa stava ne zadene, igralnica položeni denar pobere in igralec ne dobi ničesar. Pri drugačnih stavah je količnik drugačen; primer: stava, ki zadene pri vsaki lihi številki, ima količnik 2. Kakorkoli že, tebi se s tem ni treba ukvarjati, ampak predpostavi, da je funkcija Kolicnik že napisana.)

Pohlepni lastniki igralnice od tebe želijo, da **napišeš program**, ki pregleda podatke o stavah in ugotovi, katera številka mora biti izžrebana, da bo igralnica izplačala čim manj denarja. Če je več enako dobrih rešitev, je dovolj, če izpišeš samo eno od njih (katerokoli).

Še deklaracije v C/C++:

```
extern int StStav();
extern int AliZadene(int Stava, int Stevilka);
extern int VisinaStave(int Stava);
extern int Kolicnik(int Stava);
```

2004.1.4 Tekoče stopnice

Rešitev: str. 18

Končno! V Ljubljano prihaja podzemna železnica. Na nekaterih vmesnih postajah, kjer bo vstopalo in izstopalo manj ljudi, se bolj kot dvoje tekočih stopnic splača zgraditi ene same, ki bodo smer prilagajale potnikom (razen v konicah, ko bodo nekateri morali po stopnicah peš, medtem ko bodo stopnice vozile v nasprotno smer).

Prosili so nas za pomoč pri programu za krmiljenje stopnic, ki se morajo zagnati (če so prej mirovale) vedno, ko kdo stopi na enega od obeh senzorjev pred stopnicami (na vsaki strani stopnic je po en senzor, ki ti zna povedati, če kdo trenutno stoji na njem ali ne; ne vemo pa, v katero smer se tisti potnik giblje), in ustaviti, ko smo prepričani, da na stopnicah ne stoji noben potnik več (samo predstavljaj si, da bi te stopnice pripeljale do sredine, se ustavile in takoj spet obrnile v drugo smer, ti pa bi moral sredi hitenja v šolo po minuti prevažanja nazadnje še peš po navadnih stopnicah).

Pomagaj si z naslednjimi deklaracijami:

```
const CasVoznje = ...; { Toliko sekund je treba, da se pripelje potnik po stopnicah
                         od enega konca stopnišča do drugega. }
type SenzorT = (Zgoraj, Spodaj);
     SmerT = (Gor, Dol, Ustavi);
procedure PozeniStopnice(Smer: SmerT); external;
  { Ta podprogram pokliči, ko je treba pognati ali ustaviti stopnice. }
function PotnikNaSenzorju(Senzor: SenzorT): boolean; external;
  { Ta podprogram ti pove, ali na danem senzorju trenutno kdo stoji. }
```

Ti pa **napiši** naslednji **podprogram**:

```
procedure EnkratNaMilisekundo;
  { Sistem ga pokliče enkrat na milisekundo (tisočkrat na sekundo). }
```

Predpostavljaš lahko, da stopnice na začetku mirujejo. V svoji rešitvi lahko definiraš dodatne globalne spremenljivke, ki jih še potrebuješ, in jim tudi predpišeš začetne vrednosti.

Še deklaracije v C/C++:

```
const int CasVoznje = ...;
typedef enum { Zgoraj, Spodaj } SenzorT;
typedef enum { Gor, Dol, Ustavi } SmerT;
extern void PozeniStopnice(SmerT Smer);
extern bool PotnikNaSenzorju(SenzorT Senzor);

void EnkratNaMilisekundo();
```

NALOGE ZA DRUGO SKUPINO

Svoje odgovore dobro utemelji. Če pišeš izvorno kodo programa ali podprograma, jo obvezno tudi komentiraj in v nekaj stavkih z besedami opiši, na kakšni ideji temelji tvoja rešitev.

2004.2.1 Naraščajoča števila

Rešitev: str. 19

Včasih lahko pri nekaterih nalogah naletimo na podproblem, kako zamenjati števila v zaporedju, tako da imamo na koncu sama različna števila.

Opiši postopek, ki za izbran N prebere N celih števil, večjih od 0 in manjših ali enakih N (običajno niso vsa različna, ampak se nekatera števila ponavljajo). Za vsako od teh števil naj postopek izpiše najmanjše enako ali večje število, ki pa še ni bilo izpisano.

Pri tem pa izpisano število ne sme biti večje od N. Če so bila vsa števila od trenutnega vhodnega števila do N že izpisana, izpiši najmanjše število, ki je večje od 0 in še ni bilo izpisano.

Če upoštevaš vsa navodila, boš na koncu izpisal vsa števila od 1 do N, vsako natanko po enkrat. Za boljše razumevanje glej spodnja primera.

Pazi pa tudi na to, da bo tvoj postopek čim hitrejši.

Primer vhodnega zaporedja ($N = 10$):	6	6	6	6	5	3	3	1	1	1
Pripadajoče izhodno zaporedje:	6	7	8	9	5	3	4	1	2	10

Drugo število vhodnega zaporedja je 6, torej moramo na drugem mestu izhodnega števila izpisati število, večje ali enako 6. Ker smo 6 že izpisali, je najmanjše tako število 7.

Še en primer ($N = 5$):	5	5	5	5	5
Pripadajoče izhodno zaporedje:	5	1	2	3	4

Na drugem mestu bi morali izpisati število, večje ali enako 5; ker je $N = 5$, je edino tako število 5, ki pa je bilo že izpisano, zato izpišemo najmanjše še neizpisano število (v tem primeru torej 1).

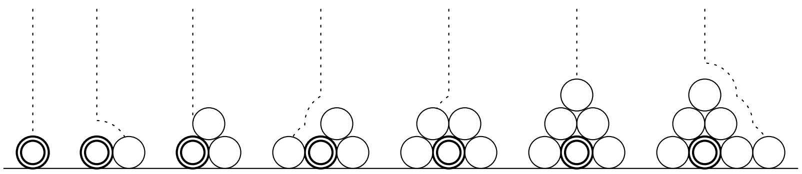
2004.2.2 Topovske krogle

Rešitev: str. 20

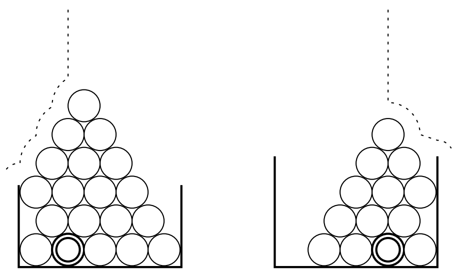
Cesarsko topništvo se je dolgo ukvarjalo s problemom, kako polniti zaboje s topovskimi kroglami, saj so želeli ustaviti polnjenje zabojev prej, preden so začele težke krogle padati čez rob zabojev. Le kdo jih bo še enkrat nalagal; ni časa ne denarja. Pametni cesarski uradniki so ugotovili, da je to odvisno od kalibra topovske krogle (premer krogle), dimenzij zaboja in položaja polnilne odprtine, zato so predpisali, da mora biti velikost zaboja, ki ga dostavijo topovske garnizije, nek večkratnik kalibra topovske krogle. To so naredili, za ostalo jim je pa zmanjkalo pameti. Na pomoč so poklicali našega velikega matematika Jurija Vego, ki jim je v kratkem času narisal kopico čudnih slik in na koncu napisal še algoritem za izračun tega, katera krogla po vrsti po padla čez rob.

Žal so čas in vojne opravile svoje in tako danes ne moremo več najti tega algoritma in te naprošamo, da ga napišeš ti — **napiši** torej **algoritem ali podprogram**, ki bo pri danih podatkih o višini, širini in položaju polnilne šobe izpisal, katera krogla po vrsti bo padla čez rob zaboja. Pri tem upoštevaj naslednje:

- Zaboj je ravno dovolj širok za s krogel in dovolj visok za v plasti krogel. Pri tem sta s in v neki celi števili (večji od 0).
- Položaj polnilne odprtine je določen s celim številom p (med vključno 1 in vključno s), ki pove, da je odprtina naravnost nad p-tim prostorom za kroglo (z drugimi besedami, ko pade prva krogla skozi odprtino v prazen zaboj, pristane tako, da je levo od nje prostora za natanko $p - 1$ krogel, desno pa za $s - p$ krogel).
- Zaboj nima tretje dimenzije oz. je v tej dimenziji dovolj velik ravno za en premer krogle. Zato lahko pri zlaganju krogel tretjo dimenzijo zanemarimo.
- Če krogla pade natančno na vrh druge in ima prostor na obeh straneh, vedno pade na desno stran. Krogle so brez inercije — nikoli se ne gibljejo vodoravno ali navzgor.



Primer, kako bi se razporedilo prvih nekaj krogel (ob predpostavki, da se ne zaletavajo v stene zaboja). Prva krogla je označena z dvojnim robom. Črtkane črtice označujejo, kako se je gibala zadnja krogla.



Črtkane črtice kažejo, kako se giblje tista krogla, ki prva pade čez rob.

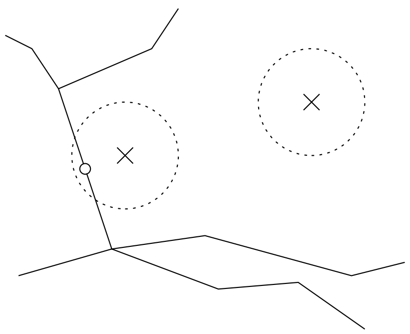
Levo je primer za $s = 5$, $p = 2$, $v = 3$; devetnajst krogel gre v zaboj, dvajseta pade čez rob.

Desno je primer za $s = 5$, $p = 4$, $v = 4$; trinajst krogel gre v zaboj, štirinajsta pade čez rob.

2004.2.3 GPS

Rešitev: str. 22

GPS je sistem lociranja v naravi z natančnostjo (radijem) okoli 16 metrov. Omejena natančnost nas lahko pri orientaciji v naravi oziroma še bolj na cesti precej moti. Natančnost se v resničnem svetu lahko spreminja v odvisnosti od vidnih satelitov in kakovosti signala. Ker hočemo natančnost umetno povečati, lahko primerjamo izračunani položaj z zemljevidom in popravimo položaj tako, da ga virtualno prestavimo na najbližjo točko na cesti oziroma poti znanega zemljevida (če seveda je kakšna cesta v radiju natančnosti).



Slika prikazuje nekaj cest in dve meritvi, označeni z $\times$. Okoli vsake meritve je črtkana krožnica, katere radij ponazarja natančnost meritve.

Pri levi meritvi je naš pravi položaj najbrž najbližja točka na daljici levo od meritve (označena s krožcem).

Pri desni meritvi pa smo predaleč od vseh poti in zato ne bi bilo pametno umetno popravljati meritve, ker bi s tem najbrž samo povečali napako.

Mi imamo na voljo nekaj podobnega; imamo funkcijo

```
function GPS(var X, Y: integer): integer;
int GPS(int* X, int* Y);
```

ki nam vrne natančnost meritve v metrih (radij), v parametrih pa vrne izmerjeni položaj v metrih od nekega izhodišča. Imamo tudi podatke o poteh in cestah: njihov potek je opisan z daljicami, za vsako daljico pa poznamo koordinate njenih dveh krajišč.

```
type Tocka = record x, y: integer end;
var Zemljevid: array [1..N, 1..2] of Tocka;

typedef struct { int x, y; } Tocka;
Tocka Zemljevid[N][2];
```

Na voljo imamo tudi funkcijo, ki zna za dano daljico (od A do B) in dano točko (recimo T) ugotoviti, katera je njej najbližja točka na daljici; poleg tega tudi vrne razdaljo med T in to najbližjo točko.

```
function NajblizjaTocka(A, B, T: Tocka; var Najblizja: Tocka): real;
float NajblizjaTocka(Tocka A, Tocka B, Tocka T, Tocka* Najblizja);
```

Opiši postopek, ki bo z uporabo teh funkcij in podatkov izboljšal natančnost izmerjenega položaja in vrnil novo izračunani položaj. Postopek naj torej prebere eno točko (izmerjeni položaj) in izpiše odgovor (ali najbližjo točko na kateri od daljic ali pa naj ugotovi, da nobena daljica ne leži dovolj blizu izmerjenega položaja).

Ker se tabela daljic ne bo pogosto spreminjala, lahko pred prvo uporabo svojega postopka podatke o daljicah tudi kako preurediš ali jih drugače organiziraš, da bo potem tvoj postopek lahko čim hitreje ugotovil, katera točka na daljicah je najbližja trenutni meritvi (po možnosti tako, da mu ne bo treba vsakič preiskati vseh daljic!). Predpostaviti smeš, da se daljice med seboj ne križajo, se pa seveda lahko stikajo.

2004.2.4 Skrajšanke

Rešitev: str. 23

Imamo skupino besed, iz katerih bi radi na vse možne načine sestavili skrajšanke. Skrajšanke (akronimi) so besede, ki jih lahko sestavimo tako, da vzamemo prvih nekaj črk vsake besede iz dane skupine besed in jih staknemo skupaj. Pri tem lahko poljubno spreminjamo vrstni red besed ter število prvih nekaj črk, ki jih bomo uporabili iz vsake besede. Vsako besedo smemo uporabiti samo enkrat.

Primer skrajšanke: RADAR = RAdio Detection And Ranging.

Napiši podprogram, ki bo iz danih N besed izpisal vse možne skrajšanke. Pri tem mora program zagotoviti:

- da bodo besede predstavljene v vseh mogočih vrstnih redih (primer za tri besede: ena, dve, tri; ena, tri, dve; dve, ena, tri; dve, tri, ena; tri, ena, dve; tri, dve, ena);
- da bo pri vsakem vrstnem redu zastopana vsaka beseda z vsemi možnimi začetki, od 0 pa vse do prvih M črk besede (primer: če je $M = 5$, moramo besedo `REGISTRACIJA` v skrajšankah predstaviti z vsemi naslednjimi začetki: „“ (prazen niz), „`R`“, „`RE`“, „`REG`“, „`REGI`“ in „`REGIS`“).

Lahko se zgodi, da ista skrajšanka nastane na več načinov (glej primer spodaj) — v tem primeru naj jo tvoj podprogram tudi po večkrat izpiše. Ne izpiše pa naj skrajšank, ki so prazni nizi.

Pomagaj si z naslednjimi deklaracijami:

```
const N = ...;
type TabelaT = array [1..N] of string;
procedure IzpisiVseSkrajsanke(M: integer; Besede: TabelaT);
```

Primer na str. 8 kaže, kaj dobimo pri skupini $N = 3$ besed, `REGISTRACIJA`, `NADZOR` in `SISTEM`, če želimo vse skrajšanke, kjer bo vsaka beseda prikazana z 0 pa do največ tremi črkami ($M = 3$).

PRAVILA TEKMOVANJA ZA TRETJO SKUPINO

Na začetku tekmovanja smo tekmovalcem najprej razdelili naslednja navodila. Nekaj minut kasneje so dobili tudi besedilo nalog, za reševanje pa so imeli približno tri ure časa.

Vsaka naloga zahteva, da napišeš program, ki prebere neke vhodne podatke, izračuna odgovor oz. rezultat ter ga izpiše v izhodno datoteko. Programi naj berejo vhodne podatke iz datoteke *imenaloge*.`in` in izpisujejo svoje rezultate v *imenaloge*.`out`. Natančni imeni datotek sta podani pri opisu vsake naloge. V vhodni datoteki je vedno po en sam testni primer. Vaše programe bomo pognali po večkrat, vsakič na drugem testnem primeru. Besedilo vsake naloge natančno določa obliko (format) vhodnih in izhodnih

(iz *registracija nadzor sistem*)	(iz *registracija sistem nadzor*)
s si sis	n na nad
n ns nsi nsis	s sn sna snad
na nas nasi nasis	si sin sina sinad
nad nads nadsi nadsis	sis sisn sisna sisnad
r rs rsi rsis	r rn rna rnad
rn rns rnsi rnsis	rs rsn rsna rsnad
rna rnas rnasi rnasis	rsi rsin rsina rsinad
rnad rnads rnadsi rnadsis	rsis rsisn rsisna rsisnad
re res resi resis	re ren rena renad
ren rens rensi rensis	res resn resna resnad
rena renas renasi renasis	resi resin resina resinad
renad renads renadsi renadsis	resis resisn resisna resisnad
reg regs regsi regsis	reg regn regna regnad
regn regns regnsi regnsis	regs regsn regsna regsnad
regna regnas regnasi regnasis	regsi regsin regsina regsinad
regnad regnads regnadsi regnadsis	regsis regsisn regsisna regsisnad
(iz *nadzor registracija sistem*)	(iz *nadzor sistem registracija*)
s si sis	r re reg
r rs rsi rsis	s sr sre sreg
re res resi resis	si sir sire sireg
reg regs regsi regsis	sis sisr sisre sisreg
. . . .	. . . .
(iz *sistem registracija nadzor*)	(iz *sistem nadzor registracija*)
n na nad	r re reg
r rn rna rnad	n nr nre nreg
re ren rena renad	na nar nare nareg
reg regn regna regnad	nad nadr nadre nadreg
. . . .	. . . .

Primer skrajšank pri nalogi 2004.2.4.

datotek. Tvoji programi lahko predpostavijo, da se naši testni primeri ujemajo s pravili za obliko vhodnih datotek, ti pa moraš zagotoviti, da se bo izpis tvojega programa ujemal s pravili za obliko izhodnih datotek.

Delovno okolje

Na začetku boš dobil mapo s svojim uporabniškim imenom ter navodili, ki jih pravkar prebiraš. Ko boš sedel pred računalnik, boš dobil nadaljna navodila za prijavo v sistem.

Na vsakem računalniku imaš na voljo enoto (disk) `U:`, na kateri lahko kreiraš svoje datoteke (datoteke, ki so tam že od prej, pusti pri miru). Programi naj bodo napisani v programskem jeziku Pascal, C ali C++, mi pa jih bomo preverili z 32-bitnimi prevajalniki FreePascal, GNU C/C++ in GCJ. Za delo lahko uporabiš `FP` oz. `ppc386` (FreePascal), `TURBO` (Turbo Pascal) `GCC/G++` (GNU C/C++ — command line compiler), `TC` (Turbo C) in Java 2 SDK.

Oglej si tudi spletno stran: `http://rtk/`, kjer boš dobil nekaj testnih primerov in program `RTK.EXE`, ki ga lahko uporabiš za preverjanje svojih rešitev. Tukaj si lahko tudi ogledaš anonimizirane rezultate ostalih tekmovalcev.

Preden boš oddal prvo rešitev, boš moral programu za preverjanje nalog sporočiti svoje ime, kar bi na primer Janez Novak storil z ukazom

```
rtk -name JNovak
```

(prva črka imena in priimek, brez presledka, brez šumnikov).

Za oddajo rešitve uporabi enega od naslednjih ukazov:

`rtk` *imenaloge*.`pas`
`rtk` *imenaloge*.`c`
`rtk` *imenaloge*.`cpp`
`rtk` *ImeNaloge*.`java`

Program `rtk` bo prenesel izvorno kodo tvojega programa na testni računalnik, kjer se bo prevedla in pognala na desetih testnih primerih. Na spletni strani boš dobil za vsak testni primer obvestilo o tem, ali je program pri njem odgovoril pravilno ali ne. Če se bo tvoj program s kakšnim testnim primerom ukvarjal več kot deset sekund, ga bomo prekinili in to šteli kot napačen odgovor pri tem testnem primeru.

Da se zmanjša možnost zapletov pri prevajanju, ti priporočamo, da ne spreminjaš privzetih nastavitev svojega prevajalnika. Tvoji programi naj uporabljajo le standardne knjižnice svojega programskega jezika in naj ne delajo z drugimi datotekami kot z vhodno in izhodno. Dovoljena je uporaba literature (papirnate), ne pa računalniško berljivih pripomočkov, prenosnih računalnikov, prenosnih telefonov itd.

Ocenjevanje

Vsaka naloga lahko prinese tekmovalcu od 0 do 100 točk. Vsak oddani program se preizkusi na desetih testnih primerih; pri vsakem od njih dobi od 0 do 10 točk (praviloma 10, če je izpisal popolnoma pravilen odgovor, sicer pa 0; izjema je 2. naloga, kjer dobijo boljše rešitve več točk kot slabše), nato pa se te točke po vseh testnih primerih seštejejo v skupno število točk tega programa. Če si oddal N programov za to nalogo in je najboljši med njimi dobil M (od 100) točk, dobiš pri tej nalogi $\max\{0, M - 3(N-1)\}$ točk. Z drugimi besedami: za vsako oddajo (razen prve) pri tej nalogi se ti odbijejo tri točke. Pri tem pa ti nobena naloga ne more prinesti negativnega števila točk. Če nisi pri nalogi oddal nobenega programa, ti ne prinese nobenih točk. Če se poslana izvorna koda ne prevede uspešno, to ne šteje kot oddaja.

Skupno število točk tekmovalca je vsota po vseh nalogah. Tekmovalce razvrstimo po skupnem številu točk.

Vsak tekmovalec se mora sam zase odločiti o tem, katerim nalogam bo posvetil svoj čas, v kakšnem vrstnem redu jih bo reševal in podobno. Verjetno je priporočljivo najprej reševati lažje naloge.

Poskusna naloga (ne šteje k tekmovanju) `poskus.in`, `poskus.out`

Napiši program, ki iz vhodne datoteke prebere eno celo število (le-to je v prvi vrstici, okoli njega ni nobenih dodatnih presledkov ipd.) in izpiše njegov desetkratnik v izhodno datoteko.

Primer vhodne datoteke:

```
123
```

Ustrezna izhodna datoteka:

```
1230
```

Primer rešitve:

```
program PoskusnaNaloga;
var T: text; i: integer;
begin
  Assign(T, 'poskus.in'); Reset(T); ReadLn(T, i); Close(T);
  Assign(T, 'poskus.out'); Rewrite(T); WriteLn(T, 10 * i); Close(T);
end.
```

```
#include <stdio.h>
int main() {
   FILE *f = fopen("poskus.in", "rt");
   int i; fscanf(f, "%d", &i); fclose(f);
   f = fopen("poskus.out", "wt"); fprintf(f, "%d\n", 10 * i);
   fclose(f); return 0;
}
```

```
#include <fstream.h>
int main() {
   ifstream ifs("poskus.in"); int i; ifs >> i;
   ofstream ofs("poskus.out"); ofs << 10 * i;
   return 0;
}
```

```
import java.io.*;
public class Poskus {
   public static void main (String[] args) {
      try {
         StreamTokenizer st = new StreamTokenizer(new FileReader("poskus.in"));
         st.nextToken(); int i = (int) st.nval;
         PrintWriter os = new PrintWriter(new FileOutputStream("poskus.out"));
         os.println(10 * i); os.close();
      } catch (Exception e) { }
   }
}
```

NALOGE ZA TRETJO SKUPINO

2004.3.1 Otoki otoki.in, otoki.out

Rešitev: str. 25

Če bi gladina svetovnih morij in oceanov narasla, bi se lahko kakšen otok popolnoma potopil in bi se tako število otokov na svetu zmanjšalo za ena. Po drugi strani pa bi lahko pri kakšnem otoku voda zalila le nižje ležeče dele, hribi pa bi še vedno štrleli ven in bi tako iz enega nastalo več samostojnih otokov; tako bi se število otokov na svetu povečalo. Podobno je, če se gladina oceanov zniža — kakšen otok lahko na novo pogleda iz vode, lahko pa se več otokov združi v enega samega, ker se plitvine med njimi izsušijo. Tako se torej, če spreminjamo gladino oceanov, število otokov zdaj povečuje, zdaj zmanjšuje. Nas pa zanima, kakšno je največje število otokov, ki bi ga lahko na svetu imeli, če bi mogli primerno izbrati gladino oceanov. (Za potrebe te naloge štejemo tudi celine kot velike otoke.)

Da bo naloga lažja, predpostavimo, da je Zemlja ploščata, ne pa okrogla, in da je njeno površje kot karirasta mreža, v kateri je višina površja znotraj vsake celice konstantna. Naj bo $v(x, y)$ višina celice na preseku vrstice y in stolpca x. Če je gladina oceanov g, si mislimo, da so vse celice (x, y), za katere je $v(x, y) < g$, potopljene pod vodo, vse ostale pa so kopne. Pri tej nalogi torej ne bomo komplicirali z jezeri, depresijami, rečnimi in jezerskimi otoki ipd. *Otok* definirajmo kot vsako tako skupino kopnih celic, ki je povezana (torej se da priti od poljubne celice otoka do poljubne druge celice otoka, pri tem pa ves čas hoditi po samih kopnih celicah in iti vedno s trenutne celice na eno od njenih štirih sosed, ki imajo z njo skupnega enega od robov) in ki ji ne moremo dodati nobene druge trenutno kopne celice, ne da bi ta pogoj prenehal veljati. (Glej primer spodaj.)

Napiši program, ki pri danih podatkih o višinah ugotovi, katero je največje možno število otokov, ki jih je mogoče dobiti s primernim izborom gladine oceanov g.

Vhodna datoteka: v prvi vrstici sta najprej števili h in w, ki povesta višino oz. širino kariraste mreže. To sta celi števili, velja $1 \leq h \leq 100$, $1 \leq w \leq 100$. Sledi h vrstic, v vsaki je po w števil, ki povedo višine celic: $v(1, y), v(2, y), \ldots, v(w, y)$. Višine so cela števila, zanje velja $0 \leq v(x, y) \leq 1\,000\,000\,000$.

Izhodna datoteka: vanjo naj tvoj program izpiše največje možno število otokov, ki ga je mogoče pri dani vhodni datoteki dobiti, če primerno izberemo gladino oceanov g.

Primer vhodne datoteke:

```
5 10
6 6 8 6 3 4 2 3 6 7
3 7 9 7 3 3 3 2 6 6
3 6 7 7 2 2 6 5 5 6
5 5 5 2 2 5 5 8 5 5
5 6 5 1 0 3 4 4 4 3
```

Tu je mogoče dobiti pet otokov (pri $g = 6$):

```
A A A A . . . . B B
. A A A . . . . B B
. A A A . . C . . B
. . . . . . . . D . .
. E . . . . . . . .
```

Pripadajoča izhodna datoteka:

```
5
```

2004.3.2 SBN   `sbn.in`, `sbn.out`

Rešitev: str. 27

Mislimo si, da imamo nek zelo preprost računalnik. Njegov procesor ima 256 registrov, ki jih označimo z `r1`, `r2`, ..., `r256`. Vsak register lahko hrani poljubno predznačeno 32-bitno celo število (kot tip integer oz. **int**). Poleg tega je v računalniku še bralni pomnilnik (ROM), v katerem je shranjeno zaporedje ukazov, ki ga bo procesor izvajal. Običajnega bralno-pisalnega pomnilnika (RAM) ali kakšnih zunanjih enot pa ta računalnik nima.

Procesor podpira eno samo inštrukcijo: „odštej in skoči, če je rezultat negativen“ (SBN — *subtract and branch if negative*). Vsak ukaz je takšne oblike:

Oznaka: SBN a, b, c, CiljnaOznaka

Procesor pri takem ukazu izračuna razliko b − c in jo shrani v a; nato pa, če je ta razlika manjša od 0, skoči na ukaz z oznako CiljnaOznaka (drugače pa se izvajanje nadaljuje pri naslednjem ukazu v zaporedju):

Oznaka: a := b − c; **if** a < 0 **then goto** CiljnaOznaka;

Operand a mora biti ime nekega registra, b in c pa sta lahko imeni registrov ali pa celoštevilski konstanti. Oznaka mora biti enolična (dva ukaza ne smeta imeti iste oznake), drugače pa je to lahko poljuben niz, sestavljen iz črk, števk in znakov „_“, ki se ne začne na števko in je dolg največ 20 znakov. Pri imenih oznak ne razlikujemo med velikimi in malimi črkami. Oznaka pred ukazom (z dvopičjem vred) lahko tudi manjka, le da se potem nanj pač ne bo dalo skakati. Tudi operand CiljnaOznaka (skupaj z vejico pred njim) lahko manjka; v tem primeru se bo izvajanje nadaljevalo pri naslednjem ukazu v zaporedju, ne glede na to, ali je bil rezultat odštevanja negativen ali ne.

Procesor začne z izvajanjem pri prvem ukazu v zaporedju, ustavi pa se, če je ravnokar izvedel zadnji ukaz v zaporedju in mu po njem ni bilo treba izvesti skoka (ker rezultat odštevanja ni bil manjši od 0 ali pa ker manjka četrti operand s ciljno oznako).

Izkaže se, da lahko z inštrukcijo SBN emuliramo več ali manj vse aritmetične in logične operacije, kakršne podpirajo običajni procesorji.

Napiši program, ki iz *vhodne datoteke* prebere neko naravno število n (zanj velja $2 \leq n \leq 100$) in v *izhodno datoteko* izpiše zaporedje ukazov SBN, ki uredi vrednosti v registrih `r1`, `r2`, ..., `r`n. Ne glede na to, kakšne so začetne vrednosti teh registrov, morajo biti ob koncu izvajanja tvojega zaporedja ukazov v teh registrih enake vrednosti, le prerazporejene tako, da velja `r1` $\leq$ `r2` $\leq \ldots \leq$ `r`n; končne vrednosti ostalih registrov so lahko poljubne. Zaporedje ukazov sme biti dolgo največ 100 000 ukazov in se ob izvajanju ne glede na začetne vrednosti registrov ne sme izvajati več kot 1 000 000 korakov. Začetne vrednosti registrov `r1`, ..., `r`n so med vključno $-1\,000\,000$ in vključno $+1\,000\,000$, začetna vrednost registrov `r`$(n+1)$, ..., `r256` pa je 2004.

Namig: zaporedje n vrednosti lahko urediš na primer tako, da najprej poiščeš najmanjšo med njimi in jo postaviš na prvo mesto; nato poiščeš najmanjšo med preostalimi in jo postaviš na drugo mesto; in tako naprej.

Točkovanje: pri vsakem testnem primeru lahko dobiš od 0 do 10 točk. Če zaporedje inštrukcij, ki ga tvoj program izpiše, ne ustreza zgornjim zahtevam in omejitvam, dobiš 0 točk, drugače pa je število točk odvisno od števila inštrukcij v tvojem zaporedju (recimo mu L):

Če je...	...dobiš toliko točk:
$L \leq 3n$	10
$3n < L \leq 6n$	9
$6n < L \leq n^2/2$	8
$n^2/2 < L \leq 3n^2/2$	7
$3n^2/2 < L \leq 3n^2$	6
$3n^2 < L$	5

Spodaj je **primer** zaporedja ukazov, ki rešuje malo drugačen problem: v registru `r6` izračuna povprečje vrednosti registrov `r1`, ..., `r5`, zaokroženo na najbližje celo število (pri tem predpostavi, da njihove vrednosti niso negativne in tudi ne prevelike).

```
               SBN r7, 0, r1
               SBN r7, r7, r2
               SBN r7, r7, r3
               SBN r7, r7, r4
               SBN r7, r7, r5
               SBN r6, r6, r6
               SBN r7, 0, r7
ZacetekZanke:  SBN r7, r7, 5, KonecZanke
               SBN r6, r6, 1, ZacetekZanke
KonecZanke:    SBN r7, r7, −2, ZaokroziDol
               SBN r6, r6, 1
ZaokroziDol:   SBN r6, 0, r6
```

2004.3.3 Kako so Butalci kupovali pamet `pamet.in`, `pamet.out`

Rešitev: str. 31

Ko so se Butalci odpravili v Tuje mesto po pamet, so najprej tja poslali butalskega policaja, da bi jim poročal o cenah in bi tako lahko izbrali najugodnejšo ponudbo. Brez tvoje pomoči pa seveda ne bo šlo.

Jim znaš **napisati program**, ki ugotovi najmanjši znesek, za katerega lahko kupijo N kosov pameti, če za vsakega od M trgovcev v mestu veš, koliko kosov pameti ima na voljo in koliko vsak kos pameti stane?

Vhodna datoteka. V prvi vrstici sta dve števili: N ($1 \leq N \leq 25$ — število Butalcev, ki potrebujejo novo pamet) in M ($1 \leq M \leq 500\,000$ — število trgovcev), v vrsticah 2 do $M+1$ pa najprej število kosov pameti K_i, ki jih ima na voljo i-ti trgovec, čemur sledi K_i nenegativnih celih števil $c_{i,j}$ s ceno pameti za vsak naslednji kupljen kos (torej: če pri njem kupiš en kos, plačaš $c_{i,1}$; če kupiš dva kosa, plačaš $c_{i,1} + c_{i,2}$ in tako naprej). Skupna vsota cen v vhodni datoteki ne presega ene milijarde, pameti pa bo vedno za vse dovolj. Skupno število kosov pameti, ki so trenutno naprodaj (torej $N_1 + N_2 + \ldots + N_M$) je največ 1 000 000.

(*Primer:* vrstica „`5 7 3 4 0 1`“ pomeni, da je pri tem trgovcu naprodaj pet kosov pameti, pri čemer en kos velja 7 tolarjev, dva kosa 10 tolarjev, trije ali štirje kosi 14 tolarjev in pet kosov 15 tolarjev.)

V *izhodno datoteko* izpiši najnižjo možno ceno, za katero je mogoče dobiti N kosov pameti.

Primer vhodne datoteke:

```
4 5
2 10 1
4 1 0 10 10
2 5 4
2 6 2
6 11 11 11 11 0 0
```

Pripadajoča izhodna datoteka:

```
9
```

Tu bi dva kosa pameti kupili pri drugem in dva pri četrtem trgovcu.

2004.3.4 Izomorfizem `izomorf.in`, `izomorf.out`

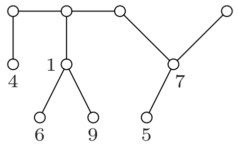
Rešitev: str. 33

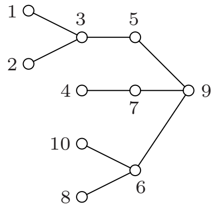
Graf je sestavljen iz množice *vozlišč* in množice *povezav*. Na spodnji sliki so vozlišča predstavljena s krožci, povezave pa s črtami med njimi. Vsaka povezava povezuje dve vozlišči.

Pri tej nalogi bomo imeli opravka z grafi, za katere veljajo naslednje lastnosti:

- iz vsakega vozlišča se da po povezavah priti do vsakega drugega;
- povezave ne tvorijo ciklov (na primer a–b, b–c in c–a);
- nobeno vozlišče ne nastopa v več kot treh povezavah.

Napiši program, ki prebere opisa dveh grafov s po n vozlišči in $n-1$ povezavami. Vozlišča so pri vsakem grafu oštevilčena od 1 do n. Pravimo, da sta grafa *izomorfna*, če lahko enega pretvorimo v drugega zgolj s preimenovanjem vozlišč. Tvoj program naj ugotovi, ali sta dana dva grafa izomorfna ali ne.





Ta dva grafa sta izomorfna — če primerno preimenujemo vozlišča prvega grafa in jih malo premaknemo po papirju (seveda ne da bi prekinili kakšno povezavo ali dodali kakšno novo ali kaj podobnega), lahko dobimo drugi graf.

Vhodna datoteka: v prvi vrstici je celo število n ($1 \leq n \leq 1000$), ki pove, koliko vozlišč imata grafa, ki ju bo treba primerjati. Temu sledi $n-1$ vrstic, ki navajajo povezave prvega grafa, in še $n-1$ vrstic, ki navajajo povezave drugega grafa. Vsaka od teh vrstic vsebuje dve števili, ločeni s presledkom — to sta številki vozlišč, ki ju povezuje ena od povezav.

Izhodna datoteka: če grafa iz vhodne datoteke nista izomorfna, naj tvoj program v izhodno datoteko ne izpiše ničesar (ustvariti pa to datoteko vendarle mora). Če pa grafa sta izomofna, pomeni, da je mogoče vozlišča prvega grafa tako preštevilčiti, da dobimo drugi graf. V tem primeru naj tvoj program izpiše n vrstic, ki opisujejo eno takšno preštevilčenje. V vsaki vrstici naj bosta dve števili, recimo u in v (ločeni s presledkom), ki povesta, da moramo vozlišče u prvega grafa preimenovati v v.

Primer vhodne datoteke za grafa z gornje slike:

```
10
2 4
2 10
10 3
7 8
5 7
7 3
10 1
6 1
1 9
1 3
3 5
4 7
3 2
7 9
9 6
5 9
10 6
6 8
```

Ena od možnih pripadajočih izhodnih datotek:

```
8 1
5 2
7 3
3 5
10 9
4 4
2 7
1 6
6 10
9 8
```

2004.3.5 Čebelica Maja gre vasovat `maja.in`, `maja.out`

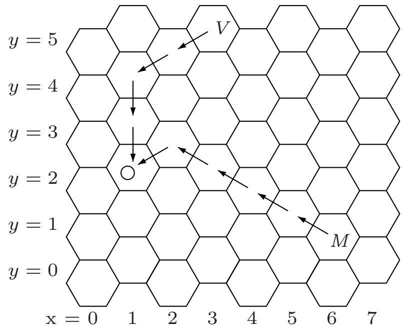
Rešitev: str. 36

Vili in čebelica Maja živita v čebeljem panju. To je satovje, ki ga sestavljajo sobice šesterokotne oblike. Ponoči čebele ne smejo leteti, zato se Maja na skrivaj odplazi vasovat k Viliju. Pri tem se plazi iz sobice v sobico, kar je zamudno, pa tudi tvegano opravilo, saj jo pri tem lahko zalotijo špeckahle. Maja pozna nekaj varnih mest v panju in jih predlaga Viliju, žal pa je ta čebelje pameti in se ne more odločiti, katerega od njih bi izbral, saj še tega ne zna prav izračunati, kako daleč je do posameznega mesta.

Podana imaš položaja Maje in Vilija ter treh predlaganih sobic. **Napiši program**, ki za vsako od njih ugotovi, kako daleč ima do te sobice Maja in kako daleč Vili. Dolžina poti se meri v številu sobic, ki jih pot vsebuje, pri tem pa se začetna sobica ne šteje. Na primer: najkrajša pot med dvema sosednjima sobicama (takima, ki imata skupno stranico) je dolga 1.

Podatki bodo podani v koordinatnem sistemu čebeljega panja, ki je predstavljen na spodnji skici. Pri 8 od 10 testnih primerov bodo vse koordinate med vključno 0 in vključno 500, pri ostalih dveh pa med 0 in 10000. Panj je dovolj velik in nima kakšnih lukenj ali manjkajočih sobic, tako da lahko predpostaviš, da vsak par celih števil (x, y) res predstavlja koordinati neke sobice.

Vhodna datoteka. V prvi vrstici imaš koordinate Maje, v drugi pa Vilija. V naslednjih treh vrsticah so koordinate še treh drugih sobic v satovju.



Primer satovja. Če bi se hotela Maja in Vili sestati v sobici, označeni s krožcem, bi imela Maja do tja po najkrajši poti pet korakov, Vili pa štiri.

V *izhodno datoteko* izpiši tri vrstice, za vsako od predlaganih treh sobic po eno. V vsaki vrstici naj bosta dve števili, ločeni s presledkom: dolžina najkrajše poti od Maje do trenutne sobice in še dolžina najkrajše poti od Vilija do trenutne sobice.

Primer vhodne datoteke:

```
6 1
3 5
1 2
6 3
7 0
```

Pripadajoča izhodna datoteka:

```
5 4
2 4
1 7
```

LETO 2004, TEKMOVANJE V POZNAVANJU UNIXA

2004.U.1 Z leti rabe računalnika ste ugotovili, da je uporabniški vmesnik s tipkovnico za normalno mislečega uporabnika neprimeren, saj se uporabnik ves čas moti pri tipkanju.

Rešitev: str. 38

Najpogostejša napaka pri tem je zamenjava sosednjih črk v besedi, seveda poleg napak neknjižne rabe besed.

Da bi uporabniku pri tem pomagali, sestavite program, ki bo za vneseno besedo izpisal vse možnosti zamenjave dveh sosednjih črk, recimo za *bal*: *abl*, *bal*, *bla*.[1]

Program naj besede prebere iz vhodne datoteke, ki je navedena kot parameter v ukazni vrstici. V vsaki vrstici datoteke je podana ena beseda. Datoteka ne

[1]Kot vidimo iz tega primera, je dovoljena tudi možnost, da sploh ne izvedemo nobene zamenjave in pustimo prvotni niz pri miru. Če je mogoče na več načinov priti do istega niza (npr. če se v prvotni besedi pojavita skupaj dve enaki črki), tak niz tolikokrat tudi izpiši.

vsebuje presledkov ali nečrkovnih znakov, le črke in znake za konec vrstice. Izpisane besede naj bodo urejene bo abecednem vrstnem redu. Morebitne prazne vrstice v vhodni datoteki preskoči.

Rešitev: str. 39

2004.U.2

Preštejte število vseh internetnih sej, ki so zabeležene v dnevniški datoteki strežnika Apache. Privzamete lahko, da je v dnevniški datoteki na prvih mestih podano število IP računalnika, ki je sodeloval pri seji. Sledi poljubno število presledkov, nato čas dostopa, zapisan kot celo število po dogovoru standard epoch time. Ta meri število pretečenih sekund od 1. januarja 1970. Časi v datoteki le naraščajo. Kot eno sejo obravnavajte vse dostope z nekega računalnika, med katerimi ni več kot 30 minut premora. Ime datoteke naj bo dano vašemu programu kot parameter v ukazni vrstici.

Dnevniška datoteka je videti takole:

```
123.123.123.123 100000
123.123.123.123   100001
123.123.123.123 100001
192.168.0.0     100002
192.168.0.1 100000000
123.123.123.123     10000000
```

Rešitev: str. 39

2004.U.3

Uredite vrstice vhodne datoteke v obratnem vrstnem redu in rezultat shranite v izhodno datoteko.

Vaša skripta naj sprejme dva parametra, prvi parameter je ime vhodne datoteke, drugi pa ime izhodne datoteke, v katero shranite rezultat.

Pričakujte, da vrstice v podani vhodni datoteki vsebujejo le črke slovenske abecede, in to brez šumnikov in presledkov. Če je več enakih vrstic, izpišite le eno. Vrstice v vhodni datoteki se vedno začno s črko.

Denimo, da zaradi težav v sistemu pri tem ne morete uporabiti programov `sort`, `perl`, `sed`, `awk`, `python`, `php` in morate najti nadomestno rešitev.

Obratni vrstni red pomeni obratno abecedno urejanje (sortiranje). Na primer: vrstice, v katerih so le znaki *A, B, V, Z, a, b, v, z*, se tako izpišejo kot *z, v, b, a, Z, V, B, A*.

Rešitev: str. 40

2004.U.4

Napišite skripto, ki bo pripravila sliko za spletno galerijo. Skripta naj sprejme kot prvi parameter ime datoteke z vhodno sliko, kot drugi parameter ime izhodne slike, kot tretji parameter njeno širino (velikost x), četrti parameter pa je največja dovoljena velikost slike.

Sintaksa:

pomanjsaj ⟨*ime vhodne slike*⟩ ⟨*ime izhodne slike*⟩
⟨*širina izhodne slike*⟩ ⟨*največja velikost izhodne slike v bajtih*⟩

Skripta mora vhodno sliko pomanjšati na določeno širino. Pri tem mora ohraniti velikostno razmerje slike. Slika ne sme biti večja od predpisane velikosti, manjša pa je lahko, vendar (če je le mogoče) ne za več kot 5 %. Sliko nato zapišite v izbrano izhodno datoteko. Privzamete lahko, da slike s tem programom vselej pomanjšujete, ne povečujete. Predpostavite lahko tudi, da vaša skripta dobi za parametre smiselne vrednosti, ki ji ne bodo zastavljale nerešljivega problema (npr. zahtevale slike velikosti 10000×10000, obenem pa omejile velikost datoteke na 1 bajt).

Namig: Pri reševanju si pomagajte z ukazom `convert`.

REŠITVE NALOG ZA PRVO SKUPINO

R2004.1.1 SMS

V konstanti Tabela hranimo podatke o tem, katero tipko se uporablja pri posamezni črki. Vhodne podatke beremo znak po znak in pri vsakem pogledamo, če pripada isti tipki kot prejšnji znak (spremenljivka PrejTipka).

Naloga: str. 1

```
program SMS;
            { abcdefghijklmnopqrstuvwxyz }
const Tabela = '22233344455566677778889999';
var Znak, Tipka, PrejTipka: char;
   StCakanj: integer;
begin
  PrejTipka := 'x'; StCakanj := 0;
  while not Eoln do begin
    Read(Znak);
    if Znak in ['a'..'z'] then { Na kateri tipki je ta znak? }
      Tipka := Tabela[Ord(Znak) − Ord('a') + 1]
    else Tipka := '1'; { Najbrž je presledek. }
    if Tipka = PrejTipka then StCakanj := StCakanj + 1;
    PrejTipka := Tipka;
  end; {while}
  WriteLn(StCakanj);
end. {SMS}
```

Pomagamo si lahko s standardno funkcijo Ord(Znak), ki vrne številsko kodo znaka Znak. Črkam `a`, ..., `z` pripadajo zaporedne številske kode, zato nam razlika Ord(Znak) − Ord('a') + 1 pove položaj Znaka v abecedi in jo lahko uporabimo kot indeks v tabelo Tabela.

R2004.1.2 Ploščice

Hranili bomo podatke o najbolj levi in najbolj desni x-koordinati, ki je še dosegljiva v trenutni vrstici (xOd, xDo), in o indeksu najbolj leve in najbolj desne dosegljive ploščice (iOd, iDo). Pri prvi vrstici je iOd = iDo = ZacetnaPloscica, x-koordinate pa dobimo s seštevanjem širin ploščic od levega roba do začetne ploščice.

Naloga: str. 1

Pri vsaki naslednji vrstici si zapomnimo, katere x-koordinate so dosegljive v prejšnji vrstici (xOdPrej, xDoPrej) in na podlagi tega izračunajmo vrednosti xOd, xDo, iOd, iDo. Vrednosti iOd in xOd moramo nastaviti pri prvi (najbolj levi) ploščici, ki je dosegljiva v tej vrstici (prepoznamo jo po tem, da je iOd takrat še 0, ker smo na začetku vrstice postavili na 0). Vrednosti iDo in xDo pa nastavimo pri vsaki dosegljivi ploščici, tako da bo na koncu obveljala najbolj desna ploščica, kar si pri teh dveh spremenljivkah tudi želimo.

```
program Ploscice;
var xOd, xDo, xOdPrej, xDoPrej, x, xx, y, i, iOd, iDo: integer;
begin
```

```
   x := 0;
   for i := 1 to ZacetnaPloscica − 1 do x := x + SirinaPloscice(1, i);
   xOd := x; xDo := xOd + SirinaPloscice(1, ZacetnaPloscica);
   iOd := ZacetnaPloscica; iDo := ZacetnaPloscica;

   for y := 2 to StVrstic do begin
      xOdPrej := xOd; xDoPrej := xDo; iOd := 0; x := 0;

      for i := 1 to StPloscic(y) do begin
         xx := x + SirinaPloscice(y, i);
         { Trenutna ploščica pokriva x-koordinate od x do xx. }
         if (x < xDoPrej) and (xx > xOdPrej) then begin
            iDo := i; xDo := xx;
            if iOd = 0 then begin iOd := i; xOd := x end;
         end; {if}
         x := xx;
      end; {for i}
   end; {for y}

   WriteLn('V zadnji vrstici so dosegljive ploščice ', iOd, '..', iDo, '.');
end. {Ploscice}
```

R2004.1.3 Ruleta

Naloga: str. 2

Pri vsaki številki se sprehodimo po vseh stavah in računajmo skupni znesek, ki bi ga morala igralnica izplačati, če bi bila izžrebana ta številka. V spremenljivki NajStevilka hranimo številko, ki zahteva najmanjši znesek (ta znesek pa hranimo v NajZnesek). Ko izračunamo znesek za naslednjo številko, ga primerjamo s spremenljivko NajZnesek in obdržimo manjši znesek.

```
program Pohlep;
var NajStevilka, NajZnesek, i, Znesek, Stava: integer;
begin
   NajZnesek := 0;
   for i := 0 to 36 do begin
      Znesek := 0;
      for Stava := 1 to StStav do
         if AliZadene(Stava, i) then
            Znesek := Znesek + VisinaStave(Stava) * Kolicnik(Stava);
      if (i = 0) or (Znesek < NajZnesek) then
         begin NajStevilka := i; NajZnesek := Znesek end;
   end; {for i}
   WriteLn('Izžrebana naj bo številka ', NajStevilka, '.');
end. {Pohlep}
```

R2004.1.4 Tekoče stopnice

Naloga: str. 3

Naš podprogram, ki se kliče vsako milisekundo, najprej pogleda, če se stopnice premikajo (TrenutnaSmer); če se, zmanjšuje vrednost števca KakoDolgoSe, ki pove, kako dolgo se morajo še premikati. Če v tem času kdo stopi na pravi senzor, pa vrednost KakoDolgoSe spet povečamo in tako zagotovimo, da bodo stopnice še dovolj časa tekle v tej smeri. Če pade števec KakoDolgoSe na 0, pa stopnice ustavimo.

Ko stopnice mirujejo, naš podprogram gleda, če kdo stoji na kakšnem od senzorjev; če pride do tega, požene stopnice v pravi smeri. Če stojijo ljudje na obeh senzorjih, se je malo težje odločiti, v katero smer bi pognali stopnice; mogoče bi bilo nepošteno, če bi se odločili vedno za isto smer, zato si raje zapomnimo zadnjo smer, v katero so se stopnice premikale, in jih poženimo v nasprotno smer. V praksi je tako ali tako malo verjetno, da bi stopila dva človeka skoraj hkrati (znotraj iste milisekunde) na senzorja vsak na svoji strani stopnišča.

```
var TrenutnaSmer: SmerT value Ustavi;
   ZadnjaSmer: SmerT value Ustavi;
   KakoDolgoSe: integer value 0;

procedure EnkratNaMilisekundo;
begin
  if TrenutnaSmer <> Ustavi then begin
    KakoDolgoSe := KakoDolgoSe − 1;

    if ((TrenutnaSmer = Gor) and PotnikNaSenzorju(Spodaj))
    or ((TrenutnaSmer = Dol) and PotnikNaSenzorju(Zgoraj))
    then KakoDolgoSe := CasVoznje * 1000;

    if KakoDolgoSe <= 0 then
      begin PozeniStopnice(Ustavi); TrenutnaSmer := Ustavi end;

  end
  else begin
    if PotnikNaSenzorju(Spodaj) then
      if PotnikNaSenzorju(Zgoraj) then
        if ZadnjaSmer = Gor then TrenutnaSmer := Dol
        else TrenutnaSmer := Gor
      else TrenutnaSmer := Gor
    else if PotnikNaSenzorju(Zgoraj) then
      TrenutnaSmer := Dol;

    if TrenutnaSmer <> Ustavi then begin
      KakoDolgoSe := CasVoznje * 1000;
      ZadnjaSmer := TrenutnaSmer;
      PozeniStopnice(TrenutnaSmer);
    end; {if}

  end; {if}
end; {EnkratNaMilisekundo}
```

REŠITVE NALOG ZA DRUGO SKUPINO

R2004.2.1 Naraščajoča števila

Naloga: str. 4

Pri tej nalogi je koristno vzdrževati množico še neizpisanih števil, recimo ji M. Ko preberemo naslednje število vhodnega zaporedja (recimo mu b), moramo izpisati najmanjše še neizpisano število, ki je večje od b, če pa takega ni, pa najmanjše neizpisano število sploh. Ko to število izpišemo, ga moramo seveda odstraniti iz M.

Vhod: $a[1..n]$

$M := \{1, \ldots, n\}$;

for $i := 1$ to n do begin
  $x :=$ najmanjše tako število iz M, ki je večje ali enako $a[i]$;
    če so vsa manjša od $a[i]$, vzemi najmanjše število iz M;
  izpiši x;
  $M := M - \{x\}$;
end;

V praksi lahko množico M izvedemo na različne načine. Zelo preprost, čeprav ne najbolj učinkovit način bi bila tabela booleanov:

```
var A: array [1..N] of integer;              { Vhodno zaporedje. }
    Neizpisano: array [1..N] of boolean;     { Še neizpisana števila. }
    i, x: integer;
begin
  for x := 1 to N do Neizpisano[x] := true;
  for i := 1 to N do begin
    x := A[i];
    while not Neizpisano[x] do
      begin x := x + 1; if x > N then x := 1 end;
    WriteLn(x);
    Neizpisano[x] := false;
  end; {for}
end.
```

Slabost te rešitve je, da lahko porabimo veliko časa za pregledovanje tabele v zanki while. Časovna zahtevnost takega postopka je v najslabšem primeru $O(n^2)$ (npr. če so vsi elementi vhodnega zaporedja enaki).

Učinkovitejšo rešitev dobimo, če hranimo neizpisane elemente v kakšni od drevesastih podatkovnih struktur (na primer AVL-drevo, rdeče-črno drevo ipd.), ki podpirajo dodajanje, brisanje in iskanje najmanjšega elementa z določenega intervala ter za vsako od teh operacij porabijo le $O(\log n)$ časa. Časovna zahtevnost našega postopka bi bila tako le $O(n \log n)$.

Namesto drevesa bi lahko tudi ostali pri tabeli Neizpisano iz gornjega programa, le da bi jo dopolnili še z več manjšimi tabelami, v katerih bi vsak element predstavljal po dve, štiri, osem, šestnajst itd. zaporednih števil in bi imel vrednost true le, če so neizpisana vsa tista števila. Potem se nam ne bi bilo treba ves čas sprehajati po tabeli Neizpisano in povečevati x za 1, ampak bi lahko uporabili še te dodatne tabele in delali z x-om večje skoke (po dva, štiri, osem itd. elementov naenkrat). Tudi tako bi prišli do časovne zahtevnosti $O(n \log n)$.

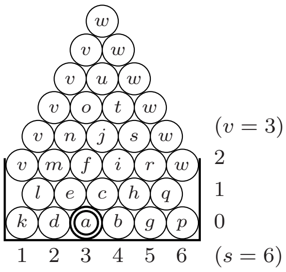
R2004.2.2 Topovske krogle

Naloga: str. 4

Oglejmo si primer na spodnji sliki:

Vidimo, da gredo L-skupine lahko do največ L_k za $k = p - 1 + (v - 1)/2$, D-skupine pa do D_k za $k = s - p + (v - 1)/2$. (Če je v sod, vzamemo $v/2 - 1$ namesto $(v - 1)/2$.) Če bi na primer v nekem konkretnem primeru ti dve formuli dovolili L-skupine do L_5 in D-skupine do D_3, bi to pomenilo, da dejansko nastopijo skupine do L_3 (za slednjo bi morala priti D_4, ki pa je nemogoča, zato tudi do L_4 in L_5 ne bomo prišli.)

Skupine $L_0, ..., L_{p-1}$ so polne in prispevajo $1, 3, ..., 2p - 1$ krogel. Vse nadaljnje L-skupine so okrnjene (zaradi levega roba zaboja) in prispevajo po $2p - 1$ krogel.

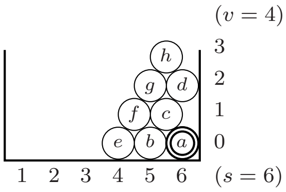


Primer za $s = 6$, $v = 3$, $p = 3$. Krogle so označene s črkami od a naprej in to v takem vrstnem redu, v kakršnem padajo v zaboj. Lahko jih združimo v „leve“ in „desne“ skupine:

$L_0 = a,\ L_1 = def,\ L_2 = klmno,\ L_3 = vvvvv,$
$D_1 = bc,\ D_2 = ghij,\ D_3 = pqrstu,\ D_4 = wwwwww.$

Vrstni red padanja je $L_0, D_1, L_1, D_2, L_2, D_3, L_3, D_4$.

Podobno so $D_1, ..., D_{s-p}$ polne in prispevajo $2, 4, ..., 2(s - p)$ krogel. Ostale D-skupine prispevajo po $2(s - p)$ krogel.



Poseben primer je, če je v sod in $p = s$:

$L_0 = a,\ D_1 = \text{(prazna)},\ L_1 = bcd,$
$D_2 = \text{(prazna)}\ L_2 = efgh.$

Na tej sliki je primer za $v = 4$, $p = s = 6$.

Prej opisani razmislek bi rekel, da nastopijo tu D-skupine do D_k za $k = s - p + v/2 - 1$, v našem primeru je to do D_1, v resnici pa moramo tu vzeti še eno D-skupino več, sicer bi potem neupravičeno zavrgli skupino L_2. Moramo pa pri tej zadnji L-skupini upoštevati, da ni čisto polna — za kroglami *efgh* bi morala v tej skupini nastopiti še ena krogla, vendar ta že pade čez rob.

Ostal je še en poseben primer — $s = 1$, ko se krogle nalagajo neposredno druga nad drugo in sosednje plasti niso zamaknjene za pol širine krogle kot pri večjih s. Pri $s = 1$ gre v zaboj preprosto v krogel.

Zdaj imamo vse potrebno, da lahko število krogel kar izračunamo:

```
function StKrogel(p, s, v: integer): integer;
var MaxKL, MaxKD, k, N: integer;
begin
  if s = 1 then begin StKrogel := v + 1; exit end;

  MaxKL := p − 1 + (v − 1) div 2;
  MaxKD := (s − p) + (v − 1) div 2;
  if (v mod 2 = 0) and (p = s) then MaxKD := MaxKD + 1;

  if MaxKL > MaxKD then MaxKL := MaxKD
  else if MaxKL < MaxKD then MaxKD := MaxKL + 1;

  N := 0;
  for k := 0 to Min(p − 1, MaxKL) do N := N + 2 * k + 1;
  if MaxKL >= p then N := N + (MaxKL − p + 1) * (2 * p − 1);
  if (v mod 2 = 0) and (p = s) then
    N := N − 1; { L_MaxKL je nepopolna — zadnja krogla se zvali čez rob. }

  for k := 1 to Min(s − p, MaxKD) do N := N + 2 * k;
  if MaxKD > s − p then N := N + (MaxKD − (s − p)) * 2 * (s − p);
  StKrogel := N + 1; { + 1 zato, ker naloga sprašuje, katera prva pade čez rob }
end; {StKrogel}
```

Malo preprostejša, vendar tudi manj učinkovita rešitev pa je simulacija — v neki tabeli vodimo za vsako x-koordinato (in še za polovične koordinate vmes) podatek o tem, kako visoko so nagrmadene krogle pri tem x. Potem lahko simuliramo, kako bi padale posamezne krogle, in vidimo, kdaj bi prva padla čez rob.

```
function StKrogel2(p, s, v: integer): integer;
const MaxS = ...;
var Visina: array [0..2 * MaxS] of integer;
    x, A, B, BB, C, N: integer;
begin
  for x := 0 to 2 * s do if Odd(x) then Visina[x] := −1 else Visina[x] := 0;
  N := 0; { N je števec krogel. }
  while true do begin
    x := 2 * p − 1; { Začetni položaj krogle pri padanju. }
    while (x > 0) and (x < 2 * s) do begin
      { B naj bo višina krogel na položaju x, A in C pa na sosednjih dveh položajih.
        BB naj bo višina krogel na položaju x, če bi trenutna krogla le še padla dol
        pri x in se ne bi več premikala levo ali desno. }
      A := Visina[x − 1]; B := Visina[x]; C := Visina[x + 1];
      BB := B; if A > BB then BB := A; if C > BB then BB := C; BB := BB + 1;
      { Premik na desno je mogoč, če je B > C in se krogla ne bi odbila od desne
        stene. Če tak premik ni mogoč, na podoben način razmislimo o razmiku v
        levo; če tudi to ne gre, se krogla ustavi. }
      if (B > C) and not ((x = 2 * s − 1) and (BB <= v)) then x := x + 1
      else if (B > A) and not ((x = 1) and (BB <= v)) then x := x − 1
      else break;
    end; {while}
    if (x <= 0) or (x >= 2 * s) then break; { Krogla je padla čez rob. }
    Visina[x] := BB; N := N + 1;
  end; {while}
  StKrogel2 := N;
end; {StKrogel2}
```

Spodnja tabela kaže, koliko krogel gre v nekatere majhne zaboje, preden pade prva čez rob.

	$s = 5$				
v	$p = 1$	2	3	4	5
6	15	28	33	22	15
5	15	28	33	22	9
4	8	19	24	13	8
3	8	19	24	13	4
2	3	10	15	6	3
1	3	10	15	6	1

	$s = 6$					
v	$p = 1$	2	3	4	5	6
6	15	30	43	37	22	15
5	15	30	43	37	22	9
4	8	19	32	26	13	8
3	8	19	32	26	13	4
2	3	10	21	15	6	3
1	3	10	21	15	6	1

R2004.2.3 GPS

Naloga: str. 6

Ko dobimo od GPSa novo meritev in njeno natančnost, nas bo zanimalo, katera točka na daljicah je najbližja naši meritvi, pri tem pa še leži znotraj kroga, ki ima središče v izmerjenem položaju in čigar radij je ravno natančnost meritve.

Razdelimo ravnino na neke manjše celice, na primer s karirasto mrežo. Za vsako daljico lahko pogledamo, v katerih celicah vsaj deloma leži; enako storimo tudi za naš poizvedovalni krog. Za naš problem so zdaj zanimive le tiste daljice, ki ležijo vsaj deloma v kakšni od celic, v kateri vsaj deloma leži tudi naš krog. (Ta kriterij še ne zagotavlja, da ne bomo kakšne daljice pregledali po nepotrebnem, zagotavlja pa, da ne bomo spregledali nobene take, ki vsaj deloma leži tudi v poizvedovalnem krogu.) Če si torej na začetku za vsako celico pripravimo seznam daljic, ki ležijo vsaj deloma v njej, moramo potem pri odgovarjanju na poizvedbo pregledati le sezname za tiste celice, ki se vsaj deloma prekrivajo z našim poizvedovalnim krogom.

Velikost celic moramo primerno izbrati; če bo premajhna, bo ležala vsaka daljica v veliko celicah in bodo zato seznami daljic za posamezno celico zasedli preveč prostora; poleg tega bo tudi poizvedovalni krog v tem primeru pripadal veliko celicam in bo treba pri vsaki poizvedbi pregledati veliko takih seznamov (to je nerodno predvsem, če jih nimamo vseh v pomnilniku, ampak bi morali za vsak seznam narediti en dostop do diska, da bi ga sploh prebrali). Če pa bodo celice prevelike, bo v vsaki od njih veliko daljic in bomo zato morali pri posamezni poizvedbi pregledati preveč daljic, kar nam bo požrlo preveč časa.

Še en koristen prijem, s katerim lahko poskusimo zmanjšati število daljic, ki jih bo treba pregledati, je ta, da prej pregledamo tiste celice, ki so bližje središču kroga. Če poznamo kakšno daljico, ki je od središča oddaljena d enot, potem nima smisla obiskovati celic, pri katerih je najbližja točka celice oddaljena od središča kroga za d ali več enot, saj v takih celicah gotovo ne bomo našli nobene boljše daljice od tiste, ki jo že poznamo.

Daljice lahko poindeksiramo tudi kako drugače, na primer tako, da ravnino z eno vodoravno in eno navpično premico razsekamo na štiri dele in to rekurzivno ponavljamo, ustavimo pa se, ko v posameznem delu ostane dovolj malo daljic ali pa ko posamezni del postane že premajhen. Na ta način dobimo drevesasto strukturo, ki se imenuje štiriško drevo (*quad-tree*). Obstaja še veliko drugih drevesastih podatkovnih struktur, namenjenih indeksiranju prostorskih podatkov, na primer R-drevesa in njihove številne različice.[2]

R2004.2.4 Skrajšanke

Uporabimo rekurzijo — na vsakem koraku poskusimo na vse možne načine izbrati naslednjo besedo (izmed tistih, ki jih doslej še nismo uporabili) in vse možne začetke (od 0 do M znakov). Naloga: str. 7

```
const MaxN = 3; MaxM = 3;
type TabelaT = array [1..MaxN] of string;

procedure IzpisiVseKrajsanke(M, N: integer; Besede: TabelaT);
type MnozicaT = set of 1..MaxN;
```

[2]Literatura: A. Guttman: *R-trees: a dynamic index structure for spatial searching*, Proc. ACM SIGMOD, 1984, pp. 47–57; N. Roussopoulos, S. Kelley, F. Vincent, *Nearest neighbor queries*, Proc. ACM SIGMOD, 1995, pp. 71–79; G. R. Hjaltason, H. Samet: *Ranking in spatial databases*, Proc. SSD 1995, LNCS vol. 951, pp. 83–95; G. R. Hjaltason, H. Samet: *Distance browsing in spatial databases*, ACM TODS 24(2):256-318, June 1999; S. Bechtold, C. Böhm, D. A. Keim, H.-P. Kriegel: *A cost model for nearest-neighbor search in high-dimensional data space*, Proc. PODS 1997, pp. 78–86. Obstajajo tudi razni postopki za hitro gradnjo takega drevesa, če imamo že pri roki seznam vseh daljic, ki bi jih radi poindeksirali z njim, npr. I. Kamel, C. Faloutsos: *On packing R-trees*, Proc. CIKM 1993, pp. 490–99; S. T. Leutenegger, M. A. Lopez, J. Edgington: *STR: a simple and efficient algorithm for R-tree packing*, Proc. ICDE 1997, pp. 497–506.

```
var Skrajsanka: array [1..MaxN * MaxM] of char;
   Dolzina: integer;

  procedure Izpisi;
  var i: integer;
  begin
    for i := 1 to Dolzina do Write(Skrajsanka[i]);
    WriteLn;
  end; {Izpisi}

  procedure Rekurzija(Proste: MnozicaT);
  var StaraDolzina, i, j: integer;
  begin
    { Ostale so nam še besede iz množice Proste. Z ostalimi besedami smo
      doslej pripravili prvih Dolzina črk nastajajoče skrajšanke. }
    StaraDolzina := Dolzina;
    { Na vse možne načine izberimo naslednjo besedo. }
    for i := 1 to N do if i in Proste then begin
      Proste := Proste – [i];
      { Na vse možne načine izberimo število črk, ki jih bomo od te besede obdržali. }
      for j := 0 to M do begin
        if j > Length(Besede[i]) then break;
        { Pritaknimo začetek trenutne besede na konec nastajajoče skrajšanke. }
        if j > 0 then Skrajsanka[StaraDolzina + j] := Besede[i][j];
        Dolzina := StaraDolzina + j;
        { Če smo porabili vse besede, skrajšanko izpišimo... }
        if Proste = [ ] then begin if Dolzina > 0 then Izpisi end
        { ...sicer pa z rekurzivnim klicem nadaljujmo sestavljanje skrajšanke. }
        else Rekurzija(Proste);
      end; {for j}
      Proste := Proste + [i];
    end; {for i}
    Dolzina := StaraDolzina;
  end; {Rekurzija}

var i: integer; Proste: MnozicaT;
begin {IzpisiVseKrajsanke}
  Dolzina := 0;
  Proste := [ ];
  for i := 1 to N do Proste := Proste + [i];
  Rekurzija(Proste);
end; {IzpisiVseKrajsanke}
```

Vseh sestavljank, ki jih na ta način dobimo, je $N!(M+1)^N$ (lahko je seveda več enakih, med drugim tudi $N!$ pojavitev praznega niza), razen če ni kakšna od vhodnih besed krajša od M znakov.

Zanimivo, vendar malo težjo različico te naloge dobimo, če predpostavimo, da imamo tudi nek slovar besed in nas zanimajo le tiste skrajšanke, ki so navedene tudi v tem slovarju. Po možnosti bi do njih seveda radi prišli, ne da bi se morali ukvarjati tudi z vsemi ostalimi skrajšankami (ki jih je ogromno).

REŠITVE NALOG ZA TRETJO SKUPINO

R2004.3.1 **Otoki**

Naloga: str. 10

Najprej potopimo ves svet; število otokov je takrat 0, gladina pa zelo visoka. Gladino potem počasi spuščamo; ko pogleda nova celica iz vode, postane nov otok; takrat pregledamo njene sosede in če so tudi že kopne, se otoki zlijejo. Ko smo pregledali vse celice na določeni višini, imamo pravo število otokov za to višino in to je zdaj eden od kandidatov za največje možno število otokov sploh.

$g := \infty$;
StOtokov := 0; MaxStOtokov := 1; StKopnih := 0;
for $x := 1$ to w do for $y := 1$ to h do Otok[y, x] := 0;

while StKopnih $< w \cdot h$ do begin
  (X, Y) := najvišja med vsemi celicami, ki imajo Otok$[y, x] = 0$; (†)
  $g' := v[Y, X]$;
  if $g' < g$ and StOtokov > MaxStOtokov then MaxStOtokov := StOtokov;
  $g := g$;

  Otok$[Y, X]$:= (neka nova enolična številka otoka);
  StOtokov := StOtokov + 1;

  za vsako od štirih sosed (X', Y') celice (X, Y) ponovi:
    if (Otok$[Y', X'] \neq 0$) and (Otok$[Y', X'] \neq$ Otok$[Y, X]$) then begin
      združi otoka Otok$[Y, X]$ in Otok$[Y', X']$; (‡)
      StOtokov := StOtokov − 1;
    end if;
  end for;
end while;

Časovna zahtevnost je odvisna od tega, kako implementiramo vrstici (†) in (‡). Preprosta izvedba (†) je, da se sprehodimo po celi tabeli višin in pogledamo, katera med potopljenimi celicami je najvišja. Preprosta izvedba (‡) je, da se sprehodimo po celi tabeli *Otok* in pri vsaki celici pogledamo, če pripada prvemu od obeh otokov; če mu, tisti element tabele *Otok* popravimo, da piše, da pripada ta celica drugemu otoku. Tako nam vsako izvajanje vrstic (†) in (‡) vzame $O(n)$ časa (če je n število vseh celic v mreži), celoten postopek pa zato $O(n^2)$.

Pri mrežah velikosti 100×100, s kakršnimi imamo opravka pri tej nalogi, je že ta preprosta izvedba dovolj hitra, zato na njej temelji tudi spodnji program. Za večje mreže, na primer 1000×1000, pa bi morali uporabiti kaj boljšega. Boljša implementacija (†) je, da na začetku posortiramo vse celice po padajoči $v[y, x]$, kar lahko naredimo v času $O(n \log n)$ (npr. postopek quicksort, ki ga imajo mnogi jeziki, npr. C in C++, že kar v standardni knjižnici). Boljša implementacija (‡) je, da z iskanjem v širino ali globino obiščemo le vse celice enega od obeh otokov in jih popravimo, da kažejo na drugi otok; pametno je obiskati manjšega od obeh otokov (velikosti otokov hranimo v neki dodatni tabeli): če naredimo tako, bo vsaka celica, ki ji spremenimo pripadnost otoku, pripadala po novem otoku, ki je vsaj dvakrat tolikšen kot tisti, ki mu je pripadala prej, in ker noben otok ne more imeti več kot n celic, tudi nobena celica ne bo več kot $\lg n$-krat spremenila pripadnosti otoku; to nam zagotavlja, da bo vrstica

(‡) porabila vsega skupaj le $O(n \log n)$ časa. Zato tudi cel postopek porabi $O(n \log n)$ časa.[3]

```
program Otoki;
const
  MaxW = 100; MaxH = 100;
  DX: array [1..4] of integer= (−1, 1, 0, 0);
  DY: array [1..4] of integer= (0, 0, −1, 1);
var
  W, H: integer; { velikost mreže }
  V, Otok: array [1..MaxH, 1..MaxW] of integer;
  i, x, y, G, NovaG, NX, NY, SX, SY, NO: integer;
  StOtokov, MaxStOtokov, StKopnih, PrviProsti: integer;
  T: text;
begin
  { Preberimo vhodno datoteko. }
  Assign(T, 'otoki.in'); Reset(T); ReadLn(T, H, W);
  for y := 1 to H do begin
    for x := 1 to W do Read(T, V[y, x]);
    ReadLn(T);
  end; {for y};
  Close(T);

  { Na začetku so vse celice pod vodo. Postavimo G na nekaj velikega. }
  G := V[1, 1];
  for y := 1 to H do for x := 1 to W do begin
    if G < V[y, x] then G := V[y, x];
    Otok[y, x] := 0;
  end; {for y}
  G := G + 1;
  StOtokov := 0; MaxStOtokov := 1;
  PrviProsti := 1; StKopnih := W * H;

  while StKopnih > 0 do begin

    { Znižajmo gladino toliko, da kakšna celica na novo pogleda iz vode. }
    NovaG := −1;
    for y := 1 to H do for x := 1 to W do
      if (Otok[y, x] = 0) and (V[y, x] > NovaG) then
        begin NovaG := V[y, x]; NX := x; NY := y end;

    { Če se je gladina res znižala, imamo v StOtokov pravo število otokov
      pri dosedanji gladini — mogoče je to največje število otokov doslej. }
    if (NovaG < G) and (StOtokov > MaxStOtokov) then
      MaxStOtokov := StOtokov;

    G := NovaG;
    { Celica (NX, NY) je pogledala iz vode. }
    Otok[NY, NX] := NY * W + NX;
    StOtokov := StOtokov + 1; StKopnih := StKopnih − 1;
    { Če ima kaj kopnih sosed, se otoki združujejo. }
    for i := 1 to 4 do begin
```

[3] Celoten postopek bi se dalo izboljšati celo do zahtevnosti $O(n)$, če bi poznali porazdelitev višin dovolj dobro, da bi lahko uporabili za urejanje celic po višini katerega od postopkov, ki porabijo le $O(n)$ časa, na primer bucket sort. Da bi tudi zlivanje otokov porabilo skupaj le $O(n)$ časa, bi morali uporabiti znano gozdnato strukturo za disjunktne množice (*disjoint-set forests*, gl. npr. Cormen *et al.*, *Introduction to Algorithms*, 22. pogl. v prvi izdaji, 21. v drugi).

```
      SX := NX + DX[i]; SY := NY + DY[i];
      if (SX >= 1) and (SY >= 1) and (SX <= W) and (SY <= H) then
      if (Otok[SY, SX] <> 0) and (Otok[NY, NX] <> Otok[SY, SX]) then begin
        { Soseda (SX, SY) je kopna in ne pripada istemu otoku
          kot (NX, NY), zato ta dva otoka združimo. }
        NO := Otok[NY, NX];
        for y := 1 to H do for x := 1 to W do
          if Otok[y, x] = NO then Otok[y, x] := Otok[SY, SX];
        StOtokov := StOtokov - 1;
      end; {if}
    end; {for i}
  end; {while}

  { Izpišimo rezultat. }
  Assign(T, 'otoki.out'); Rewrite(T); WriteLn(T, MaxStOtokov); Close(T);
end. {Otoki}
```

Na `http://www.ngdc.noaa.gov/mgg/global/relief/ETOPO2/` so podatki o površju Zemlje (z ločljivostjo $2' \approx 3{,}7\,\mathrm{km}$ — torej mreža 10800×5400). Pri $g = 0$ (torej pri sedanji gladini oceanov) je na tej mreži 6024 otokov, največ otokov (47397) pa je pri $g = 306\,\mathrm{m}$ nad sedanjo gladino. (Pri tem smo definicijo sosednosti med celicami popravili tako, da smo upoštevali, da je Zemlja okrogla.) Kakšne posebne zveze z resničnostjo te številke najbrž nimajo, saj je v resnici ogromno otokov (in vzpetin ipd.) manjših od 3,7 km in se jih na tej mreži najbrž sploh ne opazi.

Zanimivo in malo težjo različico naloge dobimo, če poskusmo podpreti tudi veliko večje mreže, tako da imamo lahko mrežo le na disku, ne gre pa cela v glavni pomnilnik našega računalnika.

R2004.3.2 SBN

Ukaza SBN ni težko uporabiti za primerjanje števil po velikosti: a je manjši od b natanko tedaj, ko je razlika $a - b$ negativna.

Naloga: str. 11

```
          SBN r256, r12, r34, Oznaka
          { Tukaj vemo, da je r12 ≥ r34. }
          ...
 Oznaka:  { Tukaj vemo, da je r12 < r34. }
          ...
```

Ker imamo precej več kot n registrov (registrov je 256, naloga pa pravi, da je n največ 100), lahko preostale registre uporabimo za odlaganje pomožnih vrednosti, kot je v gornjem primeru razlika r12 − r34, ki smo jo vpisali v r256.

Prenašanje vrednosti iz enega registra v drugega je še enostavnejše. Prireditev r12 := r34 izvedemo takole:

```
SBN r12, r34, 0
```

Če bi radi zamenjali vrednosti v dveh registrih, potrebujemo tri prireditve:

```
SBN r256, r34, 0
SBN r34, r12, 0
SBN r12, r256, 0
```

Opisane operacije so pravzaprav že vse, kar potrebujemo za urejanje števil. Spomnimo se namiga iz besedila naloge in ga zapišimo v obliki algoritma (temu algoritmu se, mimogrede, reče „urejanje z izbiranjem“, *selection sort*):

```
var i, j: integer;
begin
  for i := 1 to n − 1 do
    { V registrih r[1], . . . , r[i − 1] se že nahaja i − 1 najmanjših
      vrednosti tabele r. V tej iteraciji zunanje zanke bomo poskrbeli,
      da bo v register r[i] prišla najmanjša izmed preostalih vrednosti, torej
      izmed teh, ki so trenutno v registrih r[i], . . . , r[n]. }
    for j := i + 1 to n do
      if not (r[i] < r[j]) then
        zamenjaj r[i] in r[j];
end.
```

Naš namišljeni procesor žal ne podpira posrednega naslavljanja; ne moremo mu na primer reči, naj nekaj naredi s tistim registrom, čigar številka je ta hip shranjena v registru `r123`. Možen izhod iz te zagate je, da vse zanke „razvijemo“ (*loop unrolling*) in kar ponovimo vse inštrukcije iz telesa zanke po enkrat za vsako vrednost števca. Tako pridemo do naslednje rešitve:

```
program Sbn;
var T: text; i, j, n: integer;
begin
  Assign(T, 'sbn.in'); Reset(T); ReadLn(T, n); Close(T);
  Assign(T, 'sbn.out'); Rewrite(T);
  for i := 1 to n − 1 do
    { Cilj notranje zanke je spraviti v r[i] najmanjšo od vrednosti r[i..n]. }
    for j := i + 1 to n do begin
      { Če je r[i] < r[j], skoči na end_i_j. }
      WriteLn(T, 'sbn r256, r', i, ', r', j, ', end_', i, '_', j);
      { Sicer zamenjaj registra r[i] in r[j]. }
      WriteLn(T, 'sbn r', j, ', r', i, ', 0'); { r[j] := r[i]. }
      { r[i] := r[i] − (r[i] − stara vrednost r[j]). }
      WriteLn(T, 'sbn r', i, ', r', i, ' r256');
      Write(T, 'end_', i, '_', j, ': ');
    end; {for j}
  end; {for i}
  { Izpišimo še en NOP — le toliko, da imamo kam pripeti zadnjo labelo. }
  WriteLn(T, 'sbn r256, r256, r256');
  Close(T);
end. {Sbn}
```

Pri zamenjavi smo porabili samo dva ukaza namesto treh, ker smo si pomagali z razliko med vrednostma registrov r[i] in r[j], ki smo jo naračunali malo prej pri primerjavi in jo shranili v r256. Tako porabimo po tri ukaze za vsak par registrov, kar nam da program dolžine $3n(n-1)/2+1$ (pri shemi točkovanja iz opisa naloge bi za to praviloma dobili sedem točk od desetih); če bi za zamenjavo porabili tri ukaze, bi bili s primerjavo vred že štirje in program bi bil dolg $4n(n-1)/2+1$ ukazov (za kar bi dobili šest točk od desetih).

Tako dolgo zaporedje ukazov smo dobili, ker smo morali v celoti razviti obe zanki, po i in po j. Do krajšega zaporedja pridemo, če najmanjše doslej odkrite vrednosti ne hranimo v r[i], pač pa v nekem posebnem registru (recimo mu m), neodvisnem od i.

```
var i, j, m: integer;
begin
  for i := 1 to n − 1 do begin
    { V registrih r[1], . . . , r[i − 1] se že nahaja i − 1 najmanjših
      vrednosti tabele r. V tej iteraciji zunanje zanke bomo poskrbeli,
      da bo v register r[i] prišla najmanjša izmed preostalih vrednosti, torej
      izmed teh, ki so trenutno v registrih r[i], . . . , r[n].
      Za začetek bomo z naslednjo zanko (po j) poskrbeli, da se bo ta
      najmanjša vrednost znašla v spremenljivki m, ostale vrednosti pa
      (mogoče malo premešane) v registrih r[i], . . . , r[n − 1]. }
    m := r[n]; { Začasno si mislimo, da je register r[n] zdaj „prazen“. }
    for j := n − 1 downto 1 do begin
      if j < i then break;
      if m < r[j] then continue;
      zamenjaj r[j] in m;
    end; {for j}
    { Zdaj je m najmanjša izmed vrednosti, ki so bile prej v r[i], . . . , r[n].
      Torej jo moramo vpisati v register r[i], dosedanjo vrednost r[i] pa
      pred tem premaknimo v prazni register r[n]. }
    r[n] := r[i]; r[i] := m; {⋆}
  end; {for i}
end.
```

Ker hočemo, da nam kode glavne zanke ne bi bilo treba ponavljati po enkrat za vsako vrednost i, se v njej ne smemo neposredno sklicevati na r[i]. Zato moramo vrstico {⋆} predelati v nekaj takšnega:

```
    for j := 1 to n − 1 do begin
      if j < i then continue;
      r[n] := r[j]; r[j] := m; break;
    end; {for j}
```

Zunanja zanka zdaj registra r[i] ne uporablja več neposredno, tako da njena koda ni več odvisna od trenutne vrednosti števca i. Zato nam v zaporedju ukazov SBN ne bo treba ponavljati telesa te zanke po enkrat za vsak i. Tako dobimo naslednjo rešitev z zaporedjem $8n-4$ ukazov SBN (pri našem točkovanju bi dosegla osem točk od desetih):

```
program Sbn;
const m = 254; i = 255; temp = 256;
var T: text; j, n: integer;
begin
  Assign(T, 'sbn.in'); Reset(T); ReadLn(T, n); Close(T);
  Assign(T, 'sbn.out'); Rewrite(T);
  WriteLn(T, 'sbn r', i, ', ', 1, ', 0');             { i := 1 }
  Write(T, 'zanka: ');
  WriteLn(T, 'sbn r', m, ', r', n, ', 0');            { m := r[n] }
  for j := n − 1 downto 1 do begin
    { if j < i then break }
```

```
    WriteLn(T, 'sbn r', temp, ', ', j, ', r', i, ', a_1');
    { if m < r[j] then continue }
    WriteLn(T, 'sbn r', temp, ', r', m, ', r', j, ', a_', j);
    { Zamenjaj r[j] in m. }
    WriteLn(T, 'sbn r', j, ', r', m, ', 0');         { r[j] := m }
    WriteLn(T, 'sbn r', m, ', r', j, ', r', temp);  { m := m − (m − stara r[j]) }
    Write(T, 'a_', j, ': ');
  end; {for j}
  { Zdaj je m = min(r[i..n]). Izvesti moramo r[n] := r[i] in r[i] := m. }
  for j := 1 to n − 1 do begin
    { if j < i then continue }
    WriteLn(T, 'sbn r', temp, ', ', j, ', r', i, ', b_', j);
    { Sicer je j = i. }
    WriteLn(T, 'sbn r', n, ', r', j, ', 0');                    { r[n] := r[i] }
    WriteLn(T, 'sbn r', j, ', r', m, ', 0');                    { r[j] := m }
    WriteLn(T, 'sbn r', temp, ', 0, 1, b_', n − 1);            { break }
    Write(T, 'b_', j, ': ');
  end; {for j}
  WriteLn(T, 'sbn r', i, ', r', i, ', −1');                    { i := i + 1 }
  { if i < n then goto zanka }
  WriteLn(T, 'sbn r', temp, ', r', i, ', ', n, ', zanka');
  Close(T);
end. {SBN}
```

Še krajšo in elegantnejšo rešitev dobimo, če uporabimo malo drugačen algoritem za urejanje.

```
var i, j: integer;
begin
  for i := 1 to n − 1 do
    { V zadnjih i − 1 registrih se že nahaja i − 1 največjih vrednosti
      cele tabele. V tej iteraciji zunanje zanke bomo poskrbeli, da bo v
      register r[n − i + 1] prišla (n − i + 1)-va največja vrednost. }
    for j := 1 to n − 1 do
      if not (r[j] < r[j + 1]) then
        zamenjaj r[j] in r[j + 1];
end.
```

Ko pride notranja zanka do največje vrednosti v še neurejenem delu tabele, bo ugotovila, da ta register ni manjši od naslednjega, in ju bo zato zamenjala; tako se tista največja vrednost premakne po tabeli za eno mesto naprej; v naslednji iteraciji bomo delali z naslednjim registrom, torej ravno s tistim, v katerega smo pravkar premaknili največjo vrednost; zato bo spet treba izvesti zamenjavo in tako naprej. Tako potone že ob prvem prehodu največja vrednost do konca tabele, ob drugem prehodu druga največja vrednost na predzadnje mesto tabele in tako naprej. Ta postopek se imenuje „urejanje z mehurčki“ (*bubble sort*).

Za naše potrebe je ta postopek zelo primeren, ker sta gnezdeni zanki „neodvisni“ druga od druge — notranja zanka (po j) ne uporablja vrednosti i. Zato njenih ukazov v programu ni treba imeti v več kopijah (po eno za vsak i). Tako dobimo program, dolg samo $3n - 1$ ukazov.

```
program Sbn;
var T: text; i, n: integer;
```

```
begin
  Assign(T, 'sbn.in'); Reset(T); ReadLn(T, n); Close(T);
  Assign(T, 'sbn.out'); Rewrite(T);

  { Register r255 šteje prehode čez tabelo. }
  WriteLn(T, 'sbn r255, 0, ', n − 1'); { r255 := −(n − 1); }
  Write(T, 'zanka: ');
  for i := 1 to n − 1 do begin

    { Če je r[i] < r[i + 1], preskoči zamenjavo. }
    WriteLn(T, 'sbn r256, r', i, ', r', i + 1, ', end_', i);
    { Zamenjaj registra r[i] in r[i + 1]. }
    WriteLn(T, 'sbn r', i + 1, ', r', i, ', 0');
    WriteLn(T, 'sbn r', i, ', r', i, ' r256');

    Write(T, 'end_', i, ': ');
  end; {for i}

  { Povečajmo r255 za 1; ko ni več negativen, pomeni, da smo izvedli
    vseh n − 1 prehodov in lahko nehamo. }
  WriteLn(T, 'sbn r255, r255, -1, zanka');
  Close(T);
end. {Sbn}
```

Če nam je bolj kot dolžina programa pomemben čas izvajanja (število ukazov SBN, ki jih je treba izvesti, da uredimo n registrov), je odlična izbira urejanje z zlivanjem (mergesort). To nam zagotavlja časovno zahtevnost $O(n \log n)$, kar je načeloma precej bolje od zahtevnosti $O(n^2)$, ki jo dosežeta urejanje z izbiranjem in z mehurčki. Slabost urejanja z zlivanjem je, da potrebuje $n + 1$ pomožnih registrov namesto enega samega kot ostali tu opisani algoritmi. Če nas to hudo moti, lahko uporabimo urejanje s kopico (heapsort), ki potrebuje en sam pomožni register. Tako urejanje z zlivanjem kot s kopico lahko implementiramo tako, da dobimo (za urejanje n registrov) zaporedje približno $2n^2$ ukazov; je pa bilo urejanje s kopico vsaj v naši implementaciji malo počasnejše od urejanja z zlivanjem, saj je izvedlo slednje približno $2{,}5\, n \lg n$ ukazov, urejanje s kopico pa približno $4\, n \lg n$. Z algoritmom quicksort nam je uspelo dobiti sicer še hitrejše programe (do $2n \lg n$), vendar so tudi precej daljši (približno $n^4/6$ ukazov), razlike v hitrosti v primerjavi z zlivanjem pa postanejo opaznejše šele pri velikih n, ko je rešitev s quicksortom nesprejemljivo dolga.

R2004.3.3 Pamet

Naloga: str. 12

Nalogo lahko rešimo z rekurzijo. Rekurzivni podprogram preizkusi razne možnosti glede tega, koliko kosov pameti kupiti pri trenutnem trgovcu, pri vsaki možnosti pa izvede rekurzivni klic, da pregleda še možnosti nakupa ostalih kosov pameti pri ostalih trgovcih. Ko najdemo kakšno rešitev (torej ko uspemo nakupiti pravo število kosov pameti), jo primerjamo z najcenejšo doslej znano (spremenljivka MinCena) in če je cenejša, si jo zapomnimo. Med rekurzijo lahko vrednost MinCena uporabimo tudi za izogibanje preiskovanju neobetavnih nakupov („razveji in omeji“, *branch and bound*) — če so že doslej kupljeni kosi stali več kot MinCena denarja, nima smisla riniti še globlje v rekurzijo, saj bo končna cena tako dobljenih nakupov samo še višja in torej pri tem gotovo ne bomo dobili nobene boljše rešitve od najboljše doslej znane.

```
program Butalci;
```

```
const
  MaxNB = 25;                  { največje število Butalcev }
  MaxNT = 500000;              { največje število trgovcev }
  MaxVsotaCen = 1000000000;    { vsota vseh cen v datoteki }
var
  NB, NT: integer;
  NK, SkupajNK: array [1..MaxNT] of integer;
  Cene: array [0..MaxNT, 0..MaxNB] of integer;
  MinCena: integer; { najmanjša doslej najdena cena za NB kosov pameti }

  { Ta podprogram predpostavi, da smo doslej že kupili NB − OstaloKosov kosov
    pameti in za to plačali CenaDoslej denarja, od trgovcev TrgovciOd..NT pa bi
    radi čim ceneje kupili še ostalih OstaloKosov pameti. }
  procedure Rekurzija(CenaDoslej, TrgovciOd, OstaloKosov: integer);
  var i, Cena: integer;
  begin
    i := 0;
    while (i <= OstaloKosov) and (i <= NK[TrgovciOd]) do begin
      Cena := CenaDoslej + Cene[TrgovciOd, i];
      if Cena < MinCena then begin
        if i = OstaloKosov then MinCena := Cena
        else if TrgovciOd < NT then
          { Nadaljujmo pri ostalih trgovcih, če imajo seveda vsi skupaj
            sploh dovolj pameti za naše potrebe. }
          if SkupajNK[TrgovciOd + 1] >= OstaloKosov − 1 then
            Rekurzija(Cena, TrgovciOd + 1, OstaloKosov − i);
      end; {if}
      i := i + 1;
    end; {while}
  end; {Rekurzija}

var T: text; it, ip, Cena: integer;
begin {Butalci}
  Assign(T, 'pamet.in'); Reset(T);
  ReadLn(T, NB, NT);
  for it := 1 to NT do begin
    Cene[it, 0] := 0;
    Read(T, NK[it]);
    for ip := 1 to NK[it] do begin
      Read(T, Cena);
      if ip <= NB then Cene[it, ip] := Cene[it, ip − 1] + Cena;
    end; {for}
    ReadLn(T);
  end; {for}

  { SkupajNK[i] = koliko kosov imajo trgovci od i-tega naprej. }
  SkupajNK[NT] := NK[NT];
  for it := NT − 1 downto 1 do
    SkupajNK[it] := SkupajNK[it + 1] + NK[it];

  MinCena := MaxVsotaCen + 1;
  Rekurzija(0, 1, NB);

  { Izpišimo rezultat. }
  Assign(T, 'pamet.out'); Rewrite(T); WriteLn(T, MinCena); Close(T);
end. {Butalci}
```

Če je Butalcev in/ali trgovcev veliko oz. če je testni primer dovolj neugodno sestavljen, bo ta program porabil preveč časa. Na testnih podatkih z našega tekmovanja bi zgornji program v sprejemljivo kratkem času rešil sedem od desetih testnih primerov. Do učinkovitejše rešitve bi prišli, če bi upoštevali, da podprogram Rekurzija vedno poišče najcenejši način, kako kupiti OstaloKosov kosov pameti od trgovcev TrgovciOd..NT. Vrednost, ki jo vrne, je enaka vrednosti tega nakupa, povečani za CenaDoslej. Torej bi lahko to vrednost (brez CenaDoslej), ko jo prvič izračunamo, shranili v neki tabeli in je kasneje ne bi bilo treba računati ponovno, ampak bi jo samo pobrali od tam. Takšni tehniki shranjevanja delnih rezultatov pravimo pomnjenje ali *memoizacija*. Lahko pa bi te rezultate računali tudi čisto sistematično, z gnezdenima zankama po TrgovciOd in po OstaloKosov (*dinamično programiranje*).

R2004.3.4 Izomorfizem

Naloga: str. 13

Izberimo v vsakem od obeh grafov koren in ga tako predelajmo v hierarhično urejeno drevo. V takšnem drevesu lahko povezave usmerimo, da kažejo vedno od staršev na otroke (torej od vozlišča, ki je bližje korenu, na tisto, ki je dlje od njega); definicija izomorfizma naj ostane takšna kot prej, le da je zdaj pri ugotavljanju, ali lahko en graf dobimo iz drugega samo s preimenovanjem vozlišč, treba upoštevati tudi smer povezav.

Če v prvotnem grafu koren ni imel stopnje 3, nastane drevo, v katerem ima vsako vozlišče največ dve poddrevesi. Poddrevo, ki se začne v prvem drevesu pri u, in poddrevo, ki se začne v drugem drevesu pri v, sta izomorfni ali pa nista; naj nam to pove vrednost $I(u, v)$. Potem je: $I(u, v) = \mathsf{false}$, če nimata enako število poddreves; $I(u, v) = \mathsf{true}$, če sta oba brez poddreves; $I(u, v) = I(u', v')$, če imata vsak po eno poddrevo (u' in v'); in $I(u, v) = (I(u', v') \wedge I(u'', v'')) \vee (I(u', v'') \wedge I(u'', v'))$, če imata vsak po dve poddrevesi (u', u'' ter v', v''; eno od poddreves u-ja mora biti izomorfno enemu od poddreves v-ja, drugo pa drugemu, pri čemer vrstni red poddreves ni pomemben).[4] Vrednosti $I(u, v)$ lahko računamo z rekurzijo, lahko pa si že izračunane vrednosti tudi shranjujemo v kakšno tabelo, da jih kasneje ne bi bilo treba računati ponovno, če bi jih slučajno spet potrebovali. Slednje nam zagotavlja časovno zahtevnost $O(n^2)$. Spotoma lahko pripravljamo tudi tabelo, ki pove, katero vozlišče prvega drevesa ustreza kateremu vozlišču drugega drevesa (če sta drevesi res izomorfni).

Recimo, da sta prvotna grafa izomorfna, torej obstaja med njunima množicama vozlišč bijekcija f, ki spoštuje povezave: za vsak par vozlišč u in v velja $(u, v) \in E_1 \Leftrightarrow (f(u), f(v)) \in E_2$ (pri tem je E_1 množica povezav prvega, E_2 pa drugega grafa). Izberimo prvemu grafu poljubno vozlišče (s stopnjo 1 ali 2) kot koren; recimo mu r. Če zdaj poskusimo pri drugem grafu za koren vzeti po vrsti vsa njegova vozlišča, bomo prej ali slej preizkusili kot koren tudi $f(r)$ in takrat s primerjanjem dreves lahko ugotovimo, da izomorfizem res obstaja.

Po drugi strani, če enkrat ugotovimo, da izomorfizem med drevesoma z izbranima korenoma res obstaja, to seveda pomeni, da obstaja tudi med prvotnima grafoma.

Torej lahko izomorfizem res odkrivamo tako, da izberemo koren prvega drevesa in potem preizkusimo vse možne korene drugega; skupaj ima ta

[4] Če bi imeli opravka z drevesi, v katerih imajo lahko vozlišča tudi po več kot dve poddrevesi, bi ob primerjavi dveh vozlišč s po k poddrevesi morali preizkusiti vseh $k!$ načinov, kako razporediti poddrevesa enega vozlišča v pare s poddrevesi drugega vozlišča.

postopek časovno zahtevnost $O(n^3)$ in je za našo nalogo dovolj dober.

```
program Izomorfizem;
const MaxN = 1000;
type
  GrafT = record
    Stopnja: array [1..MaxN] of integer;
    Sosedje: array [1..MaxN, 1..3] of integer;
    { Ostala polja uporabljamo, ko grafu izberemo koren in ga predelamo v drevo. }
    Koren: integer;
    Otrok: array [1..MaxN, 1..2] of integer;
    StOtrok: array [1..MaxN] of integer;
  end; {GrafT}

procedure PreberiGraf(var T: text; N: integer; var G: GrafT);
var i, u, v: integer;
begin
  for i := 1 to N do G.Stopnja[i] := 0;
  for i := 1 to N − 1 do begin
    ReadLn(T, u, v);
    G.Stopnja[u] := G.Stopnja[u] + 1; G.Sosedje[u, G.Stopnja[u]] := v;
    G.Stopnja[v] := G.Stopnja[v] + 1; G.Sosedje[v, G.Stopnja[v]] := u;
  end; {for i}
end; {PreberiGraf}

{ Ko v grafu izberemo koren, lahko iz njega naredimo drevo in za vsako
  vozlišče pogledamo, kdo so njegovi otroci oz. poddrevesa. }
procedure PostaviKoren(var G: GrafT; R: integer);

  procedure Rekurzija(u, Oce: integer);
  var i, v: integer;
  begin
    G.StOtrok[u] := 0;
    for i := 1 to G.Stopnja[u] do begin
      v := G.Sosedje[u, i]; if v = Oce then continue;
      G.StOtrok[u] := G.StOtrok[u] + 1;
      G.Otrok[u, G.StOtrok[u]] := v;
      Rekurzija(v, u);
    end; {for i}
  end; {Rekurzija}

begin {PostaviKoren}
  G.Koren := R;
  Rekurzija(R, −1);
end; {PostaviKoren}

var Preslikava: array [1..MaxN] of integer;

function StaDrevesiIzomorfni(N: integer; var G1, G2: GrafT): boolean;

  { Naslednji podprogram preveri izomorfnost dveh poddreves. }
  function Preveri(u, v: integer): boolean;
  var Sta: boolean;
  begin
    if G1.StOtrok[u] <> G2.StOtrok[v] then Sta := false
    else begin
      if G1.StOtrok[u] = 0 then Sta := true
      else if G1.StOtrok[u] = 1 then
```

```
        Sta := Preveri(G1.Otrok[u, 1], G2.Otrok[v, 1])
      else Sta :=   (Preveri(G1.Otrok[u, 1], G2.Otrok[v, 1])
                          and Preveri(G1.Otrok[u, 2], G2.Otrok[v, 2]))
                 or (Preveri(G1.Otrok[u, 1], G2.Otrok[v, 2])
                          and Preveri(G1.Otrok[u, 2], G2.Otrok[v, 1]));
      if Sta then Preslikava[u] := v;
    end; {if}
    Preveri := Sta;
  end; {Preveri}

begin {StaDrevesiIzomorfni}
  StaDrevesiIzomorfni := Preveri(G1.Koren, G2.Koren);
end; {StaDrevesiIzomorfni}

var T: text; u, N: integer; G1, G2: GrafT; StaIzo: boolean;
begin
  { Preberimo oba grafa. }
  Assign(T, 'izomorf.in'); Reset(T); ReadLn(T, N);
  PreberiGraf(T, N, G1); PreberiGraf(T, N, G2);
  Close(T);

  { Izberimo koren prvega grafa. }
  u := 1;
  while u <= N do if G1.Stopnja[u] < 3 then break else u := u + 1;
  PostaviKoren(G1, u);

  { Preizkusimo možne korene drugega grafa. }
  u := 1; StaIzo := false;
  while (u <= N) and not StaIzo do begin
    if G2.Stopnja[u] = G1.Stopnja[G1.Koren] then begin
      PostaviKoren(G2, u);
      StaIzo := StaDrevesiIzomorfni(N, G1, G2);
    end; {if}
    u := u + 1;
  end; {while}

  { Izpišimo rezultat. }
  Assign(T, 'izomorf.out'); Rewrite(T);
  if StaIzo then for u := 1 to N do WriteLn(T, u, ' ', Preslikava[u]);
  Close(T);
end. {Izomorfizem}
```

Ta algoritem je mogoče še precej izboljšati. Ko si enkrat izberemo oba korena, lahko izomorfnost dreves preverjamo učinkoviteje kot zgoraj. Poddrevesa pregledujmo po naraščajoči globini (globina poddrevesa = dolžina najdaljše veje v njem); označevali jih bomo s številkami ali „barvami“ in to tako, da izomorfna poddrevesa dobijo isto barvo, neizomorfna pa različno. Za začetek dajmo vsem listom (poddrevesa globine 0) številko 0, saj so si vsa izomorfna. Poddrevesi globine 1 sta si izomorfni natanko tedaj, ko imata obe enako število listov, zato lahko takim poddrevesom dodelimo kot barvo kar število listov. Za globlja poddrevesa pa lahko razmišljamo takole: dve taki poddrevesi sta si izomorfni, če sta obe enako globoki in če za vsako barvo velja, da imata obe enako število otrok take barve. Če torej za vsako poddrevo pripravimo seznam barv njegovih otrok in ga uredimo, morata dve izomorfni poddrevesi dobiti zdaj popolnoma enak seznam. V našem primeru nima nobeno poddrevo več kot dveh otrok, zato so ti seznami dolgi največ dva elementa in je urejanje trivialno. Nato sestavimo zaporedje, katerega elementi so vsi seznami, dobljeni pri poddrevesih določene globine; to zaporedje uredimo, tako da pridejo enaki seznami (ki predstavljajo izomorfna poddrevesa) skupaj; zdaj torej ni težko pripisati vsaki skupini izomorfnih poddreves neko številko, ki je prej še nismo uporabili. Na koncu moramo le pogledati, če sta celotni drevesi dobili isto številko ali ne. Če obstaja n_i poddreves globine i, porabimo za urejanje tistega zaporedja $O(n_i \log n_i)$ časa, za vse globine skupaj pa zato $O(\sum_i n_i \log n_i) = O(\sum_i n_i \log n) = O(n \log n)$, kar je vsekakor precej bolje kot $O(n^2)$, kar je dosegel zgornji preprostejši algoritem.

Še ena izboljšava pa je, da pri drugem grafu ni treba preizkušati vseh možnih korenov, če pri prvem grafu koren pazljivo izberemo. Recimo, da bi porezali iz grafa vsa vozlišča s stopnjo 1; dobimo nek manjši graf, v katerem imajo nekatera vozlišča najbrž spet stopnjo 1; porežimo še ta; tako nadaljujemo in graf lupimo po plasteh od zunaj navznoter kot čebulo. Na koncu ostane neka plast z enim ali dvema vozliščema in ko pobrišemo še to, je graf prazen. Vozlišče (ali vozlišči) v zadnji plasti se imenuje *center* (središče) grafa. Če je bilo vsega skupaj k plasti in mi zdaj izberemo središče kot koren ter graf predelamo v drevo, bodo vsi listi na globini vsaj $k - 1$.

Poiščimo zdaj še center drugega grafa. Če se dobljeno število plasti kaj razlikuje od tistega pri prvem grafu, grafa že ne moreta biti izomorfna (kajti če sta izomorfna, lahko enega dobimo iz drugega samo s preimenovanjem vozlišč, to pa na odkrivanje centrov ne more vplivati). Če pa ima drugi graf tudi k plasti in bi ga zdaj radi predelali v drevo, da ga bomo lahko primerjali z drevesom, dobljenim iz prvega grafa, je jasno, da mora imeti drevo drugega grafa tudi vse liste na globini vsaj $k - 1$, tako kot jih ima prvo, saj drugače drevesi ne bosta mogli biti izomorfni. Torej za koren v drugem grafu nima smisla izbirati česa drugega kot centra drugega grafa! Ker ima lahko graf dva centra, moramo zdaj preizkusiti le dve možni drevesi, ki ju lahko dobimo iz drugega grafa, ne pa več $O(n)$ možnih dreves kot prej.

Ker lahko center poiščemo v času $O(n)$, bo preverjanje izomorfizma trajalo vsega skupaj $O(n^2)$, če uporabimo preprosti algoritem za izomorfizem dreves z začetka te rešitve, in $O(n \log n)$, če uporabimo izboljšani algoritem.

Majhna slabost naše naloge je, da v primeru, ko drevesi nista izomorfni, od programa zahteva samo prazno izhodno datoteko. Torej bi lahko kak bleferski program vedno pustil prazno izhodno datoteko in s tem pravilno rešil vse testne primere, pri katerih vhodna grafa res nista izomorfna (od desetih testnih primerov na našem tekmovanju je bil tak sicer en sam).

R2004.3.5 Čebelica Maja

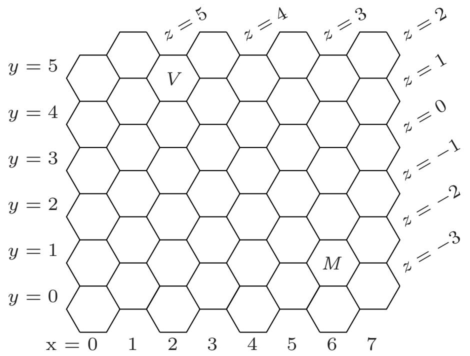
Naloga: str. 14

Pri računanju razdalj na šesterokotni mreži je koristno vpeljati novo koordinatno os $z = y - \lfloor x/2 \rfloor$, kot to prikazuje spodnja slika.

Recimo, da se hočemo premakniti iz točke (x_1, z_1) v (x_2, z_2); označimo $\triangle x = x_2 - x_1$ in $\triangle z = z_2 - z_1$.

Nekatere smeri premikanja nam spremenijo eno koordinato, drugo pa pustijo pri miru, dve smeri pa spremenita obe koordinati, vendar v nasprotni smeri (eno povečata in drugo zmanjšata). Če torej uporabljamo samo premike, ki spreminjajo po eno koordinato, lahko do $(\triangle x, \triangle z)$ pridemo v $|\triangle x| + |\triangle z|$ korakih.

Če sta $\triangle x$ in $\triangle z$ oba enako predznačena, je to tudi najkrajša možna pot. Smeri, ki spreminjata obe koordinati naenkrat, nam pri takih premikih ne ko-



Opisi premikov po mreži so precej preprostejši, če celice predstavimo s koordinatami (x, z) namesto (x, y).

Smer premika	Sprememba koordinat $\triangle x$	$\triangle z$
↑	0	1
↗	1	0
↘	1	-1
↓	0	-1
↙	-1	0
↖	-1	1

ristita, ker eno koordinato povečujeta in drugo zmanjšujeta, mi pa bi radi ali obe zmanjšali ali pa obe povečali.

Če pa sta $\triangle x$ in $\triangle z$ različno predznačena, lahko uporabimo najprej eno od smeri, ki spreminja obe koordinati, da dosežemo pravi premik pri tisti koordinati, ki ima manjšo absolutno vrednost. Pri tem pa smo opravili že tudi del premika pri drugi koordinati (in to v pravo smer) in moramo zdaj opraviti v eni od smeri, ki spreminja le to koordinato, samo še toliko korakov, da doseže tudi ta koordinata pravo vrednost. Če je na primer $|\triangle x| \leq |\triangle z|$, naredimo najprej $|\triangle z|$ korakov, ki spreminjajo tako x kot z, nato pa še $|\triangle x| - |\triangle z|$ korakov, ki spreminjajo samo x. Če velja $|\triangle x| \leq |\triangle z|$, je razmislek podoben. Skupno število korakov je tako $\max\{|\triangle x|, |\triangle z|\}$.

```
program HotInsectAction;

  function Razdalja(X1, Y1, X2, Y2: integer): integer;
  var Z1, Z2, DX, DZ: integer;
  begin
    Z1 := Y1 − X1 div 2; Z2 := Y2 − X2 div 2;
    DX := X2 − X1; DZ := Z2 − Z1;
    if ((DX > 0) and (DZ > 0)) or ((DX < 0) and (DZ < 0))
    then Razdalja := Abs(DX) + Abs(DZ)
    else if Abs(DX) > Abs(DZ) then Razdalja := Abs(DX) else Razdalja := Abs(DZ);
  end; {Razdalja}

var MX, MY, VX, VY, i, X, Y: integer; T, U: text;
begin {HotInsectAction}
  Assign(T, 'maja.in'); Reset(T);
  Assign(U, 'maja.out'); Rewrite(U);
  ReadLn(T, MX, MY); ReadLn(T, VX, VY);
  for i := 1 to 3 do begin
    ReadLn(T, X, Y);
    WriteLn(U, Razdalja(MX, MY, X, Y), ' ', Razdalja(VX, VY, X, Y));
  end; {for i}
  Close(T); Close(U);
end. {HotInsectAction}
```

REŠITVE NALOG ŠESTEGA TEKMOVANJA IZ UNIXA

Naloga: str. 15

R2004.U.1

Primer rešitve v pythonu:

```
import sys
seznam = []
for s in file(sys.argv[1]):        # Prebirajmo vhodno datoteko po vrsticah.
  s = s.strip()                    # Odrežimo znak za konec vrstice.
  if not s: continue               # Preskočimo morebitne prazne vrstice.
  seznam.append(s)                 # Dodajmo prvotno besedo.
  for i in range(len(s) - 1):      # Dodajmo besede, dobljene z zamenjavami.
    seznam.append(s[:i] + s[i + 1] + s[i] + s[i + 2:])
seznam.sort()                      # Uredimo rezultate po abecedi
for s in seznam: print s           # in jih izpišimo.
```

Kot zanimivost povejmo, da je program vsaj pri naših poskusih (z veliko vhodno datoteko, ki je vsebovala milijon in pol angleških besed) več kot polovico časa porabil za urejanje seznama rezultatov. Izpis pa lahko še malo pospešimo, če zamenjamo zadnjo vrstico s

```
print "\n".join(seznam)
```

Tako program stakne vse besede v en dolg niz, vmes postavi znake za konec vrstice in potem izpiše vse v enem kosu. Seveda pa zato porabimo malo več pomnilnika.

Majhna slabost te rešitve je, da gradi seznam vseh rezultatov v pomnilniku, kar utegne biti problematično, če je vhodna datoteka zelo dolga. V tem primeru lahko rezultate sproti izpisujemo v neko pomožno datoteko in nato uporabimo program `sort`, ki naj bi znal urejati tudi zelo velike datoteke (pri tem si pomaga z dodatnimi pomožnimi datotekami, če je to potrebno). Spodaj je rešitev z `awk`om in ukazno lupino. Pomožno datoteko ustvarimo s programom `mktemp`, ki poišče primerno ime, ki še ne obstaja. Klic `sort` na koncu pomožno datoteko uredi in izpiše, nato pa jo z `rm` še pobrišemo.

```
#!/bin/bash
TMP=`mktemp -t premetavanje.XXXXXXXXXX`
while read BESEDA; do
  echo "$BESEDA" | awk '{
    for (i = 1; i <= length($1); i++) {
      beg = substr($1, 1, i - 1);
      r1 = substr($1, i, 1);
      r2 = substr($1, i + 1, 1);
      end = substr($1, i + 2, length($1));
      print beg r2 r1 end ;
    }
  }' >> "$TMP"
done < "$1"

sort "$TMP"
rm -f "$TMP"
```

R2004.U.2

Primer rešitve v pythonu:

Naloga: str. 16

```
import sys
zadnjiDostop = {}; stSej = 0
for vrstica in file(sys.argv[1]):
  vrstica = vrstica.split(); ip = vrstica[0]; cas = int(vrstica[1])
  if ip not in zadnjiDostop or zadnjiDostop[ip] < cas − 1800: stSej += 1
  zadnjiDostop[ip] = cas
print stSej
```

V razpršeni tabeli zadnjiDostop imamo za vsak naslov IP zapisan čas zadnjega dostopa s tega naslova. Če naletimo na naslov, ki ga v tabeli še ni, ali pa sicer je, vendar je njegov zadnji dostop že prestar, vemo, da se je začela nova seja.

Če so v vhodni datoteki sami različni IPji, bo naša razpršena tabela na koncu hranila praktično že celotno vsebino vhodne datoteke. Če je vhodna datoteka zelo dolga, nas torej lahko skrbi, da bo naš program porabil preveč pomnilnika. Podobno kot pri prejšnji nalogi lahko vhodno datoteko tudi tu najprej uredimo; tako pridejo vrstice, ki se nanašajo na isti IP, skupaj in so tudi urejene po naraščajočem času dostopa. Zdaj je za prepoznavanje novih sej dovolj, če primerjamo po dve zaporedni vrstici.

```
#!/bin/bash
sort −s −k 1,1 $1 | python −c 'import sys
prejsnjiIP = None; stSej = 0
for vrstica in sys.stdin:
  vrstica = vrstica.split(); ip = vrstica[0]; cas = int(vrstica[1])
  if ip != prejsnjiIP or cas > prejsnjiCas + 1800: stSej += 1
  prejsnjiIP = ip; prejsnjiCas = cas
print stSej'
```

Program `sort` lahko razbije vsako vrstico na „polja“, ločena s presledki (ali čim drugim, če mu s parametrom `−t` naročimo drugače); mi bomo s parametrom `−k 1,1` zahtevali, naj za urejanje uporabi le prvo polje, torej naslov IP. Pri vrsticah z enakim naslovom IP pa moramo ohraniti njihov dosedanji medsebojni vrstni red, tako da bodo ostale urejene po naraščajočem času dostopa; potrebujemo torej stabilno urejanje, kar povemo s stikalom `−s`. Druga možnost je, da bi eksplicitno zahtevali tudi urejanje po drugem polju, vendar pa moramo vrednosti v njem gledati kot števila in ne kot nize; lahko bi torej rekli:

```
sort −k 1,1 −k 2,2n $1 | ...
```

R2004.U.3

Lahko si pomagamo z ukazno lupino. Z ukazom `read` berimo vhodno datoteko `$1` po vrsticah. Trenutno vrstico dobimo v spremenljivki `$IME` in jo podamo programu programu `touch`, da ustvari prazno datoteko s tem imenom. Nato s programom `ls` izpišimo imena nastalih datotek; stikalo `−r` zahteva obrnjeni abecedni vrstni red, stikalo `−w 1` pa ga prisili, da izpiše vsako ime v svojo vrstico. Tako torej dobimo seznam nizov, urejen v obrnjenem abecednem vrstnem redu, in ga lahko shranimo v izhodno datoteko `$2`.

Naloga: str. 16

Vse skupaj raje počnimo v nekem pomožnem direktoriju (`$TMPDIR`), da bomo na koncu lažje počistili za sabo (`rm`). Pomožni direktorij ustvarimo s programom `mktemp`, ki sam zamenja niz XX...X s takšnimi znaki, da nastalo ime še ni v rabi; s stikalom −d zahtevamo, naj ustvari direktorij, ne pa navadne datoteke, s stikalom −t pa, naj se nahaja pod pomožnim direktorijem (običajno `/tmp`). Uporabljeno ime izpiše `mktemp` na svoj standardni izhod in ga lahko prestrežemo v spremenljivko $TMPDIR.

```
#!/bin/bash
TMPDIR=`mktemp -t -d urejanje.XXXXXXXXXX`
while read IME; do
  touch "$TMPDIR/$IME" 2> /dev/null
done < "$1"
ls "$TMPDIR" -w 1 -r > "$2"
rm -rf "$TMPDIR"
```

Naloga: str. 16

R2004.U.4

Spodaj je primer rešitve v ukazni lupini. Za začetek s programom `identify` ugotovimo velikost vhodne slike. `identify` izpiše več podatkov o sliki, ločenih s presledki; na tretjem mestu je niz oblike *širina*x*višina*, tako da lahko do širine in višine pridemo s pomočjo `sed`a in `awk`a. Potem z lupininim vgrajenim ukazom `let` izračunajmo višino izhodne slike, da bo razmerje višine in širine enako kot pri vhodni.

Sliko bomo pretvarjali s programom `convert`; s parametrom −resize mu naročimo spremembo velikosti slike, s parametrom −quality pa lahko vplivamo na velikost izhodne datoteke. Če je izhodni format JPEG, ima lahko −quality vrednosti od 0 do 100. Pri manjših vrednostih bo izhodna datoteka manjša, vendar bo slika zato tudi bolj popačena. Do primerne nastavitve pridemo s poskušanjem; če je kvaliteta 100 prevelika, jo zmanjšujemo, dokler ne dobimo dovolj majhne datoteke. Da ne bo trajalo predolgo, jo zmanjšujmo v korakih po 10, nato pa jo po potrebi še povečajmo do največje dopustne vrednosti. Tega bi se lahko lotili tudi kako drugače, npr. z bisekcijo.

Za ugotavljanje velikosti izhodne datoteke uporabimo ukaz ls −l, ki kot peto polje izpiše dolžino datoteke; to lahko izluščimo z `awk`om. Izraz \`ukaz\` se pri izvajanju skripte nadomesti z nizom, ki ga izpiše ukaz ukaz na svoj standardni izhod. Spomnimo se še, da se pri preverjanju pogojev obnaša operator −a kot logični in, operatorja −lt in −gt pa kot primerjalna operatorja < in >. Za računanje aritmetičnih izrazov uporabljamo lupinin vgrajeni ukaz let.

```
#!/bin/bash
# Določimo velikost vhodne in izhodne slike.
Sirina=`identify "$1" | awk '{print $3}' | sed 's/x/ /g' | awk '{print $1}'`
Visina=`identify "$1" | awk '{print $3}' | sed 's/x/ /g' | awk '{print $2}'`
NovaSirina=$3
let NovaVisina=$Visina*$NovaSirina/$Sirina
# Začnimo z največjo možno kakovostjo.
Q=100
convert $1 -resize "$NovaSirina"x"$NovaVisina" -quality $Q $2
# Zmanjšujmo kakovost v korakih po 10, dokler ne dobimo dovolj majhne slike.
while [ $Q -gt 0  -a  `ls -l $2 | awk '{print $5}'` -gt $4 ]; do
  let Q=$Q-10
  convert $1 -resize "$NovaSirina"x"$NovaVisina" -quality $Q $2
done
```

```
# Povečujmo kakovost, dokler je slika še dovolj majhna.
while [ $Q -lt 100 ]; do
  let NovaQ=$Q+1
  convert $1 -resize "$NovaSirina"x"$NovaVisina" -quality $NovaQ $2
  if [ `ls -l $2 | awk '{print $5}'` -gt $4 ]; then
    break; fi   # NovaQ je že prevelika.
  Q=$NovaQ
done

# Pripravimo končno verzijo slike.
convert $1 -resize "$NovaSirina"x"$NovaVisina" -quality $Q $2
```

Viri nalog za leto 2004: topovske krogle, GPS — Boris Gašperin; skrajšanke — Mitja Lasič; naraščajoča števila — Ivo List; tekoče stopnice, Butalci — Mojca Miklavec; čebelica Maja — Mojca Miklavec in Miha Vuk; SMS, izomorfizem — Boštjan Slivnik; otoki — Peter Keše, Jure Leskovec, Janez Brank; ploščice, ruleta, SBN — Janez Brank. Hvala Blažu Novaku za implementacijo rešitev nalog SBN in otoki.

Tekmovanje v poznavanju Unixa 2004 so pripravili: Aleš Košir, Saša Divjak, Boris Gašperin, Rok Papež, Andraž Sraka, Boštjan Müller, Jure Čuhalev, Primož Bratanič, Špela Kraner, Andraž Tori, Primož Peterlin in Miha Tomšič.