26. državno tekmovanje v znanju računalništva (2002)

NALOGE ZA PRVO SKUPINO

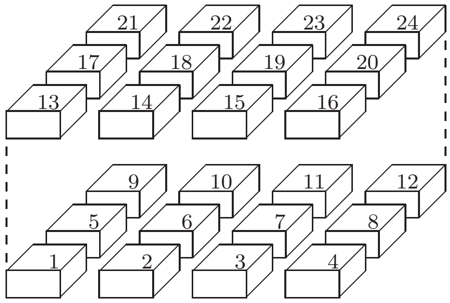
2002.1.1 **Pristanišče**

Rešitev: str. 17

Naročniki večjih količin tovora se običajno odločajo za prevoz tovora z ladjo. Ko zabojniki prispejo v luko, jih morajo tam uskladiščiti, vse dokler ponje ne pride naročnik ali pa jih morajo natovoriti na vlak. V neki luki zabojnike nalagajo enega ob drugega in ko zapolnijo celo vrsto, se lotijo nalaganja v drugo vrsto, nato tretjo in tako dalje. Ko se zapolni cela površina, začnejo zabojnike nalagati še na naslednjo višino v enakem zaporedju kot prej (torej prvi zabojnik na drugi višini pride položen na prvi zabojnik v prvi višini itd.).

Podano imamo število zabojnikov v vsaki vrsti (recimo n), število vrst v vsaki plasti (recimo m) in število plasti (recimo l).

Na spodnji sliki je za primer prikazana postavitev 24 zabojnikov za $n = 4$, $m = 3$, $l = 2$.



Napiši program ali podprogram, ki bo za dane n, m in l ter za neko številko zabojnika izračunal, v kateri plasti in v kateri vrsti je ta zabojnik ter kateri v svoji vrsti je. Vse troje se šteje od 1 naprej in to v takem vrstnem redu, da manjše številke predstavljajo tiste dele skladišča, ki so bili zapolnjeni prej.

Primer: zabojnik s številko 16 na gornji sliki je četrti zabojnik v prvi vrsti druge plasti.

2002.1.2 **Najdaljši cikel v permutaciji**

Rešitev: str. 17

Liliput je majhno mesto z veliko avtobusnimi postajami. Vse proge mestnih avtobusov so krožne. Vsaka postaja pa pripada natančno eni progi, torej nobena dva avtobusa nikoli ne obiščeta iste avtobusne postaje.

Težava je v tem, da Liliputanci objavljajo vozne rede svojih avtobusov na prav poseben način. Avtobusne postaje so oštevilčene s števili od 1 do N, opis vseh prog v njihovem mestu pa predstavijo kot zaporedje N števil.

Za primer vzemimo zaporedje števil (8, 10, 6, 3, 9, 4, 2, 5, 1, 7), ki ga lahko predstavimo kot:

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ 8 & 10 & 6 & 3 & 9 & 4 & 2 & 5 & 1 & 7 \end{pmatrix}$$

Prvo število v zaporedju je 8, kar pomeni, da gre avtobus, ki odpelje s postaje 1, na postajo 8. Osmo število v zaporedju je 5, kar pomeni, da avtobus, ki pripelje na postajo številka 8, nadaljuje svojo pot do postaje številka 5; peto število je 9, zato avtobus nadaljuje pot do postaje 9 in tako naprej, dokler se ne vrne na začetno postajo.

Iz gornjega zaporedja lahko po tem postopku izluščimo tri krožne avtobusne proge:

$$(1, 8, 5, 9),\ (2, 10, 7),\ (3, 6, 4).$$

Tvoja naloga je **napisati program**, ki bo nevajenemu tujcu za poljubno zaporedje števil izpisal vse proge mestnih avtobusov ter mu povedal, koliko postaj ima najdaljša proga.

Na voljo imaš polje velikosti N, ki vsebuje poljubno zaporedje števil, ki ustreza opisu prog mestnih avtobusov:

```
const N = ...;
var Postaje: array [1..N] of integer;
```

2002.1.3 Razbijanje kode

Rešitev: str. 18

Pri uporabi bančne kartice moramo dokazati, da poznamo kodo PIN (osebno identifikacijsko številko). Podobno v deželi PinLand uporabljajo štirimestne *črkovne* kode PIN tudi za dostop do pomembnih podatkov na mreži. Pri razbijanju kode PIN si pomagajmo s poskušanjem: vemo, kakšne so dovoljene kode, in preizkusimo vse možnosti.

Za testiranje imamo na voljo funkcijo

```
function Test(Geslo: string): boolean; external;
```

oziroma

```
extern bool Test(const char* Geslo);
```

Ta funkcija vrne `true`, če je koda pravilna.

Za kodo PIN veljata naslednji omejitvi:

- dolga je točno štiri znake;
- dovoljeni so le znaki `A`, ..., `Z`, torej ravno vse velike črke angleške abecede.

Napiši program, ki bo s poskušanjem odkril pravo kodo PIN in jo izpisal.

2002.1.4 Bencin

Rešitev: str. 18

Janez P. se odpravlja na pot z avtomobilom iz Ljubljane v Bruselj. Na poti bo moral večkrat doliti gorivo. Ker je cena goriva na bencinskih črpalkah različna, ga zanima, kako naj v Bruselj pride kar najceneje. Opiši postopek, ki mu bo pomagal. Na voljo imaš naslednje podatke: dolžina poti (recimo 1200 km); velikost posode za gorivo (recimo 60 l); poraba goriva (recimo 7 litrov na 100 km);[1]

[1] Pri teh številkah je sicer videti, kot da ni treba dotočiti bencina več kot enkrat, vendar utegne biti ceneje, če dotakamo večkrat po malem. Tako ali tako so te konkretne številke mišljene le za oporo pri razmišljanju, sicer pa nas zanima postopek, ki bi deloval za poljubne vrednosti teh podatkov.

število črpalk ob poti; za vsako črpalko pa še njen položaj na poti (oddaljenost od Ljubljane v kilometrih) in ceno bencina na njej. (Za cene bencina si misli, da so vse izražene v istih denarnih enotah — s pretvarjanjem med valutami se ti ni treba ubadati.)

Janez P. pot začne s prazno posodo, prva črpalka pa je že takoj na začetku poti. **Opiši postopek**, ki lahko za poljubno razporeditev črpalk in cene bencina ugotovi, kje in koliko goriva je treba doliti, da ga ne bo nikjer zmanjkalo, obenem pa bomo za gorivo porabili čim manj denarja.

NALOGE ZA DRUGO SKUPINO

2002.2.1 Kodiranje

Rešitev: str. 21

Recimo, da se lahko v besedilih, s katerimi imamo opravka, pojavijo le črke od a do h. V tipičnem besedilu se ne pojavljajo vse črke enako pogosto, pač pa so pogostosti pojavljanja posameznih črk takšne:

Črka	a	b	c	d	e	f	g	h
Pogostost	12,5 %	6,25 %	25 %	6,25 %	25 %	6,25 %	12,5 %	6,25 %

Ta tabela pove, da se v tisoč črkah značilnega besedila pojavi črka a 125-krat, črka c 250-krat itd. Primer tovrstnega (a malo krajšega) besedila bi bil npr.:

ccfdcceabbeeegceabccefhaageaaahgbegceefbcecacfaddeegeeeecefghgcecdcccgeghahccdcg

Opiši, kako bi z ničlami in enicami predstavil (kodiral) posamezno od teh črk, da bi bilo tipično besedilo zapisano čim krajše (vendar pa bi se ga še vedno dalo dekodirati nazaj v prvotno obliko)?

Primer takšnega kodiranja za neko drugo vrsto besedil: recimo, da bi bila naša besedila lahko sestavljena le iz štirih črk, p, q, r in s, pri čemer bi se p pojavil v 90 % primerov, q v 8 %, r in s pa v 1 % primerov. Potem bi bilo smiselno p predstaviti s kodo 0, q s kodo 10, r s kodo 110 in s s kodo 111.

2002.2.2 Prijatelji in sovražniki

Rešitev: str. 22

Jurij W. Grm mlajši je sila pomemben in vpliven mož, ki pozna veliko ljudi. Pozna jih celo toliko, da sploh ne ve več, kateri so njegovi pravi prijatelji in kateri pravi sovražniki. Jurij je najel skupino detektivov, ki je za vsakega od ljudi, ki jih kakorkoli pozna (tudi posredno), naredila seznam *neposrednih prijateljev* in *neposrednih sovražnikov*.

Jurija zanima, kdo so njegovi pravi prijatelji in kdo pravi sovražniki. Pri tem veljajo naslednje definicije:

Jurijevi *prijatelji* so tisti, ki so bodisi njegovi neposredni prijatelji, neposredni prijatelji njegovih prijateljev ali pa neposredni sovražniki njegovih sovražnikov.

Njegovi *sovražniki* pa so tisti, ki so njegovi neposredni sovražniki, neposredni prijatelji kakšnega sovražnika ali neposredni sovražniki Jurijevih prijateljev.

Jurijev *pravi prijatelj* je vsakdo, ki je njegov prijatelj, ne pa tudi sovražnik. *Pravi sovražnik* pa je vsakdo, ki je njegov sovražnik, ne pa tudi prijatelj.

Opomba: te definicije dopuščajo tudi, da je Jurij sam svoj sovražnik. V tem primeru so vsi, ki jih posredno ali neposredno pozna, hkrati njegovi prijatelji in sovražniki. Tudi to se pač lahko zgodi.

Na voljo imaš tabelo s podatki o neposrednih prijateljih in neposrednih sovražnikih vsakega človeka:

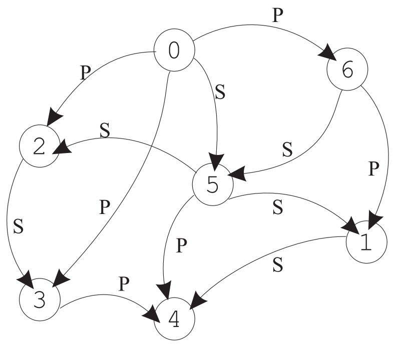
```
const N = . . . ;
type KdoJeKdo = (Neznanec, Prijatelj, Sovrag);
var Odnos: array [0..N − 1, 0..N − 1] of KdoJeKdo;
```

```
#define N . . .
typedef enum { Neznanec, Prijatelj, Sovrag } KdoJeKdo;
KdoJeKdo Odnos[N][N];
```

Polje Oseba je velikosti $N \times N$ in vsebuje podatke o neposrednih prijateljstvih in neposrednih sovraštvih. Vrednost Oseba[i, j] pove, kaj je oseba j osebi i. Poznanstva so enostranska, torej sploh ni nujno, da velja Oseba[i, j] = Oseba[j, i]. Tako lahko Jurij misli, da je Vlado njegov (neposredni) prijatelj, Vlado pa ima Jurija za svojega (neposrednega) sovražnika. Jurij ima v tej tabeli številko 0.

Opiši postopek, s katerim bi Jurij ugotovil, kdo so njegovi pravi prijatelji in kdo pravi sovražniki. Lahko si pomagaš z gornjo deklaracijo tabele Odnos, ni pa treba pisati izvorne kode v kakšnem konkretnem programskem jeziku.

Primer (povezava P oz. S od a do b pomeni, da je b a-jev neposredni prijatelj oz. neposredni sovražnik):



V tem primeru so 2, 6 in 1 Jurijevi pravi prijatelji, 5 pravi sovražnik, osebi 3 in 4 pa mu nista niti prava prijatelja niti prava sovražnika.

Opomba: na tej sliki lahko od Jurija do vseh oseb pridemo že prek samo dveh poznanstev, kar pa v splošnem ne velja — do nekaterih oseb včasih potrebujemo več kot dva koraka (ali pa do njih sploh ne moremo priti).

2002.2.3 Razbijanje gesel

Rešitev: str. 22

Radi bi vdrli v nek sistem, zaščiten z geslom. Pri razbijanju gesel si pomagamo s poskušanjem. Vemo, kakšna so dovoljena gesla, in preizkusimo vse možnosti.

Za testiranje imamo na voljo funkcijo

```
function Test(Geslo: string): boolean; external;
```

oziroma

```
extern bool Test(const char* Geslo);
```

Ta funkcija vrne `true`, če je koda pravilna, drugače pa `false`.

Za geslo veljajo naslednje omejitve:

- dolgo je najmanj tri znake in največ osem znakov;
- dovoljeni znaki so `A`, ..., `Z`, `a`, ..., `z` in `0`, ..., `9` (velike in male črke angleške abecede ter števke od 0 do 9);
- vsako geslo mora vsebovati vsaj eno črko in vsaj eno števko.

Naloga:

- Napiši program, ki bo s poskušanjem poiskal pravo geslo in ga izpisal.
- Oceni, kakšno je navečje, najmanjše in pričakovano število poskusov za naključno geslo.
- Ali je tako razbijanje gesel smiselno (časovno sprejemljivo)? Odgovor utemelji.

2002.2.4 TiVo

Rešitev: str. 24

Pri gledanju televizijskega programa bi si včasih želeli, da bi lahko začasno ustavili program med kakšnim opravkom, potem pa nadaljevali z gledanjem, ne da bi kaj zamudili — pa tudi kakšno posebno zanimivo sceno bi radi v miru ponovno pogledali. Nastalo zamudo bi lahko nadoknadili ob prvi primerni priložnosti, na primer s hitrim preskakovanjem reklam.

Za silo si lahko pomagamo z videorekorderjem, vendar ta metoda ne deluje dobro za spremljanje živega programa. Ko bi le lahko imeli isto kaseto speljano skozi dva snemalnika...

Na tržišču so se že pojavili snemalniki (pravzaprav računalniki), ki obvladajo tudi tovrstne zahteve gledalcev. Namesto da bi snemali na trak, zapisujejo sliko na računalniški disk; zmorejo pa tudi snemati program iz sprejemnika in hkrati predvajati katerikoli del že posnetega programa, tako najbolj svežega skoraj brez zamika, kot tudi starejšega z zamikom. Ker je velikost diska omejena na nekaj ur, se najstarejši deli posnetka samodejno brišejo, ko začne zmanjkovati prostora za nov posnetek.

Denimo, da uspemo zapisati eno televizijsko sličico na en diskovni blok. Sličice si pri snemanju in pri normalnem predvajanju sledijo s hitrostjo 25 na sekundo. Diskovni bloki (sličice) so oštevilčeni od 0 do 300 000, kar zadošča za dobre tri ure posnetka. Ob zagonu sistema so vsi bloki prazni, snemanje in predvajanje se začneta takoj (istočasno) in potem potekata neprekinjeno.

Določi in opiši podatkovno strukturo, ki ti bo pomagala pri učinkovitem določanju oz. iskanju zaporednih sličic tako, da bo omogočeno stalno snemanje (shranjevanje novih sličic, prihajajočih iz sprejemnika), brisanje oziroma pozabljanje najstarejših sličic, ter hkrati s snemanjem tudi predvajanje poljubnega dela posnetega programa v načinih: normalna hitrost predvajanja, zamrznjena slika in pospešeno predvajanje z 10-kratno hitrostjo naprej ali nazaj.

Napiši tudi naslednje **tri podprograme**, potrebne za upravljanje s takšnim snemalnikom:

- za snemalni del:

– podprogram KamShranitiNovo, ki ga bo sprejemniški del klical 25-krat v sekundi (za vsako novo sprejeto sličico), podprogram pa bo vsakokrat vrnil številko diskovnega bloka, na katerega naj sprejemniški del shrani pravkar sprejeto sličico;

- za predvajalni del:

  – podprogram KateroSlikoPrikazati, ki ga bo predvajalni del elektronike klical 25-krat v sekundi; podprogram mora vsakokrat vrniti številko diskovnega bloka, na katerem je zapisana sličica, ki mora biti prikazana na televizijskem ekranu za naslednjo petindvajsetinko sekunde. Na zaporedje sličic naj vpliva zadnja nastavitev, določena s klicem tvojega podprograma IzberiNacinPredvajanja;

  – podprogram IzberiNacinPredvajanja, ki ga aktivira gledalec s pritiskom na tipko. Trenutna nastavitev naj vpliva na zaporedje sličic, kot jih mora vračati tvoj podprogram KateroSlikoPrikazati. Kot argument dobi celo število, ki pomeni:

0	pavza — mirujoča slika, vsakokrat vidimo isto sličico;
1	normalno predvajanje naprej – sličice naj si vrstijo v istem zaporedju, kot so bile posnete;
10	hitro predvajanje naprej z desetkratno hitrostjo — prikaže naj se le vsaka deseta sličica, vmesne izpustimo;
-10	hitro predvajanje nazaj z desetkratno hitrostjo, torej proti vedno starejšim sličicam (dokler ne naletimo na najstarejše).

Predpostaviš lahko, da se ne bo hkrati izvajalo po več tvojih podprogramov; torej, medtem ko se eden izvaja, ga drugi ne more prekiniti, pač pa se lahko začne izvajati šele, ko se prvi konča.

Obnašanje ob robnih pogojih (na primer ko pri hitrem predvajanju naletimo na najstarejše ali na najnovejše posnetke) ni predpisano, vendar se spodobi, da se program odloči za varianto, ki bo gledalca čim manj presenetila. **Napiši**, kako se tvoja rešitev obnaša v takšnih primerih.

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 slabe 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 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. Programi naj bodo napisani v programskem jeziku Pascal, C ali C++, mi pa jih bomo preverili z 32-bitnimi prevajalniki FreePascal in GNU C++. Za delo lahko uporabiš `turbo` (Turbo Pascal), `fp` (FreePascal), `tc` (Turbo C), ali `gcc/g++` (GNU C/C++ — command line compiler). Ves potreben softver lahko najdeš na `c:\Programi` ter v meniju `Start` pod `Programs` in `Prevajalniki`.

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.

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).

Za oddajo rešitve uporabi enega od naslednjih ukazov:

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

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 obvestilo o tem, ali je program na testne primere 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. Če si oddal N programov za to nalogo in je najboljši med njimi pravilno rešil M (od desetih) testnih primerov, dobiš pri tej nalogi $\max\{0, 10M - 3(N-1)\}$ točk. Z drugimi besedami: vsak pravilno rešen testni primer ti prinese 10 točk, za vsako oddajo (razen prve) pri tej nalogi pa 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.

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 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. {PoskusnaNaloga}
```

```
#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;
}
```

NALOGE ZA TRETJO SKUPINO

2002.3.1 Limuzine `limo.in`, `limo.out`

Rešitev: str. 25

Podjetje LepeLimuzine ima vozni park limuzin, ki so na voljo poslovnežem, ki se jim mudi na sestanke. Zaradi narave posla šoferji veliko časa presedijo na parkiriščih; zaradi visokih stroškov parkiranja so se v podjetju odločili, da bodo v vsako limuzino vgradili računalniški sistem, ki bi jim znal svetovati, kam naj gredo parkirat.

Napiši program, ki bo prebral ceno ure vožnje, trajanje sestanka in seznam parkirišč z njihovimi cenami parkiranja in oddaljenostmi od kraja sestanka. Program naj ugotovi, kje je treba parkirati (in za koliko časa), da bodo skupni stroški (vožnja + parkirnina) najmanjši in da bo limuzina ravno ob koncu sestanka spet prišla nazaj na prvotno mesto.

Vhodna datoteka vsebuje v prvi vrstici ceno ure vožnje (v SIT) in trajanje sestanka (v minutah). Nato sledi za vsako parkirišče po ena vrstica, v kateri sta dve števili; prvo pove ceno ure parkiranja na tem parkirišču (v SIT), drugo pa pove, koliko minut potrebujemo, da se pripeljemo od kraja, kjer smo odložili potnika, do tega parkirišča (ali pa nazaj — predpostavimo, da traja vožnja v nasprotno smer enako dolgo). Na koncu je vrstica, ki vsebuje dve ničli. Parkirišč ni več kot 1000; za ceno vožnje in vse cene parkiranja velja, da so vsaj 1 in največ 10000 SIT/h; čas vožnje do posameznega parkirišča je vsaj 1 minuto in največ 1000 h, ista omejitev pa velja tudi za trajanje sestanka. Vse cene in trajanja so cela števila.

V *izhodno datoteko* izpiši številko parkirišča, na katerem moramo parkirati, in trajanje našega parkiranja na njem (v minutah). Številka 1 pomeni prvo parkirišče, 2 drugo in tako naprej. Če so parkirišča predraga ali predaleč in je zato bolje kar voziti po mestu, ne da bi kje parkirali, izpiši dve ničli. Če bi se dalo do najmanjših stroškov priti na različne načine (npr. z različnimi parkirišči), je vseeno, katerega opišeš. Če parkiraš na nekem parkirišču, moraš tam ostati vsaj eno minuto.

Primer vhodne datoteke:

```
500 60
200 10
700 20
0 0
```

Ustrezna izhodna datoteka:

```
1 40
```

2002.3.2 Uvrstitve tekmovalcev `tekma.in`, `tekma.out`

Rešitev: str. 26

Skupina kolesarjev hkrati začne krožno dirko. Vsakič, ko kateri izmed njih prevozi štartno-ciljno črto, štarter vtipka njegovo štartno številko, računalnik pa jo zapiše v datoteko. Dirka se konča, ko prvi tekmovalec prevozi predpisano število krogov — preostali le še končajo trenutni krog (tudi to se vpiše v datoteko).

Napiši program, ki prebere datoteko in izpiše uvrstitve tekmovalcev. (Kriterij za razvrstitev je ta, da so višje uvrščeni tisti tekmovalci, ki so prevozili več krogov, med takimi, ki so prevozili enako število krogov, pa tisti, ki so jih prevozili prej.) Če neki tekmovalec ne konča nobenega kroga, pomeni, da ni uvrščen in ga tudi v rezultatih ne sme biti. Upoštevaj, da lahko tekmovalci prehitevajo eden drugega tudi za več krogov.

Vhodna datoteka vsebuje v prvi vrstici število tekmovalcev (recimo N, ki je celo število, $1 \leq N \leq 2000$). V naslednjih vrsticah (ki jih ni več kot milijon) so štartne številke tekmovalcev v takšnem vrstnem redu, v kakršnem so vozili čez štartno-ciljno črto; v vsaki vrstici je po ena, na koncu pa pride vrstica, ki vsebuje število 0. (Štartne številke tekmovalcev so cela števila od 1 do N.)

V *izhodno datoteko* izpiši štartne številke tekmovalcev po uvrstitvah, vsako v svojo vrstico: v prvo vrstico zmagovalca, v drugo drugega in tako naprej.

Primer vhodne datoteke:

```
4
1
2
3
4
2
4
3
4
2
1
4
1
2
3
0
```

Pripadajoča izhodna datoteka:

```
4
2
1
3
```

2002.3.3 Število vsot `vsote.in`, `vsote.out`

Rešitev: str. 27

Zanima nas, na koliko načinov lahko naravno število N (naravna števila so tista, ki so cela in večja od nič) zapišemo kot vsoto K naravnih števil (K je celo število, za katero velja $1 \leq K \leq N$). Pri tem nam vrstni red seštevancev v vsoti ni pomemben (če lahko eno izražavo dobimo iz druge tako, da le premešamo seštevance, ju obravnavamo kot eno in isto in ne štejemo vsake posebej). Tudi zapis $N = N$ velja kot izražava N-ja kot vsote enega samega seštevanca (za $K = 1$ bi torej dobili odgovor 1, ker lahko N samo na ta edini način izrazimo kot vsoto enega pozitivnega celega števila).

Napiši program, ki prebere N in K ter izračuna, koliko je teh načinov:

- *Vhodna datoteka* vsebuje v prvi vrstici dve celi števili, najprej N in nato K, ločeni z enim presledkom. Veljalo bo $1 \leq K \leq N \leq 100$.
- V *izhodno datoteko* izpiši število, ki pove, na koliko načinov lahko N zapišemo kot vsoto K seštevancev.

Primer vhodne datoteke:

```
10 3
```

Pripadajoča izhodna datoteka:

```
8
```

Število 10 lahko namreč kot vsoto treh pozitivnih celih števil zapišemo na osem načinov:

$$\begin{aligned} 10 &= 8+1+1 = 7+2+1 = 6+3+1 = 6+2+2 \\ &= 5+4+1 = 5+3+2 = 4+4+2 = 4+3+3. \end{aligned}$$

2002.3.4 Sestanki `sestanki.in`, `sestanki.out`

Rešitev: str. 28

Jurij W. Grm mlajši bi rad sestankoval. Ker pa so vsi, ki naj bi bili prisotni na sestanku, zelo zaposleni z drugimi sestanki, ni enostavno izbrati takega termina, ki bi ustrezal vsem. Zato je za pomoč prosil svojega svetovalca za informatiko — tebe, da mu **napišeš program**, ki mu bo pomagal pri določitvi ustreznega časa za sestanek.

Vhod: vhodni podatki programa so urniki vseh oseb, ki bodo prisostvovale na sestanku, in želeno časovno obdobje, v katerem si Jurij želi sestanka. Oblika vhodne datoteke je naslednja:

$$\begin{array}{lll} f & t & d \\ s_{11} & d_{11} & \\ s_{12} & d_{12} & \\ . & . & . \\ 0 & 0 & \\ s_{21} & d_{21} & \\ . & . & . \\ 0 & 0 & \\ . & . & . \\ 0 & 0 & \end{array}$$

Števili f in t predstavljata meji časovnega intervala, znotraj katerih je možen sestanek (sestanek se ne sme začeti prej kot ob času f in se mora končati najkasneje do časa t). Število d je trajanje sestanka (v minutah). Nato so za vsakega sodelavca na sestanku navedeni drugi sestanki, zaradi katerih v določenih obdobjih nima časa. Vrednost s_{ij} je čas, ko se osebi i prične j-ti sestanek, d_{ij} pa je trajanje tega sestanka (v minutah). Seznam sestankov posameznega sodelavca se konča z vrstico, ki vsebuje dve ničli, čisto na koncu datoteke pa je še dodatna vrstica z dvema ničlama. Sodelavcev je največ sto in vsak od njih ima največ sto drugih sestankov. Vsi časi (f, t, s_{ij}) so navedeni kot število minut, ki so potekle od polnoči 6. aprila 2002. Vsi časi so med 6. aprilom 2002 in 1. januarjem 5500. Predpostaviš lahko, da za prehajanje z enega sestanka na drugega ljudje ne porabijo nič časa.

Izhod programa naj bo ena sama vrstica: število minut od polnoči 6. aprila 2002, ko naj se sestanek prične, tako da bodo lahko na celotnem sestanku sodelovali vsi povabljeni in da bo sestanek kar se da kmalu. Če takšen termin ne obstaja, naj program izpiše „`SESTANEK NI MOZEN`“ (brez teh narekovajev).

Primera dveh vhodnih datotek:	Pripadajoči izhodni datoteki:

`10 30 5`
`0 15`
`25 10`
`0 0`
`10 10`
`0 0`
`0 0`
	`20`

`10 30 6`
`0 15`
`25 10`
`0 0`
`10 10`
`0 0`
`0 0`
	`SESTANEK NI MOZEN`

2002.3.5 Produkt števil `produkt.in`, `produkt.out`

Rešitev: str. 30

Dano je zaporedje celih števil: $\langle a_1, a_2, \ldots, a_{n-1}, a_n \rangle$; vsako od njih je ali enako 0 ali pa je oblike $\pm 2^k$ za kakšno nenegativno celo število k, $0 \leq k \leq 30$.

Napiši program, ki poišče tako podzaporedje $\langle a_i, a_{i+1}, \ldots, a_{j-1}, a_j \rangle$ (za neka i in j, pri katerih velja $1 \leq i \leq j \leq n$), da bo produkt števil $a_i \cdot a_{i+1} \cdot \ldots \cdot a_{j-1} \cdot a_j$ največji.

V prvi vrstici *vhodne datoteke* bo podano število n (zanj velja $1 \leq n \leq 100\,000$), v naslednji a_1, v naslednji a_2, itd., v zadnji (($n + 1$)-vi vrstici) pa a_n.

V *izhodno datoteko* napiši, v eni sami vrstici, števili i in j (ločeni s presledkom), pri katerih je produkt $a_i \cdot a_{i+1} \cdot \ldots \cdot a_j$ največji. Če je možnih več enako dobrih parov (i, j), je vseeno, katerega izpišeš.

Dva primera vhodnih datotek: Možni pripadajoči izhodni datoteki:

`5`
`4`
`-4`
`8`
`-2`
`-8`
	`1 4`

`5`
`4`
`0`
`4`
`0`
`8`
	`5 5`

2002.3.6 **Prüferjev kod** `prufer.in`, `prufer.out`

Rešitev: str. 32

Drevo je graf, ki ga sestavljajo *vozlišča* (ali *točke*) in neusmerjene *povezave* med njimi, pri tem pa povezave nikoli ne tvorijo ciklov (pri sprehajanju iz neke točke ne moremo priti nazaj v isto točko drugače kot tako, da se obrnemo in gremo nazaj po isti poti, po kateri smo prišli), je pa mogoče iz vsake točke priti po enem ali več korakih do vsake druge (temu rečemo, da je graf *povezan*). Če sta dve vozlišči povezani s povezavo, pravimo, da sta *soseda*. Vozlišče, ki ima enega samega soseda, imenujemo *list*.

Profesor Hrast je bil v mladih letih zelo navdušen nad drevesi. Iznašel je postopek, s katerim je lahko drevo z N vozlišči zakodiral v zaporedje $N-2$ števil. (Dobljeno zaporedje je poimenoval „Prüferjev kod“ začetnega drevesa.) To je naredil takole:

1. Poljubno je oštevilčil vsa vozlišča drevesa s števili od 1 do N; vsakemu je pripisal drugo številko.
2. Potem je ponavljal naslednji postopek, dokler mu ni ostala le ena povezava:
   (a) poiskal je list z največjo številko,
   (b) ga izbrisal iz drevesa
   (c) in zapisal številko njegovega soseda.

Ta postopek se vedno izteče in nam da za različna drevesa vedno tudi različna zaporedja $N-2$ števil. Iz dobljenega zaporedja lahko vedno rekonstruiramo prvotno drevo.

Leta so minila, profesor se je postaral in v tem času so njegova drevesa precej zrastla. Ročno iskanje zaporedja postaja vedno bolj zamudno. Pomagaj profesorju in **napiši program**, ki bo prebral opis drevesa, izračunal njegov Prüferjev kod in izpisal tako dobljeno zaporedje $N-2$ števil. (Pravzaprav mora izvesti le korak 2, ker bo dobil vozlišča že oštevilčena.)[2]

Vhodna datoteka vsebuje v prvi vrstici število vozlišč, recimo mu N. Sledi za vsako povezavo po ena vrstica, v njej pa sta dve števili, ki povesta, kateri

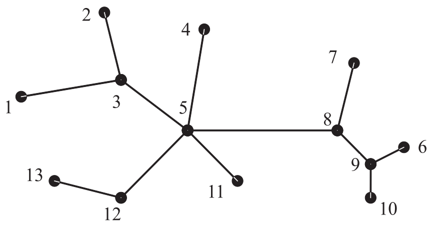
[2]Zanimiva naloga je tudi sestaviti postopek za dekodiranje, torej takega, ki bo znal zaporedja $N-2$ števil pretvarjati v drevesa (izpisal bi na primer seznam parov števil, ki predstavljajo povezave drevesa).

dve vozlišči povezuje ta povezava. Drevo z N vozlišči ima vedno $N-1$ povezav; vozlišča so oštevilčena s celimi števili od 1 do N. Povezave so neusmerjene, zato je lahko na primer povezava med vozliščema 12 in 34 navedena bodisi kot `12 34` ali pa kot `34 12`. Predpostaviš lahko, da velja $3 \le N \le 10000$.

V *izhodno datoteko* izpiši zaporedje $N-2$ števil, ki ga dobiš po zgoraj opisanem postopku iz drevesa, ki si ga prebral iz vhodne datoteke. Vsako število naj bo v svoji vrstici.

Primer: drevo na spodnji sliki nam dá naslednje zaporedje:

$$\langle 12, 5, 5, 9, 8, 9, 8, 5, 5, 3, 3 \rangle.$$



Ena od možnih vhodnih datotek za to drevo:

```
13
5 4
5 11
3 1
8 9
13 12
5 8
12 5
2 3
9 10
8 7
9 6
3 5
```

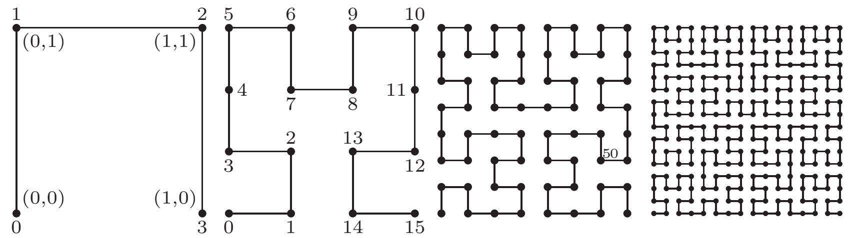
Ustrezna izhodna datoteka:

```
12
5
5
9
8
9
8
5
5
3
3
```

2002.3.7 **Hilbertova krivulja** `hilbert.in`, `hilbert.out`

Rešitev: str. 35

Dano je zaporedje krivulj, recimo jim $\mathcal{H}_1, \mathcal{H}_2, \ldots$. Krivulja $\mathcal{H}_k$ poteka po karirasti mreži velikosti $2^k \times 2^k$ in obišče vsako točko v tej mreži natanko enkrat. Prvih nekaj krivulj je prikazanih na spodnji sliki. Vidimo, da lahko do $\mathcal{H}_{k+1}$ pridemo tako, da vzamemo štiri kopije krivulje $\mathcal{H}_k$, jih postavimo v kvadrat, spodnji dve zavrtimo drugo proti drugi in jih nato vse štiri povežemo.



$\mathcal{H}_1$ $\mathcal{H}_2$ $\mathcal{H}_3$ $\mathcal{H}_4$

Pri določenem k lahko s pomočjo krivulje $\mathcal{H}_k$ točke na mreži $2^k \times 2^k$ oštevilčimo: začnimo v spodnjem levem kotu, tisti točki pripišimo številko 0, nato pa se sprehajajmo vzdolž krivulje in pripisujmo točkam po vrsti vse višja cela števila. Končamo v točki v spodnjem desnem kotu, ki dobi številko $4^k - 1$.

Točke lahko opišemo tudi s koordinatami; tiste na spodnji vrstici imajo y-koordinato 0, tiste tik nad njimi 1, tiste v vrhnji vrstici pa $2^k - 1$. Podobno imajo tiste v najbolj levem stolpcu x-koordinato 0, tiste tik desno od njih 1, tiste v najbolj desnem stolpcu pa $2^k - 1$.

Primeri: na krivulji $\mathcal{H}_2$ ima točka s koordinatama $(1,2)$ številko 7, tista s koordinatama $(2,1)$ pa številko 13. Na krivulji $\mathcal{H}_3$ ima točka s koordinatama $(6,2)$ številko 50.

Napiši program, ki iz *vhodne datoteke* prebere cela števila k, a in b, ki so vsa v prvi vrstici, ločena s po enim presledkom. Vedno velja $1 \leq k \leq 15$, $0 \leq a < 2^k$ in $0 \leq b < 2^k$. V *izhodno datoteko* naj tvoj program izpiše številko, ki jo na krivulji $\mathcal{H}_k$ dobi točka s koordinatama (a,b) (a je x-koordinata, b pa y-koordinata).[3]

Trije primeri vhodnih datotek:	Pripadajoče izhodne datoteke:
`1 0 0`	`0`
`3 6 2`	`50`
`10 1023 0`	`1048575`

2002.3.8 Slovar `slovar.in`, `slovar.out`

Rešitev: str. 36

Polde se je že v svoji rani mladosti zasvojil s televizijo. Vsak dan je od jutra do večera presedel pred ekranom in se navduševal nad vsako še tako brezzvezno oddajo. Tako je bilo tudi ob nedeljah zvečer, vse dokler niso na Pok TVju uvedli novega besednega kviza. Kviz je Poldeta tako prevzel, da se je odločil v njem tudi sam sodelovati, ker pa se zaradi skromnega televizijskega besednega zaklada ni mogel potegovati za glavne nagrade, se sedaj obrača na vas, da mu spišete program, s katerim si bo lahko spomagal pri goljufanju.

Kviz zahteva, da na osnovi danega besedišča opišete najkrajši način, kako iz ene besede priti do druge, pri čemer lahko v vsakem koraku spremenite le eno črko, vsaka vmesna beseda pa mora tudi dejansko obstajati v slovarju. Možne besede v tem sprehodu morajo torej vedno imeti enako število črk.

Za primer vzemimo naslednji slovar (besede niso nujno urejene — niti po številu znakov niti po abecednem redu):

kot	*pot*	*ris*	*kapa*	*kepa*	*koča*	*milo*	*peka*
pesa	*reka*	*repa*	*roka*	*solo*	*šapa*	*šepa*	*teka*

Zanima nas, kakšen je najkrajši sprehod od besede *kapa* do *pesa*. Dobimo sprehode:

kapa → *kepa* → *repa* → *reka* → *peka* → *pesa*;
kapa → *šapa* → *šepa* → *repa* → *reka* → *peka* → *pesa*;
kapa → *kepa* → *repa* → *reka* → *teka* → *peka* → *pesa*; …

Najkrajši sprehod je očitno prvi, ki zahteva 5 korakov (število puščic „→“).

Napiši program, ki na podlagi danega slovarja ter parov začetnih in končnih besed za vsak tak par poišče najkrajši sprehod od začetne do končne besede. Kot rezultat naj izpiše le število korakov ali pa „`Ni prehoda`“.

Vhod: v vhodni datoteki je najprej podan slovar; v prvi vrstici je celo število N, ki pove, koliko je besed v slovarju (vsaj 1, največ 10000). V naslednjih N vrsticah sledijo besede, vsaka v svoji vrstici; posamezne besede so dolge

[3] Zanimiva naloga je tudi pretvarjanje v nasprotno smer, torej iz zaporedne številke v koordinate.

vsaj 2 in največ 10 znakov. Vse besede skupaj so dolge največ 55000 znakov. Parov besed, ki se razlikujejo le v eni črki, je največ 55000 (če parov $\{a, b\}$ in $\{b, a\}$ ne štejemo vsakega posebej) in pri tem nobena beseda ne nastopa v paru z več kot 30 drugimi. Vse besede so vedno podane le z malimi črkami angleške abecede, drugih znakov v besedah ni. — Sledi vrstica, ki vsebuje le celo število M, ki pove, koliko parov besed sledi ($1 \leq M \leq 10$). V naslednjih M vrsticah so pari besed „⟨*začetna beseda*⟩ ⟨*končna beseda*⟩“ (obe v eni vrstici, ločeni s presledkom). Obe besedi (začetna in končna) sta vedno vsebovani v slovarju.

Izhod: za vsak par besed v samostojni vrstici izpiši dolžino najkrajše poti oziroma niz „`Ni prehoda`“ (brez teh narekovajev), če poti med njima sploh ni.

Primer dveh vhodnih datotek:	Pripadajoči izhodni datoteki:
6 kot pot pet ris kapa kepa 3 kapa kepa kot pet kot ris	1 2 Ni prehoda
2 pet pot 1 pet pot	1

LETO 2002, TEKMOVANJE V POZNAVANJU UNIXA

Navodila

Naloge boste pisali v tekstovne datoteke z urejevalnikom po svoji izbiri. Za pošiljanje rešitve naloge številka N uporabite skript `submit`N, ki mu podate ime datoteke z rešitvijo. Če rešitev nikakor ne ustreza, vam bo program javil „`Naloga je zavrnjena`“; če popolnoma ustreza, dobite 10 točk. Če rešitev ni popolnoma ustrezna, lahko dobite manj kot 10 točk. Vsako nalogo lahko pošljete poljubno mnogokrat. Seštevek svojih točk lahko pregledate z ukazom `score`. V primeru enakega skupnega števila točk bo komisija ocenjevala tudi razumljivost rešitve.

2002.U.1 Delo skrbnice računalniških sistemov Metke je široko. Ena izmed njenih nalog je dodeljevanje internetnih naslovov računalnikom, za katere skrbi. Da bi si olajšala delo, je Metka sestavila program, ki pove, ali se izbrano podomrežje prekriva z že dodeljenim podomrežjem.

Rešitev: str. 39

Podomrežje je opisano s skupino štirih polj, ločenih s piko. Polja so opisana s števili od vključno 0 do vključno 255 ali pa z intervalom, ki vključuje ta števila. Interval je označen z znakom minus (`-`) med dvema številoma. Namesto kateregakoli polja je lahko znak zvezdica (`*`), ki označuje vsa števila z intervala.

Primeri tako opisanih podomrežij so:

```
192.168.1.1
192.168.0.1-3
0-255.*.255.*
```

Pomagajte Metki in sestavite program, ki v ukazni vrstici sprejme kot parametra dve podomrežji, opisani na zgornji način.

Program naj vrne izhodno vrednost (*exit status*):

0, če območji nimata skupnih naslovov (podomrežji sta tuji), kot na primer podomrežji 192.168.1–30.64 in 192.168.31–35.0;

1, če imata podomrežji natanko en skupni naslov (presek množic je natanko en element), kot na primer 10.0.0.* in 10.0.0.1;

2, če imata območji več kot en skupni naslov, kot na primer 10.*.*.* in 10.0–12.3.4;

Privzameš lahko, da so podatki pravilni.

Rešitev: str. 40

2002.U.2

Vrstice v besedilnih datotekah so v sistemih Unix zaključene z znakom LF. V nekaterih sistemih, na primer MS-DOS in Windows, so vrstice zaključene z dvema zaporednima znakoma CR in LF. Napiši program, ki bo v datoteki, podani z imenom v ukazni vrstici, zamenjal vse konce vrstic iz zaporedja CR LF v LF. Privzameš lahko, da znaka CR in LF vselej nastopata v paru.

V pomoč: znak CR je desetiško 13, predstavljen pa je tudi kot `^M` ali `\r`, znak LF je desetiško 10 ali `^J` ali `\n`.

Program naj se izvede takole:

```
naloga2 ime_datoteke
```

Ko se program konča, morajo biti zaključki vrstic v datoteki zamenjani. Na disku ne smejo ostati morebitne pomožne datoteke.

Rešitev: str. 40

2002.U.3

V računalniku hkrati teče več procesov. Vsi izvirajo iz procesa `init`. Vsak proces pa ima lahko več sinov. Procesi tako tvorijo drevesno strukturo. Procesna veriga so procesi od procesa `init` pa do zadnjega procesa, ki je brez potomca. Napiši program, ki globino najdaljše procesne verige vrne kot izhodno vrednost.

Primer:

```
init-+
     |-cron
     |-gpm
     |-httpd---10*[httpd]
     `-in.identd---in.identd---3*[in.identd]
```

Najdaljša veriga je dolžine 4.

Rešitev: str. 41

2002.U.4

Sčasoma se je nabralo več datotek, v katerih so zapisani statistični podatki. Ker je za datoteke skrbelo več oseb, ki so datoteke poimenovali po svoje, imena niso sistematično urejena in iz njih ni razvidno, kateremu časovnemu obdobju pripadajo.

Da bi datoteke lahko kronološko uredili, potrebujemo program, ki primerja dve datoteki in pove, katera je starejša.

Napišite program, ki prejme kot parameter dvoje imen in vrne kot izhodno vrednost:

0 — če sta datoteki enako stari,
1 — če je prva datoteka starejša od druge,
2 — če je druga datoteka starejša od prve,
3 — če je kaj narobe.

Starost datoteke je določena s trenutkom zadnje spremembe podatkov v njej.

REŠITVE NALOG ZA PRVO SKUPINO

R2002.1.1 Pristanišče

Naloga: str. 1

Naj bo z številka zabojnika, ki nas zanima. V vsaki plasti je $n \cdot m$ zabojnikov. Zabojniki v prvi plasti imajo torej številke od 1 do nm, tisti v drugi številke od $nm+1$ do $2nm$ in tako naprej. Da pridemo do številke plasti, si lahko pomagamo z deljenjem: če delimo $z-1$ z nm, nam celi del količnika pove številko plasti (le 1 ji moramo še prišteti, ker štejemo plasti od 1 naprej in ne od 0 naprej), ostanek (recimo mu s) pa si lahko mislimo kot številko zabojnika znotraj plasti: za tiste v prvi vrsti dobimo številke od 0 do $n-1$, za tiste v drugi vrsti številke od n do $2n-1$ in tako naprej. Do številke vrste in položaja znotraj vrste pridemo zdaj na podoben način, torej spet z deljenjem.

```
var N, M, L, Z, S: integer;
begin
  ReadLn(N, M, L, Z);
  WriteLn('Plast: ', 1 + (Z − 1) div (N * M));
  S := (Z − 1) mod (N * M);
  WriteLn('Vrsta: ', 1 + S div N);
  WriteLn('Stolpec: ', 1 + S mod N);
end.
```

R2002.1.2 Najdaljši cikel

Naloga: str. 1

Če začnemo na kateri koli postaji (najprej recimo kar na prvi) in sledimo navodilom iz tabele Postaje, bomo sčasoma obhodili ves cikel in prišli nazaj na postajo, pri kateri smo začeli. Ob tem lahko izpisujemo postaje te proge in tudi štejemo, koliko postaj smo obhodili. Nato začnemo pri kakšni postaji, ki je doslej še nismo obiskali, in na enak način poiščemo naslednjo progo. Ob vsaki progi še preverimo, če je mogoče daljša od doslej najdaljše znane. V tabeli Obiskana si zapisujemo, katere postaje smo že obiskali.

```
const N = ...;
var Postaje: array [1..N] of integer;
     i, j, Dolzina, Najdaljsa, StProg: integer;
     Obiskana: array [1..N] of boolean;
begin
   for i := 1 to N do Obiskana[i] := false;
   Najdaljsa := 0; StProg := 0;
   for i := 1 to N do if not Obiskana[i] then begin
      Dolzina := 0; j := i;
      StProg := StProg + 1; Write('Proga ', StProg, ':');
      repeat
         Dolzina := Dolzina + 1;
         Obiskana[j] := true; Write(' ', j);
         j := Postaje[j];
      until j = i;
      WriteLn(', dolžina: ', Dolzina);
      if Dolzina > Najdaljsa then Najdaljsa := Dolzina;
   end; {for, if}
   WriteLn('Dolžina najdaljše proge: ', Najdaljsa, '.');
end.
```

R2002.1.3 Razbijanje kode

Naloga: str. 2

Ker so kode omejene le na štiri znake, lahko uporabimo kar štiri gnezdene zanke. Ker je kod le $26^4 = 456974$, bi jih lahko tudi našteli kot števila in vsako pretvorili (podobno kot pri nalogi o zabojnikih) v štiri znake dolg niz, ki bi ga potem podali funkciji Test. Lahko bi uporabili tudi različico malo splošnejšega postopka, kakršnega potrebujemo pri nalogi „Razbijanje gesel“ za drugo skupino.

```
program IskanjeGesla;

   function Test(Geslo: string): boolean; external;

var A, B, C, D: char;
     S: string[4];
begin
   S := '....';
   for A := 'A' to 'Z' do begin S[1] := A;
   for B := 'A' to 'Z' do begin S[2] := B;
   for C := 'A' to 'Z' do begin S[3] := C;
   for D := 'A' to 'Z' do begin S[4] := D;
      if Test(S) then begin WriteLn(S); exit end;
   end; end; end; end;
end. {IskanjeGesla}
```

R2002.1.4 Bencin

Naloga: str. 2

Na vsaki črpalki pogledamo, katera je naslednja cenejša (in kako daleč je do nje). Če potrebujemo do nje polno posodo goriva ali manj, natočimo toliko bencina, da ga bomo imeli ravno dovolj za do tam (če imamo bencina že od prej dovolj, ga mogoče sploh ni treba dotakati), sicer pa polno. Če na poti ne bo nobene cenejše črpalke več, natočimo ravno dovolj goriva za do Bruslja (ali pa poln tank, če bi potrebovali več); to pravilo lahko gledamo kot poseben primer prejšnjega, če si mislimo, da je v Bruslju še ena črpalka, kjer dajejo bencin zastonj.

Prepričajmo se, da je ta načrt dotakanja bencina (recimo mu A) res najboljši. Recimo, da obstaja nek še boljši razpored B. Glede dotakanja na prvih nekaj črpalkah se mogoče ujema z A, prej ali slej pa mora nastopiti razlika (saj bi bila drugače razporeda čisto enaka). Recimo, da je prva razlika na i-ti črpalki; do sem torej A in B prideta z enako količino bencina v tanku in si do sem nakopljeta enake stroške, na i-ti črpalki pa dolijeta različno količino goriva.

Če napolni A tank do konca, pomeni, da B vzame tu manj bencina kot A; toda A napolni tank do konca le v primerih, ko se do naslednje cenejše črpalke (recimo ji j) ne da priti z manj bencina; recimo torej, da se za pot do j porabi poln tank in še x litrov bencina; B bo moral torej po poti doliti več kot x litrov, ker zapušča i z manj kot polnim tankom; ker pa so vse črpalke med i in j vsaj tako drage kot i (če ne še dražje), se ne bi B-jev razpored nič poslabšal, če bi na i natočil poln tank in nato pri naslednjem dotakanju malo manj.

Če pa A na črpalki i ne napolni tanka do konca, pomeni, da je natočil le toliko, kolikor potrebuje do naslednje cenejše črpalke (spet ji recimo j). Če je B natočil manj, bo moral dotakati nekje vmes (ker črpalke i ne zapušča z dovolj bencina, da bi dosegel j), tam pa bo bencin vsaj tako drag kot pri i; torej ne bi bil B nič na slabšem, če bi pri i natočil toliko kot A in nato med i in j ne bi nič dotakal (kot tudi A ne bo). Če pa je B natočil na i več kot A, ima ob odhodu z i več bencina, kot je potrebno za pot do j; torej bi lahko nekaj prihranil, če bi natočil na i malo manj in to razliko dotočil pri j, saj je tam bencin cenejši, za pot od i do j pa bi ga bilo vseeno dovolj; toda to je potemtakem sploh nemogoče, saj smo na začetku rekli, naj bo B najboljši razpored.

Tako torej vidimo: če prva razlika med najboljšim razporedom B in našim razporedom A nastopi pri i-ti črpalki, je mogoče B spremeniti v nek razpored B', ki ni nič slabši od B, se pa z A-jem ujema tudi na i-ti črpalki. Če zdaj ta korak ponavljamo, vidimo, da lahko B predelamo v A, ne da bi se kaj poslabšal, torej je tudi A najboljši razpored.

Takšen način dokazovanja, da je naš razpored najboljši (torej da predpostavimo nek boljši razpored in se potem prepričamo, da bi ga lahko postopoma spremenili v naš razpored, ne da bi se pri tem kaj poslabšal, kar torej pomeni, da ni naš razpored nič slabši od najboljšega možnega in je zato tudi sam eden od najboljših), je zelo značilen za požrešne algoritme (*greedy algorithms*), med katere spada tudi naš gornji postopek.

```
const N = ...;                          { Število črpalk. }
var   Kje, Cena: array [1..N] of real;  { Položaj črpalk in cena goriva na njih. }
      Poraba: real;                     { Poraba goriva v litrih na 100 km. }
      Posoda: real;                     { Velikost posode za gorivo v litrih. }
      Pot: real;                        { Skupna dolžina poti v kilometrih. }

var i, j: integer; Gorivo, Razd, Koliko: real;
begin
  Gorivo := 0; i := 1;
  while i <= N do begin
    j := i + 1; { j bo naslednja cenejša črpalka. }
    while j <= N do
      if Cena[j] < Cena[i] then break
      else j := j + 1;
    { Koliko goriva potrebujemo do naslednje cenejše črpalke? }
    if j > N then Razd := Pot − Kje[i] else Razd := Kje[j] − Kje[i];
    Koliko := (Razd / 100.0) * Poraba;
```

```
    if Gorivo < Koliko then begin { Treba bo dotočiti. }
      if Koliko <= Posoda then begin { Natočimo dovolj za pot do j. }
        WriteLn('Na črpalki ', i, ' dotočimo ', (Koliko − Gorivo):0:2, ' l.');
        Gorivo := 0; i := j;
      end else begin { Natočimo poln tank, a ne bo dovolj do j. }
        WriteLn('Na črpalki ', i, ' dotočimo ', (Posoda − Gorivo):0:2, ' l.');
        Gorivo := Posoda − ((Kje[i + 1] − Kje[i]) / 100.0) * Poraba; i := i + 1;
      end; {if}
    end else begin { Imamo dovolj do j. }
      Gorivo := Gorivo − Koliko; i := j;
    end; {if}
  end; {while}
end.
```

Gornji program pa ima še eno slabost: vgnezdeno zanko za iskanje naslednje cenejše črpalke. Če je vsaka črpalka dražja od prejšnje, bo šel j vsakič od i + 1 vse do N + 1; in če so črpalke obenem dovolj daleč narazen, da jih ne bomo mogli preskakovati, ampak bomo na vsaki dotakali, se bo zunanja zanka res izvedla N-krat. To dvoje skupaj pomeni, da ima naš postopek v najslabšem primeru časovno zahtevnost $O(N^2)$. V praksi do tega verjetno ne bi prišlo, pa tudi če bi, bi lahko na današnjih računalnikih vseeno dovolj hitro obdelali tudi poti z več tisoč črpalkami; kljub temu pa poskusimo razmisliti še o učinkovitejši rešitvi.

Recimo, da bi si hoteli na začetku v tabeli Nasl pripraviti za vsako črpalko podatek o naslednji najcenejši črpalki. Postopek bi bil lahko tak:

```
Iščemo := {};
for i := 1 to N do
  for j ∈ Iščemo do
    if Cena[i] < Cena[j] then
      Nasl[j] := i; Iščemo := Iščemo − {j};
  Iščemo := Iščemo ∪ {i};
```

Vzdržujemo torej množico črpalk, za katere še nismo našli naslednje cenejše; pri vsaki črpalki to množico pregledamo in zbrišemo iz nje tiste črpalke, od katerih je trenutna (i-ta) črpalka cenejša. Nato še i dodamo v množico, da bomo v nadaljevanju poiskali tudi naslednjo cenejšo za njo. Na koncu tega postopka ostanejo v množici *Iščemo* še črpalke, za katerimi ni nobene cenejše.

Kako naj v praksi predstavimo množico *Iščemo*? Lahko bi uporabili seznam črpalk, urejen po padajočih cenah. Če je namreč i sploh cenejša od katere izmed črpalk v *Iščemo*, bo gotovo cenejša od najdražjih nekaj. Pri vsakem i bi pregledovali ta seznam od najdražjih k cenejšim; dokler opažamo take, ki so dražje od i, jih brišemo iz seznama, čim pa naletimo na cenejšo od i, se ustavimo, saj takrat vemo, da ni v seznamu nobene več dražje od i. To pa tudi pomeni, da je i vsaj tako draga kot katera koli druga v seznamu in jo lahko dodamo kar na začetek seznama. Zdaj pa, ko smo ugotovili, da bomo vedno dodajali in brisali le pri začetku seznama, vidimo, da pravzaprav ni treba pisati res splošnega seznama; dovolj bo že čisto navaden sklad. Začetek gornjega programa bi torej lahko spremenili takole:

```
var i, j: integer; Gorivo, Razd, Koliko: real;
  Sklad, Nasl: array [1..N] of integer;
begin
```

```
j := 0;
for i := 1 to N do begin
  while j > 0 do { Od katerih na skladu je i cenejša? }
    if Cena[Sklad[j]] <= Cena[i] then break
    else begin Nasl[Sklad[j]] := i; j := j − 1 end;
  j := j + 1; Sklad[j] := i; { Dodajmo še i na sklad. }
end; {for}
while j > 0 do { Ostale so črpalke brez naslednje cenejše. }
  begin Nasl[Sklad[j]] := N + 1; j := j − 1 end;
Gorivo := 0; i := 1;
while i <= N do begin
  j := Nasl[i];
  { Koliko goriva potrebujemo do naslednje cenejše črpalke? }
  ...
```

Zdaj vsaka iteracija zanke **while** j bodisi vzame neko črpalko s sklada ali pa se ustavi (kliče **break**). Poleg tega dodamo vsako črpalko na sklad le enkrat. Skupaj imamo torej največ N dodajanj in N brisanj, pa še največ N takih iteracij zanke **while** j, ki kličejo **break**. Tako predelan postopek ima časovno zahtevnost $O(N)$.

REŠITVE NALOG ZA DRUGO SKUPINO

R2002.2.1 Kodiranje

Naloga: str. 3

Da bo povprečno besedilo predstavljeno s čim krajšim zaporedjem bitov, je koristno, če dobijo pogostejše črke krajše kode kot redkejše črke. Če bi imeli štiri črke in bi se vsaka pojavljala v 25 % primerov, bi bilo gotovo najbolj smiselno dati vsaki po eno dvobitno kodo; če bi imeli osem črk s pogostostjo pojavitev 12,5 %, bi dali vsaki eno trobitno kodo in tako naprej. Mi imamo dve črki s pogostostjo 25 %, namreč c in e, zato jima dodelimo dvobitni kodi; c-ju na primer 00, e-ju pa 01. Črki a in g, ki se pojavita vsaka v 12,5 % primerov, pokrijeta torej skupaj 25 % primerov in je zato dobro, da obe skupaj dobita tretjo dvobitno kodo, npr. 10; da ju bomo lahko ločili med sabo, pa dodajmo temu še tretji bit in torej a kodirajmo s 100, g pa s 101. Preostale štiri črke s pogostostjo 6,25 % vse skupaj tudi pokrijejo 25 % primerov in jim zato dodelimo še četrto dvobitno kodo, 11, ter vsaki še po dva bita, da bodo kode enolične. Dobimo takšen kod:

Črka	a	b	c	d	e	f	g	h
Koda	100	1100	00	1101	01	1110	101	1111

Opazimo lahko pomembno lastnost, da nobena od teh osmih kod ni podaljšek kakšne druge; drugače bi bilo mogoče nekatera zaporedja bitov težko dekodirati v prvotna besedila ali pa bi se lahko celo zgodilo, da bi se več besedil zakodiralo v isto zaporedje bitov.

Splošnejša oblika opisanega razmisleka se imenuje *Huffmanovo kodiranje*.

R2002.2.2 Prijatelji in sovražniki

Naloga: str. 3

Za vsako osebo vzdržujmo podatek o tem, ali je Jurijev prijatelj in ali je njegov sovražnik. Ti podatki imajo lahko dve stanji: ali smo že ugotovili, da je ta oseba prijatelj (oz. sovražnik) ali pa še nismo (slednje lahko pomeni, da ta človek res ni naš prijatelj/sovražnik ali pa da sicer je, vendar tega še nismo odkrili). Ko o neki osebi na novo izvemo, da je prijatelj ali sovražnik, je treba preiskati še njene neposredne prijatelje in neposredne sovražnike, ker lahko zdaj tudi o teh mogoče izvemo, v kakšni zvezi so z Jurijem. Zato imejmo še seznam (pravzaprav sklad) oseb, ki si jih bo treba še pobliže ogledati. Vsaka oseba lahko vstopi na ta seznam največ dvakrat: ko izvemo, da je Jurijev prijatelj, in ko izvemo, da je njegov sovražnik. Celoten postopek lahko poženemo tako, da dodamo Jurija na seznam in označimo, da je prijatelj samemu sebi.

```
program ZunanjaPolitika;

  const N = ...;
  type KdoJeKdo = (Neznanec, Prijatelj, Sovrag);
  var Odnos: array [0..N − 1, 0..N − 1] of KdoJeKdo;

var Prij, Sovr: array [0..N − 1] of boolean; Dodaj: boolean;
    Seznam: array [0..2 * N − 1] of integer; i, j, Dolzina: integer;
begin
  for i := 0 to N − 1 do begin Prij[i] := false; Sovr[i] := false end;
  Seznam[0] := 0; Dolzina := 1; Prij[0] := true;
  while Dolzina > 0 do begin
    Dolzina := Dolzina − 1; i := Seznam[Dolzina];
    for j := 0 to N − 1 do begin
      Dodaj := false;
      if (   (Prij[i] and (Odnos[i, j] = Prijatelj))
          or (Sovr[i] and (Odnos[i, j] = Sovrag)) ) and not Prij[j] then
          begin Prij[j] := true; Dodaj := true end;
      if (   (Prij[i] and (Odnos[i, j] = Sovrag))
          or (Sovr[i] and (Odnos[i, j] = Prijatelj)) ) and not Sovr[j] then
          begin Sovr[j] := true; Dodaj := true end;
      if Dodaj then begin Seznam[Dolzina] := j; Dolzina := Dolzina + 1 end;
    end; {for}
  end; {while}
  WriteLn('Pravi prijatelji:');
  for i := 1 to N − 1 do if Prij[i] and not Sovr[i] then WriteLn(i);
  WriteLn('Pravi sovražniki:');
  for i := 1 to N − 1 do if Sovr[i] and not Prij[i] then WriteLn(i);
end. {ZunanjaPolitika}
```

R2002.2.3 Razbijanje gesel

Naloga: str. 4

Gesla lahko generiramo s podprogramom, ki rekurzivno kliče samega sebe, da doda geslu še naslednjo črko, če pa je geslo že dovolj dolgo, lahko kliče tudi funkcijo Test. Pazimo še na to, da mora imeti geslo vsaj eno števko in vsaj eno črko, zato lahko, ko dodajamo zadnji znak, nekaj možnosti preskočimo, da ne bi funkcije Test po nepotrebnem klicali prevečkrat.

```
program IskanjeGesla;
```

```
  function Test(S: string): boolean; external;

const Najkrajse = 3; Najdaljse = 8;
type KajIma = set of (Crko, Stevko);
var Geslo: string; Nasel: boolean;

procedure Test2(Dolzina: integer);
begin
  Geslo[0] := Chr(Dolzina);
  if Test(Geslo) then begin WriteLn(Geslo); Nasel := true end;
end; {Test2}

procedure Generiraj(Mesto: integer; Ima: KajIma);
var i, Mala: integer;
begin
  { Geslo ima zdajle Mesto − 1 znakov. Kot naslednji znak mu lahko
    dodamo črko, razen če je to zadnji znak in ima geslo doslej same
    črke; če to ni zadnji znak in ima same črke, bomo lahko kasneje
    dodali še kakšno števko, tako da ni narobe, če zdaj dodamo črko. }
  if (Mesto < Najdaljse) or (Stevko in Ima) then
    for Mala := 0 to 1 do for i := 0 to 25 do begin
      Geslo[Mesto] := Chr(Ord('A') + Mala * (Ord('a') − Ord('A')) + i);
      { Geslo je zdaj dolgo Mesto znakov. Če ga lahko še podaljšujemo,
        rekurzivno kličimo podprogram Generiraj, sicer pa geslo le
        preizkusimo (s podprogramom Test2). }
      if Mesto = Najdaljse then Test2(Mesto)
                           else  Generiraj(Mesto + 1, Ima + [Crko]);
      if Nasel then exit;
    end; {for}
  { Na enak način kot črko lahko lahko poskusimo dodati tudi števko. }
  if (Mesto < Najdaljse) or (Crko in Ima) then
    for i := 0 to 9 do begin
      Geslo[Mesto] := Chr(Ord('0') + i);
      if Mesto = Najdaljse then Test2(Mesto)
                           else  Generiraj(Mesto + 1, Ima + [Stevko]);
      if Nasel then exit;
    end; {for}
  { Ostane še možnost, da gesla ne podaljšujemo in ostanemo le pri
    dosedanjih Mesto − 1 znakih. Seveda moramo še vseeno preveriti,
    da geslo ni prekratko in da vsebuje tako črke kot števke. }
  if (Mesto > Najkrajse) and (Ima = [Crko, Stevko]) then Test2(Mesto − 1);
end; {Generiraj}

begin
  Generiraj(1, []);
end. {IskanjeGesla}
```

Na voljo nam je 52 različnih črk in 10 različnih števk; torej obstaja 62^n gesel dolžine n, vendar pa jih je med temi 10^n takih, ki so iz samih števk, in 52^n takih, ki so iz samih črk. Upoštevajmo še, da je $\sum_{n=0}^{N-1} a^n = (a^N - 1)/(a-1)$ in zato $\sum_{n=b}^{c} a^n = (a^{c+1} - a^b)/(a-1)$; vseh sprejemljivih gesel dolžine od 3 do 8 je torej

$$\frac{62^9 - 62^3}{62 - 1} - \frac{52^9 - 52^3}{52 - 1} - \frac{10^9 - 10^3}{10 - 1} = 167\,411\,381\,963\,280.$$

Če imamo srečo, bomo sicer že ob prvem poskusu zadeli pravo geslo, v najslabšem primeru pa bo pravo šele zadnje in v povprečju lahko pričakujemo (če so res vsa enako verjetna), da jih bomo morali preizkusiti približno polovico. Četudi bi jih lahko v sekundi preizkusili milijardo, bi to trajalo skoraj ves dan. Zato ta pristop k odkrivanju gesel verjetno ni sprejemljiv.

R2002.2.4 TiVo

Naloga: str. 5

Podatkovna struktura je krožni izravnalni pomnilnik, za katerega potrebujemo le kazalce na njegov začetek, na konec in na trenutno mesto, s katerega predvajamo posneto sliko. Da lažje primerjamo kazalce med seboj, vedno ohranjamo urejenost: Najstarejsi $\leq$ Prikazani $\leq$ Najnovejsi, ko pa moramo sporočiti številko bloka, jo preračunamo po modulu velikosti diska. Da nam vrednosti ne pobegnejo v neskončnost, jih brzdamo, vendar vse tri kazalce hkrati glede na najmanjšega.

Če pri predvajanju pridemo do najnovejše ali najstarejše slike, obtičimo pri njej. Če smo pri najstarejši, nam najstarejše kar naprej bežijo in je najbolje, da kar preklopimo na normalno hitrost predvajanja v izogib neenakomernim skokom zaradi morebitne časovne neusklajenosti snemanja in predvajanja. Podobno naredimo, če se s povečano hitrostjo zaletimo ob najnovejši rob.

```
const StBlokov = 300001;

{ Velja urejenost: Najstarejsi ≤ Prikazani ≤ Najnovejsi,
  dejanska številka bloka pa je po modulu StBlokov. }
var Najstarejsi: integer value 0;
    Najnovejsi: integer value 0;
    Prikazani: integer value 0;
    Shranjenih: integer value 0;    { Število shranjenih sličic. }
    Hitrost: integer value 1;       { Hitrost predvajanja (1 = normalno). }

function KamShranitiNovo: integer;
begin
  Najnovejsi := Najnovejsi + 1; { Glava se premakne. }
  if Shranjenih < StBlokov then Shranjenih := Shranjenih + 1
  else begin { Rep prirežemo. }
    Najstarejsi := Najstarejsi + 1;
    if Prikazani <= Najstarejsi then { Raje enega več, kot bi bilo nujno. }
      begin Prikazani := Najstarejsi + 1; Hitrost := 1 end;
    if Najstarejsi >= StBlokov then begin { Naokrog, da ne gre v neskončnost. }
      Prikazani := Prikazani − StBlokov;
      Najnovejsi := Najnovejsi − StBlokov;
      Najstarejsi := Najstarejsi − StBlokov;
    end; {if}
  end; {if}
  KamShranitiNovo := Najnovejsi mod StBlokov;
end; {KamShranitiNovo}

procedure IzberiNacinPredvajanja(NovaHitrost: integer);
  begin Hitrost := NovaHitrost end;

function KateroSlikoPrikazati: integer;
begin
  Prikazani := Prikazani + Hitrost;
```

```
  if Hitrost > 0 then begin
    { Rep nas ne more ujeti, lahko pa se zaletimo v najnovejšega. }
    if Prikazani >= Najnovejsi then { Bolje se je ustaviti tik pred njim. }
      begin Prikazani := Najnovejsi - 1; Hitrost := 1 end;
  end else begin
    { Stojimo ali gremo nazaj, a pred najstarejšim moramo bežati. }
    if Prikazani <= Najstarejsi then
      begin Prikazani := Najstarejsi + 1; Hitrost := 1 end;
  end; {if}
  KateroSlikoPrikazati := Prikazani mod StBlokov;
end; {KateroSlikoPrikazati}
```

REŠITVE NALOG ZA TRETJO SKUPINO

R2002.3.1 Limuzine

Naloga: str. 8

Recimo, da traja sestanek t minut, ura vožnje nas stane v tolarjev, do parkirišča i je t_i minut vožnje in cena ure parkiranja na njem je c_i tolarjev. Parkiranje na parkirišču i je potem možno le, če je $2t_i < t$, stane pa nas $2t_i \cdot v + (t - 2t_i) \cdot c_i$ šestdesetink tolarja. Možnost, da sploh ne parkiramo, ampak se le vozimo naokoli, pa nas stane $t \cdot v$ šestdesetink tolarja. Zdaj ni treba drugega, kot da ugotovimo, kaj od tega je najceneje. Spodnji program si v Najceneje zapisuje stroške za doslej najcenejšo najdeno različico, v NajKje oz. NajCas pa sta številka parkirišča oz. čas parkiranja pri tej različici.

```
program Limuzine;
var T: text; CenaVoz, DolzSest, CenaPark, CasVoz, CasPark: longint;
    Cena, Najceneje, NajKje, NajCas, StPark: longint;
begin
  Assign(T, 'limo.in');
  Reset(T); ReadLn(T, CenaVoz, DolzSest);

  Najceneje := CenaVoz * DolzSest; NajKje := 0; NajCas := 0; StPark := 0;
  while true do begin { berimo parkirišča }
    ReadLn(T, CenaPark, CasVoz); StPark := StPark + 1;
    if CenaPark <= 0 then break; { konec vhodne datoteke }
    CasPark := DolzSest - 2 * CasVoz;
    if CasPark > 0 then begin { poskusimo parkirati tukaj }

      Cena := CenaPark * CasPark + CenaVoz * 2 * CasVoz;
      if Cena < Najceneje then begin { novo najcenejše parkirišče }
        Najceneje := Cena; NajKje := StPark;
        NajCas := DolzSest - 2 * CasVoz;
      end; {if}

    end; {if}
  end; {while}
  Close(T);

  Assign(T, 'limo.out');
  Rewrite(T); WriteLn(T, NajKje, ' ', NajCas); Close(T);
end. {Limuzine}
```

R2002.3.2 Uvrstitve tekmovalcev

Naloga: str. 9

Sprehodimo se skozi zaporedje in si za vsakega tekmovalca zapomnimo, koliko krogov je naredil in kdaj je naredil zadnjega. Nato jih uredimo padajoče po številu krogov, znotraj istega števila krogov pa po naraščajočih časih. V tem vrstnem redu jih izpišemo. Spodnji program si pomaga s strukturo KolesarT, ki hrani za posameznega tekmovalca poleg štartne številke (St) še število prevoženih krogov (StKrogov) in čas, ko je prevozil zadnjega od njih (Cas; to v resnici ni čisto pravi čas, pač pa indeks znotraj zaporedja prehodov skozi ciljno črto, kar pa nam že zadostuje, da lahko ugotovimo, kdo je prej prevozil svoj zadnji krog).

```
program Tekma;
const MaxN = 2000;
type KolesarT = record St, StKrogov: integer; Cas: longint end;
var T: text; j, k, N: integer; Zdaj: longint;
    Kolesarji: array [1..MaxN] of KolesarT; Kol: KolesarT;
begin
  { Preberimo podatke o tekmi. }
  Assign(T, 'tekma.in');
  Reset(T); ReadLn(T, N); Zdaj := 0;
  for k := 1 to N do with Kolesarji[k] do
    begin St := k; StKrogov := 0; Cas := Zdaj end;
  while true do begin
    ReadLn(T, k); Zdaj := Zdaj + 1; if k <= 0 then break;
    with Kolesarji[k] do begin StKrogov := StKrogov + 1; Cas := Zdaj end;
  end; {while}
  Close(T);

  { Uredimo kolesarje po rezultatih. }
  for k := 2 to N do begin
    Kol := Kolesarji[k]; j := k - 1;
    { Prvih k - 1 celic tabele Kolesarji je že urejenih. Vrinimo kolesarja k
      na pravo mesto v ta del tabele (torej pred vse take, ki so se na tekmi odrezali }
    while j > 0 do with Kolesarji[j] do                      { slabše od njega). }
      { Je bil kolesar j boljši od trenutnega? Če da, moramo nehati in
        vpisati trenutnega na (j + 1)-vo mesto; če pa ne, se bo trenutni
        uvrstil pred j in moramo kolesarja j premakniti za eno mesto naprej. }
      if StKrogov > Kol.StKrogov then break
      else if (StKrogov = Kol.StKrogov) and (Cas < Kol.Cas) then break
      else begin Kolesarji[j + 1] := Kolesarji[j]; j := j - 1 end;
    Kolesarji[j + 1] := Kol;
  end; {for k}

  { Izpišimo rezultate. }
  Assign(T, 'tekma.out'); Rewrite(T);
  for k := 1 to N do if Kolesarji[k].StKrogov > 0 then
    WriteLn(T, Kolesarji[k].St);
  Close(T);
end. {Tekma}
```

Postopek, ki smo ga uporabili za urejanje kolesarjev, se imenuje urejanje z vstavljanjem (*insertion sort*). Ko se začnemo ukvarjati s kolesarjem k, imamo kolesarje $1, \ldots, k-1$ že urejene v pravem vrstnem redu. Potem gremo od konca tega zaporedja proti začetku in dokler srečujemo kolesarje, slabše od k, jih premikamo za eno mesto naprej; takoj ko naletimo na takega, ki je boljši od k, pa vstavimo kolesarja k v izpraznjeno celico tik za njim. Tako imamo zdaj urejeno zaporedje kolesarjev $1, \ldots, k$ in se lahko lotimo kolesarja $k+1$.

R2002.3.3 Število vsot

Naloga: str. 10

Naj bo $a(n,k)$ število načinov, na katere lahko n zapišemo kot vsoto k števil. Glavni razmislek, ki ga moramo opraviti, da pridemo do rešitve, je naslednji: vsak razcep n-ja na vsoto k števil ima ali vsaj en seštevanec enak 1 ali pa vse seštevance večje od 1. V prvem primeru predstavljajo ostali seštevanci razcep števila $n-1$ na vsoto $k-1$ števil, teh razcepov pa je $a(n-1,k-1)$; v drugem primeru pa bi, če bi vsakega od seštevancev zmanjšali za 1, dobili nek razcep števila $n-k$ na vsoto k števil, takih razcepov pa je $a(n-k,k)$. Torej je $a(n,k) = a(n-1,k-1) + a(n-k,k)$. Posebni (robni) primeri so: $a(n,k) = 0$ za $n < k$; $a(0,0) = 1$; in $a(n,0) = 0$ za $n > 0$.

Da se dokopljemo do vrednosti $a(N,K)$, ki nas zanima, moramo prej izračunati vrednosti $a(n,k)$ za n od 1 do N in za k od 1 do K. To lahko naredimo z dvema gnezdenima zankama, ki gresta po n od 1 do N in pri vsakem n-ju še po k od 1 do $\min\{n,K\}$ (ker števila n ne moremo zapisati kot vsoto več kot n števil, vsote več kot K števil pa nas ne zanimajo).

```
program SteviloVsot;
const MaxN = 100;
var T: text; ni, ki, K, N: integer; a: array [0..MaxN, 0..MaxN] of longint;
begin
  Assign(T, 'vsote.in'); Reset(T); ReadLn(T, N, K); Close(T);

  { Izračunajmo. }
  a[0, 0] := 1;
  for ni := 1 to N do begin
    a[ni, 0] := 0; ki := 1;
    while (ki <= ni) and (ki <= K) do begin
      a[ni, ki] := a[ni − 1, ki − 1] + a[ni − ki, ki];
      ki := ki + 1;
    end; {while ki}
  end; {for ni}

  { Izpišimo rezultat. }
  Assign(T, 'vsote.out'); Rewrite(T); WriteLn(T, a[N, K]); Close(T);
end. {SteviloVsot}
```

Gornji program mora hraniti v tabeli `a` vrednosti $a(n,k)$ za vse doslej izračunane pare (n,k), zato je njegova prostorska (pomnilniška) zahtevnost reda $O(NK)$. To lahko zmanjšamo na samo $O(N)$, če računamo vrednosti $a(n,k)$ po naraščajočih k in pri vsakem k še po vseh potrebnih n (od k do N). Pri tem vrstnem redu bomo, ko enkrat dosežemo določeno vrednost k, potrebovali od starih rezultatov le še tiste za $k-1$, starejših (za $k-2$, $k-3$ itd.) pa ne. Spodnji program ima v tabeli `a` prostora le za vrednosti $a(n,k)$ (v celici `a[j, n]`) in $a(n,k-1)$ (v celici `a[1 − j, n]`). Ko pridemo pri trenutnem k do konca, lahko s prireditvijo `j := 1 − j` dosežemo tak učinek, kot da bi se vrstici tabele `a` zamenjali; rezultati, ki jih bomo računali pri $k+1$, se bodo vpisovali čez stare rezultate za $k-1$, ki jih ne bomo več potrebovali.

```
program SteviloVsot2;
const MaxN = 100;
var T: text; ni, ki, K, N, j: integer; a: array [0..1, 0..MaxN] of longint;
begin
  Assign(T, 'vsote.in'); Reset(T); ReadLn(T, N, K); Close(T);

  { Izračunajmo. }
  j := 0; a[j, 0] := 1;
  for ki := 1 to K do begin
    j := 1 − j;
    for ni := ki to N do begin
      a[j, ni] := a[1 − j, ni − 1];
      if ni − ki >= ki then a[j, ni] := a[j, ni] + a[j, ni − ki];
    end; {for ni}
  end; {for ki}

  { Izpišimo rezultat. }
  Assign(T, 'vsote.out'); Rewrite(T); WriteLn(T, a[j, N]); Close(T);
end. {SteviloVsot2}
```

Kot zanimivost lahko omenimo, da je največje število, s katerim imamo opravka pri tej nalogi, $a(100, 18) = 11\,087\,828$. Vsota $A(n) := \sum_{k=1}^{n} a(n, k)$ pa se pri velikih n obnaša približno tako kot[4] $\frac{1}{4n\sqrt{3}} \cdot e^{\pi\sqrt{2n/3}}$; $A(100) = 190\,569\,292$.

R2002.3.4 Sestanki

Naloga: str. 10

Začetne in končne čase vseh sestankov si uredimo v naraščajočem vrstnem redu, pri vsakem pa si še zapomnimo, ali gre za začetni ali za končni čas. Če se nato sprehajamo po tem vrstnem redu, bomo za vsako obdobje med dvema takima časoma vedeli, koliko sestankov je takrat v teku (števec sestankov povečamo vsakič, ko naletimo na začetni čas nekega sestanka, in ga zmanjšamo ob vsakem končnem času). Čim najdemo dovolj dolg interval, ko ni v teku nobenih drugih sestankov, lahko tja postavimo naš novi sestanek in nehamo. Da upoštevamo še pogoja f in t, lahko dodamo še 0 kot začetni in f kot končni čas. Če pridemo do konca zaporedja, ne da bi našli dovolj dolg prost interval s koncem pred časom t, lahko zaključimo, da sestanek ni mogoč.

```
program Sestanki1;
const MaxSest = 10000;
var T: text; Casi: array [1..2 * MaxSest + 2] of longint; i, StCasov, StSest: integer;

  procedure Dodaj(Cas: longint);
  var i: integer;
  begin
    i := StCasov; while i >= 1 do
      if Abs(Casi[i]) < Abs(Cas) then break
      else begin Casi[i + 1] := Casi[i]; i := i − 1 end;
    Casi[i + 1] := Cas; StCasov := StCasov + 1;
  end; {Dodaj}

var MinCas, MaxCas, Dolz, Zac, Trajanje: longint;
begin
  { Preberimo vhodne podatke. }
```

[4] Glej *The On-Line Encyclopedia of Integer Sequences*, A000041.

```
  Assign(T, 'sestanki.in');
  Reset(T); ReadLn(T, MinCas, MaxCas, Dolz); StSest := 0;
  while true do begin
    ReadLn(T, Zac, Trajanje);
    if Zac + Trajanje = 0 then { 0 0 preskočimo; če sta dva zaporedna, končamo }
      begin ReadLn(T, Zac, Trajanje); if Zac + Trajanje = 0 then break end;
    { Časi koncev bodo dobili negativni predznak, da jih bomo lahko
      ločili od časov začetkov. Predpostavili bomo, da se noben
      sestanek ne začne ob negativnem času ali pa konča ob času 0. }
    { Vstavimo čas začetka in konca v urejeno zaporedje. }
    Dodaj(Zac); Dodaj(−(Zac + Trajanje));
  end; {while}
  Close(T);

  { V zaporedje dodajmo še „sestanek“ 0..MinCas. To nas bo prisililo,
    da sestanka ne bomo začeli prezgodaj. }
  if MinCas > 0 then begin Dodaj(0); Dodaj(−MinCas); Zac := Casi[1] end
  else Zac := MinCas;

  { Poiščimo primeren čas začetka sestanka. }
  i := 1; StSest := 0;
  while (i <= StCasov) and (Zac + Dolz <= MaxCas) do begin
    if (StSest <= 0) and (Zac + Dolz <= Abs(Casi[i])) then break;
    if Casi[i] < 0 then StSest := StSest − 1 else StSest := StSest + 1;
    Zac := Abs(Casi[i]); i := i + 1;
  end; {while}

  { Izpišimo rezultat. }
  Assign(T, 'sestanki.out'); Rewrite(T);
  if Zac + Dolz <= MaxCas then WriteLn(T, Zac)
  else WriteLn(T, 'SESTANEK NI MOZEN');
  Close(T);
end. {Sestanki1}
```

Še ena različica iste rešitve je, da uredimo sestanke po začetnih časih, nato pa se sprehajamo po zaporedju in hranimo v neki spremenljivki najzgodnejši naslednji čas (recimo S), ko ni nobenega sestanka (na začetku postavimo S na f). Ob sestanku (s_{ij}, d_{ij}) je mogoče, da je $S + d \leq s_{ij}$ in v tem primeru lahko novi sestanek začnemo ob času S; sicer pa bo treba novi sestanek začeti po času $s_{ij} + d_{ij}$, zato S povečamo do toliko, če je trenutno manjši. Če pride S čez vrednost $t - d$, pa vemo, da sestanek ni mogoč. Lepo pri tej drugi rešitvi je, da moramo pri k sestankih urejati le k elementov, ne pa $2k$ (oz. $2k + 2$) kot zgoraj. Postopek urejanja z vstavljanjem, ki ga uporabljamo tu za urejanje časov, je preprost, vendar precej neučinkovit, saj za urejanje n elementov porabi $O(n^2)$ časa. To pomeni, da porabi zgornji program, ki mora urejati dvakrat več elementov kot spodnji, za to približno štirikrat več časa, ta razlika pa se pošteno pozna tudi pri celotnem času izvajanja, saj porabita oba programa večino svojega časa ravno za urejanje (drugi deli postopka imajo linearno časovno zahtevnost). Pri kakšnem učinkovitejšem postopku urejanja bi bila ta razlika najbrž manjša.

```
program Sestanki2;
const MaxSest = 10000;
var T: text; Zacetki, Konci: array [1..MaxSest] of longint; i, StSest: integer;
   MinCas, MaxCas, Dolz, Zac, Trajanje: longint;
begin
```

```
  { Preberimo vhodne podatke. }
  Assign(T, 'sestanki.in');
  Reset(T); ReadLn(T, MinCas, MaxCas, Dolz); StSest := 0;
  while true do begin
    ReadLn(T, Zac, Trajanje);
    if Zac + Trajanje = 0 then { 0 0 preskočimo; če sta dva zaporedna, končamo }
      begin ReadLn(T, Zac, Trajanje); if Zac + Trajanje = 0 then break end;
    { Zaporedje sestankov naj bo urejeno naraščajoče po začetnem času. }
    i := StSest; while i >= 1 do
      if Zacetki[i] <= Zac then break
      else begin Zacetki[i + 1] := Zacetki[i]; Konci[i + 1] := Konci[i]; i := i − 1 end;
    Zacetki[i + 1] := Zac; Konci[i + 1] := Zac + Trajanje;
    StSest := StSest + 1;
  end; {while}
  Close(T);
  { Poiščimo primeren čas začetka sestanka. }
  Zac := MinCas; i := 1;
  while (i <= StSest) and (Zac + Dolz <= MaxCas) do begin
    if Zac + Dolz <= Zacetki[i] then break;
    if Konci[i] > Zac then Zac := Konci[i];
    i := i + 1;
  end; {while}
  { Izpišimo rezultat. }
  Assign(T, 'sestanki.out'); Rewrite(T);
  if Zac + Dolz <= MaxCas then WriteLn(T, Zac)
  else WriteLn(T, 'SESTANEK NI MOZEN');
  Close(T);
end. {Sestanki2}
```

R2002.3.5 Produkt števil

Naloga: str. 11

Ničle se v podzaporedje gotovo ne splača vzeti, če hočemo imeti velik produkt (razen če so vsi drugi produkti, ki jih lahko dobimo, negativni). Torej v mislih razrežimo zaporedje pri ničlah in se posebej posvetimo vsakemu od nastalih kosov (zanje torej predpostavimo, da ne vsebujejo ničel); za vsak kos poiščemo njegovo najboljše podzaporedje in na koncu izpišemo tisto, ki nam da največji produkt od vseh.

Ker imamo opravka s celimi števili, so po absolutni vrednosti vsa večja ali enaka 1, torej absolutni vrednosti našega produkta ne morejo škodovati — če jih vzamemo več, se bo absolutna vrednost lahko le povečala ali ostala enaka. Paziti moramo le še na predznak. Če je v opazovanem kosu zaporedja sodo mnogo negativnih števil, lahko vzamemo kar celoten kos, saj bo dal pozitiven produkt z največjo možno absolutno vrednostjo. Drugače pa se moramo odpovedati enemu od negativnih števil, torej vzeti ali čim več pred takim številom ali pa čim več za njim. Očitno bomo imeli največje možnosti za velik produkt, če bomo vzeli vse pred zadnjim negativnim številom ali pa vse za prvim. Če je kos sestavljen iz enega samega števila in je le-to negativno, nam ne ostane drugega, kot da vzamemo to in upamo, da bo v kakšnem drugem kosu produkt boljši (pravzaprav bo boljša že katera od ničel med kosi, če imamo več kosov).

Ker lahko postanejo zmnožki, s katerimi moramo tu delati, zelo veliki, si je koristno pomagati z naslednjo predstavitvijo števil: število $\pm 2^k$ predstavimo s parom $(\pm 1, k)$. Produkt dveh takih števil, recimo (s_1, k_1) in (s_2, k_2), lahko potem predstavimo s parom $(s_1 s_2, k_1 + k_2)$. Ničlo lahko predstavimo s parom $(0, 0)$. Števili (s_1, k_1) in (s_2, k_2) lahko tudi preprosto primerjamo — najprej po prvi komponenti (ker je vsako pozitivno število večje od vsakega negativnega), če pa sta po tej enaki, primerjamo še k_1 in k_2 (če sta pozitivni, je večje tisto z večjim eksponentom, sicer pa tisto z manjšim).

Spodnji program računa v (s1, k1) produkt vsega, kar se v trenutnem kosu pojavlja pred zadnjim doslej najdenim negativnim številom; v (s2, k2) produkt vsega za prvim negativnim številom v kosu; in v (s3, k3) vse od vključno zadnjega negativnega števila naprej (če v trenutnem kosu še ni negativnih števil, je to produkt celega kosa). Na koncu zaporedja (pri i = N + 1) si mislimo še eno ničlo, da nam zaključi trenutni kos.

```
program ProduktStevil;
var sb, kb, bOd, bDo: longint; { najboljši doslej znani zmnožek }

  procedure Kandidat(s, k, iOd, iDo: longint);
  begin
    if (iDo >= iOd) and ((s > sb) or ((s = sb) and (s * k > sb * kb))) then
      begin sb := s; kb := k; bOd := iOd; bDo := iDo end;
  end; {Kandidat}

var T: text; i, N, a, KosOd, PrvaNeg, ZadnjaNeg, s, k, s1, k1, s2, k2, s3, k3: longint;
begin
  { Preberimo vhodne podatke. }
  Assign(T, 'produkt.in');
  Reset(T); ReadLn(T, N); KosOd := 0;
  for i := 1 to N + 1 do begin

    { Preberimo naslednje število, a. Na koncu zaporedja si
      mislimo še eno ničlo, da nam konča prejšnjo skupino števil. }
    if i <= N then ReadLn(T, a) else a := 0;
    { Izrazimo a v obliki s · 2^k.
      k je torej pravzaprav dvojiški logaritem vrednosti |a| (če a ni 0).
      Pomagali si bomo z operatorjem shl: „x shl y“ je vrednost x,
      zamaknjena za y bitov v levo. }
    if a < 0 then begin s := −1; a := −a end
    else if a > 0 then s := 1 else s := 0;
    k := 0; while a > longint(1) shl k do k := k + 1;
    { Začetni kandidat za najboljši zmnožek naj bo kar prvo število zaporedja. }
    if i = 1 then begin sb := s; kb := k; bOd := i; bDo := i end;

    { s1 · 2^k1 = produkt v trenutnem kosu pred zadnjim negativnim številom.
      s2 · 2^k2 = produkt v trenutnem kosu po prvem negativnem številu.
      s3 · 2^k3 = produkt v trenutnem kosu od vklj. zadnjega negativnega števila. }

    if KosOd = 0 then begin { inicializacija na začetku kosa }
      s1 := 1; k1 := 0; s2 := 1; k2 := 0; s3 := 1; k3 := 0;
      PrvaNeg := 0; ZadnjaNeg := 0; KosOd := i;
    end; {if}

    if s < 0 then begin { negativno število }
      if PrvaNeg = 0 then PrvaNeg := i
      else begin s2 := s2 * s; k2 := k2 + k end;
      s1 := s1 * s3; k1 := k1 + k3; s3 := s; k3 := k; ZadnjaNeg := i;
    end else if s > 0 then begin { pozitivno število }
```

```
      if PrvaNeg > 0 then begin s2 := s2 * s; k2 := k2 + k end;
      s3 := s3 * s; k3 := k3 + k;
    end else begin { konec kosa }
      Kandidat(s1, k1, KosOd, ZadnjaNeg − 1);  { vse pred zadnjim neg. številom }
      Kandidat(s2, k2, PrvaNeg + 1, i − 1);     { vse po prvem negativnem številu }
      Kandidat(s1 * s3, k1 + k3, KosOd, i − 1); { cel kos }
      if i <= N then Kandidat(s, k, i, i);       { ničla }
      KosOd := 0; { začenja se nov kos }
    end; {if}
  end; {for i}

  { Izpišimo rezultat. }
  Assign(T, 'produkt.out'); Rewrite(T); WriteLn(T, bOd, ' ', bDo); Close(T);
end. {ProduktStevil}
```

R2002.3.6 Prüferjev kod

Naloga: str. 12

Drevo je mogoče predstaviti na različne načine, katerega izbrati, pa je odvisno od tega, kakšne operacije bomo izvajali nad njim. Nas bo zanimalo vedeti, kdaj smo vozlišču zbrisali že toliko sosedov, da je to vozlišče postalo list; ko postane list, nas zanima, kateri je tisti edini preostali sosed; in ko nato list zbrišemo, moramo v njegovem sosedu evidentirati, da ima ta zdaj enega soseda manj. Zanimalo nas bo tudi, kateri izmed trenutnih listov ima največjo oznako.

Zato je koristna naslednja podatkovna struktura: za vsako vozlišče (struktura VozlisceT v spodnjem programu) hranimo stopnjo (Stopnja) (število sosedov) in kazalec na seznam njegovih sosedov (Sosedje). Ta seznam naj bo dvojno povezan (Prejsnja in Naslednja v strukturi PovezavaT), da bo enostavneje zbrisati iz njega poljuben element. Če je vozlišče a sosed vozlišča b, je tudi b sosed vozlišča a in zato za vsako povezavo dobimo dve strukturi PovezavaT, eno v seznamu a-jevih sosedov in eno v seznamu b-jevih sosedov. Koristno je, če ti dve strukturi kažeta druga na drugo (Nasprotna); tako lahko za vozlišče a z enim samim sosedom b hitro pridemo do strukture, ki predstavlja a-ja v seznamu b-jevih sosedov (tudi to strukturo moramo namreč pobrisati, ko pobrišemo povezavo med a in b).

Vzdrževati moramo tudi množico trenutnih listov, da bomo med njimi lahko izbrali najmanjšega. Spodnji program si pomaga z navadno dvojiško kopico, ki jo hrani v tabeli Listi in z njo dela prek podprogramov ZadnjiList in DodajList. Lahko bi uporabili tudi rdeče-črno drevo ali kaj podobnega. Pri majhnih drevesih, kakršna so tule, bi lahko imeli najbrž tudi navaden seznam listov in ga vsakič v celoti prečesali, da bi videli, kateri je največji.

```
program PruferjevKod;
const MaxN = 10000;
var Listi: array [1..MaxN] of integer; StListov: integer;

  function ZadnjiList: integer;
  var i, ci, x: integer;
  begin
    ZadnjiList := Listi[1]; x := Listi[StListov]; StListov := StListov − 1; i := 1;
    while 2 * i <= StListov do begin
      ci := 2 * i;
      if ci + 1 <= StListov then if Listi[ci + 1] > Listi[ci] then ci := ci + 1;
      if Listi[ci] <= x then break;
```

```
      Listi[i] := Listi[ci]; i := ci;
    end; {while}
    Listi[i] := x;
  end; {ZadnjiList}

  procedure DodajList(x: integer);
  var i, pi: integer;
  begin
    StListov := StListov + 1; i := StListov;
    while i > 1 do begin
      pi := i div 2; if Listi[pi] >= x then break;
      Listi[i] := Listi[pi]; i := pi;
    end; {while}
    Listi[i] := x;
  end; {DodajList}

type
  VozlisceT = record Stopnja, Sosedje: integer end;
  PovezavaT = record Sosed, Prejsnja, Naslednja, Nasprotna: integer end;
var
  T: text; N, i, u, w: integer;
  V: array [1..MaxN] of VozlisceT;
  E: array [1..2 * (MaxN − 1)] of PovezavaT;
begin
  { Preberimo število vozlišč. }
  Assign(T, 'produkt.in');
  Reset(T); ReadLn(T, N);
  for u := 1 to N do begin V[u].Stopnja := 0; V[u].Sosedje := 0 end;

  { Preberimo povezave. }
  for i := 1 to N − 1 do begin
    ReadLn(T, u, w);
    { w postane u-jev sosed. }
    with E[2 * i − 1] do begin Sosed := w; Prejsnja := 0; Naslednja := V[u].Sosedje;
                                Nasprotna := 2 * i end;
    if V[u].Sosedje > 0 then E[V[u].Sosedje].Prejsnja := 2 * i − 1;
    V[u].Sosedje := 2 * i − 1; V[u].Stopnja := V[u].Stopnja + 1;
    { u postane w-jev sosed. }
    with E[2 * i] do begin Sosed := u; Prejsnja := 0; Naslednja := V[w].Sosedje;
                            Nasprotna := 2 * i − 1 end;
    if V[w].Sosedje > 0 then E[V[w].Sosedje].Prejsnja := 2 * i;
    V[w].Sosedje := 2 * i; V[w].Stopnja := V[w].Stopnja + 1;
  end; {for i}

  { Dodajmo liste v kopico. }
  Close(T); StListov := 0;
  for u := 1 to N do if V[u].Stopnja = 1 then DodajList(u);

  { Izpišimo rezultat. }
  Assign(T, 'produkt.out');
  Rewrite(T);
  while N > 2 do begin
    u := ZadnjiList; N := N − 1;
    w := E[V[u].Sosedje].Sosed; WriteLn(T, w);

    { Zbrišimo zapis i, ki pravi, da je u sosed vozlišča w. }
    i := E[V[u].Sosedje].Nasprotna;
```

```
    if E[i].Prejsnja > 0 then E[E[i].Prejsnja].Naslednja := E[i].Naslednja
    else V[w].Sosedje := E[i].Naslednja;
    if E[i].Naslednja > 0 then E[E[i].Naslednja].Prejsnja := E[i].Prejsnja;
    { Če je w postal list, ga dodajmo v kopico. }
    V[w].Stopnja := V[w].Stopnja − 1;
    if V[w].Stopnja = 1 then DodajList(w);
  end; {while}
  Close(T);
end. {PruferjevKod}
```

Razmislimo še o algoritmu za dekodiranje, torej za rekonstrukcijo drevesa iz kodnega zaporedja. Vsako vozlišče s k sosedi se pojavlja v kodnem zaporedju točno $(k-1)$-krat (vsakič, ko eden od njegovih sosedov postane list in ga odtrgamo; ko ostane le en sosed, je naše vozlišče že samo postalo list in takrat ga ne bomo več izpisovali). Torej lahko iz števila pojavitev vozlišč v zaporedju ugotovimo, kakšna je bila stopnja (število sosedov) vsakega vozlišča v prvotnem drevesu. Vemo tudi, da je število vseh vozlišč za 2 večje od dolžine kodnega zaporedja. Zdaj torej ni težko ugotoviti, kateri je bil v prvotnem drevesu list z največjo oznako, iz prvega števila v kodnem zaporedju pa zdaj tudi vemo, kdo je bil takrat njegov sosed. Potem lahko v mislih ta list vzamemo iz drevesa, stopnjo soseda zmanjšamo in zdaj s pomočjo drugega števila v zaporedju ugotovimo, kdo je bil drugi zbrisani list in na koga je bil povezan.

Za hranjenje listov in ugotavljanje, kateri ima trenutno najvišjo oznako, lahko uporabimo prav takšno kopico kot zgornji program za kodiranje. Spodnji postopek izpiše za vsako povezavo drevesa po eno vrstico z dvema številoma, ki predstavljata krajišči te povezave. Na koncu, ko smo obdelali že vseh $n-2$ členov kodnega zaporedja, nam ostaneta od drevesa le še dva lista, ki sta morala biti torej tudi povezana s povezavo, tako da na koncu izpišemo še to.

Algoritem *Refürp:*
Vhod: kodno zaporedje $\langle a_1, \ldots, a_{n-2} \rangle$
Recimo, da so oznake vozlišč $1, \ldots, n$.

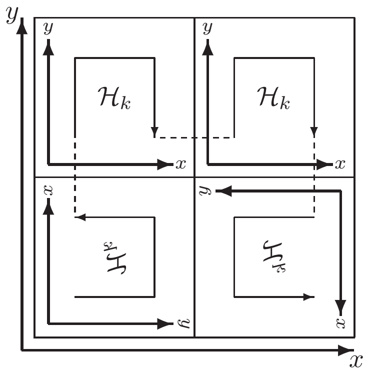
```
  for u := 1 to n do Stopnja[u] := 1;
  for i := 1 to n − 2 do Stopnja[a[i]] := Stopnja[a[i]] + 1;
  for u := 1 to n do if Stopnja[[u]] = 1 then DodajList(u);
  for i := 1 to n − 2 do
    u := ZadnjiList;
    WriteLn(u, ' ', a[i]);
    Stopnja[a[i]] := Stopnja[a[i]] - 1;
    if Stopnja[a[i]] = 0 then DodajList(u);
  u := ZadnjiList; v := ZadnjiList;
  WriteLn(u, ' ', v);
```

Pred tekmovanjem smo razmišljali o tem, da bi v nalogi opisali postopek za kodiranje, od tekmovalca pa zahtevali, naj napiše program za dekodiranje, potem pa smo to opustili, češ da bi bilo pretežko in smo v nalogi raje zahtevali izvedbo programa za kodiranje. Tule pa je pravzaprav videti, da bi bil program za dekodiranje znatno preprostejši (odpade zapletena podatkovna struktura za predstavitev drevesa); no, res pa bi bila naloga težja v tem smislu, da bi se morali tekmovalci postopka za dekodiranje šele domisliti, ker jim ne bi bil že podan.

R2002.3.7 Hilbertova krivulja

Naloga: str. 13

Pomagamo si lahko z dejstvom, da je krivulja $\mathcal{H}_{k+1}$ sestavljena iz štirih kopij krivulje $\mathcal{H}_k$ in zato najprej obišče vse točke v spodnji levi četrtini svoje mreže, nato vse v zgornji levi, nato v zgornji desni in končno vse v spodnji desni. Za pretvorbo koordinat v številko točke moramo izračunati njeno številko znotraj ustrezne kopije krivulje $\mathcal{H}_k$ in prišteti število točk v tistih delih mreže, ki jih $\mathcal{H}_{k+1}$ obišče še prej.[5]



Upoštevati moramo, da ima vsaka kopija krivulje $\mathcal{H}_k$ svoj posebni koordinatni sistem, v katerem je izhodišče drugje kot pri koordinatnem sistemu krivulje $\mathcal{H}_{k+1}$, pa tudi osi sta lahko obrnjeni drugače.

Spodnji program zna tudi pretvarjati številke točk v koordinate (če kot a dobi negativno število, si b razlaga kot številko točke in izpiše njene koordinate). Tu moramo ugotoviti, v kateri četrtini mreže leži točka (vsaka četrtina mreže $2^k \times 2^k$ ima 4^{k-1} točk), nato pa jo lahko pretvorimo v koordinate ustrezne kopije krivulje $\mathcal{H}_k$.

```
program Hilbert;

  function Kodiraj(k, x, y: longint): longint;
  var h, j: longint;
  begin
    if k = 0 then begin Kodiraj := 0; exit end;
    h := 1 shl (k − 1); j := h * h;
    if (x < h) and (y < h) then Kodiraj := Kodiraj(k − 1, y, x)
    else if (x < h) then Kodiraj := j + Kodiraj(k − 1, x, y − h)
    else if (y < h) then Kodiraj := 3 * j + Kodiraj(k − 1, h − 1 − y, 2 * h − 1 − x)
    else Kodiraj := 2 * j + Kodiraj(k − 1, x − h, y − h);
  end; {Kodiraj}

  procedure Dekodiraj(k, i: longint; var x, y: longint);
  var h, j, xx, yy: longint;
  begin
    if k = 0 then begin x := 0; y := 0; exit end;
    h := 1 shl (k − 1); j := h * h; Dekodiraj(k − 1, i mod j, xx, yy);
    if i < j then begin x := yy; y := xx end
    else if i < 2 * j then begin x := xx; y := yy + h end
    else if i < 3 * j then begin x := xx + h; y := yy + h end
    else begin x := 2 * h − 1 − yy; y := h − 1 − xx end;
  end; {Dekodiraj}

var T: text; k, a, b: longint;
begin
```

[5]Mimogrede, Hilbertovo krivuljo lahko smiselno posplošimo tudi na tri ali več dimenzij in sestavimo postopek, ki bo znal za poljuben d pretvarjati med indeksi in koordinatami na d-razsežnih Hilbertovih krivuljah. (T. Bially: *Space-filling curves: Their generation and their application to bandwidth reduction*, IEEE Trans. on Inf. Theory, IT-15(6):658–664, Nov. 1969; W. Gilbert: *A cube-filling Hilbert curve*, Math. Intelligencer 6(3):78, 1984, `http://www.math.uwaterloo.ca/~wgilbert/Research/HilbertCurve/HilbertCurve.html`).

```
  Assign(T, 'produkt.in'); Reset(T); ReadLn(T, k, a, b); Close(T);
  Assign(T, 'produkt.out'); Rewrite(T);
  if a >= 0 then WriteLn(T, Kodiraj(k, a, b))
  else begin Dekodiraj(k, b, a, b); WriteLn(T, a, ' ', b) end;
  Close(T);
end. {Hilbert}
```

R2002.3.8 Slovar

Naloga: str. 14

Slovar lahko predstavimo z neusmerjenim grafom, v katerem vsaki besedi ustreza ena točka, dve točki pa sta povezani, če se njuni besedi razlikujeta le v eni črki. Problem, ki ga rešujemo, se tako prevede v problem iskanja najkrajših poti (najkrajših po številu povezav) in ga lahko rešujemo na primer z iskanjem v širino.

Za gradnjo grafa imamo več možnosti: (1) Naivni pristop, ki primerja vsak par besed in preveri, če se razlikujeta le v enem mestu — če je besed veliko, bo trajalo to predolgo časa. (2) Za vsako dolžino besed, npr. d, in za vsak i od 1 do d, vzamemo vse besede dolžine d in jih uredimo, pri čemer se ob primerjanju besed delamo, kot da i-te črke ni. Tako pridejo skupaj tiste, ki se razlikujejo le v eni črki. (3) Za vsako dolžino besed d in za vsak i od 1 do d pripravimo razpršeno tabelo, v katero dodamo vse besede dolžine d, le da se pri tem delamo, kot da jim manjka i-ta črka. Tako spet lahko opazimo, če se kakšne besede razlikujejo le v eni črki.

```
program Slovar;
const
  MinDolz = 2; MaxDolz = 10;
  MaxN = 10000; MaxStopnja = 30;
  MaxSkupDolz = 55000;
  RazpMax = 10037; { primerno veliko praštevilo }
type
  SosedeT = packed array [1..MaxStopnja * MaxN] of integer;
  SosedeP = ↑SosedeT;
var
  T, TT: text; S, S1, S2: string;
  i, i2, j, k, d, dNasl, iOd, iDo, Izp, L, N, hc, Prehod, StPrimerov: integer;
  { Črke vseh besed, zbite skupaj v eno dolgo tabelo;
    besedi i pripadajo Crke[PrvaCrka[i]..PrvaCrka[i] + Dolzina[i] − 1]. }
  Crke: packed array [1..MaxSkupDolz + MaxDolz] of char;
  PrvaCrka, Dolzina: array [0..MaxN] of word;
  { Sosede[d] kaže na tabelo s seznami sosed za vse besede dolžine d;
    teh sosed je vsega skupaj SkupajSosed[d]. Sosede besede i so v celicah
    Sosede[Dolzine[i]]↑[PrvaSoseda[i]..PrvaSoseda[i] + StSosed[i] − 1]. }
  Sosede: array [MinDolz..MaxDolz] of SosedeP;
  SkupajSosed: array [MinDolz..MaxDolz] of word;
  StSosed, PrvaSoseda: array [0..MaxN] of word;
  { Razpršena tabela (za lažje odkrivanje enakih besed). Reze[hc]
    kaže na prvo besedo z razpršilno kodo hc; če je to beseda i,
    kaže potem Verige[i] na naslednjo, Verige[Verige[i]] na še
    naslednjo in tako naprej. Konec verige označuje indeks 0. }
  Reze: array [0..RazpMax − 1] of integer;
  Verige: array [1..MaxN] of integer;
  { Vrsta za iskanje v širino. Trenutna beseda je na oddaljenosti
```

```
    d od začetne besede, tiste, ki so v vrsti od indeksa dNasl naprej,
    pa so že na oddaljenosti d + 1. }
  Vrsta, Obiskana: array [1..MaxN] of integer; Glava, Rep: integer;

  { Izračuna razpršilno kodo dane besede (z eno izpuščeno črko).
    Takšno kodo se lahko uporabi kot indeks v tabelo Reze. }
  function Razprsi(StBesede, IzpuscenaCrka: integer): integer;
  var r: longint; i: integer;
  begin
    r := 0; for i := 1 to Dolzina[StBesede] do if i <> IzpuscenaCrka then
      r := (r * 256 + Ord(Crke[PrvaCrka[StBesede] + i - 1])) mod RazpMax;
    Razprsi := r;
  end; {Razprsi}

  function Enaki(StBesede1, StBesede2, IzpuscenaCrka: integer): boolean;
  var i: integer;
  begin
    Enaki := false; if Dolzina[StBesede1] <> Dolzina[StBesede2] then exit;
    for i := 1 to Dolzina[StBesede1] do if i <> IzpuscenaCrka then
      if Crke[PrvaCrka[StBesede1] + i - 1] <> Crke[PrvaCrka[StBesede2] + i - 1]
      then exit;
    Enaki := true;
  end; {Enaki}

begin
  Assign(T, 'slovar.in');
  Reset(T); ReadLn(T, N);
  { Preberimo slovar. }
  for i := 1 to N do begin
    ReadLn(T, S); L := Length(S); Dolzina[i] := L;
    if i = 1 then PrvaCrka[i] := 1
    else PrvaCrka[i] := PrvaCrka[i - 1] + Dolzina[i - 1];
    for j := 1 to L do Crke[PrvaCrka[i] + j - 1] := S[j];
  end; {for i}
  { Za vsako besedo pripravimo seznam sosed. To naredimo v dveh
    prehodih; najprej bomo sosede samo prešteli, v drugem prehodu
    pa jih bomo zapisali v tabele Sosede. }
  for Prehod := 1 to 2 do begin
    for i := 1 to N do StSosed[i] := 0;
    for d := MinDolz to MaxDolz do for Izp := 1 to d do begin
      { Začnimo s prazno razpršeno tabelo. }
      for i := 0 to RazpMax - 1 do Reze[i] := 0;
      for i := 1 to N do Verige[i] := 0;
      { Preglejmo vse besede dolžine d. }
      for i := 1 to N do if Dolzina[i] = d then begin
        { V razpršeni tabeli poglejmo, katere se ujemajo z i-to
          v vseh črkah razen na mestu Izp. }
        hc := Razprsi(i, Izp); i2 := Reze[hc];
        while i2 <> 0 do begin
          if Enaki(i, i2, Izp) then begin
            { i2 je res soseda besede 1. }
            if Prehod = 2 then begin { dodajmo ju v seznam sosed }
```

```
            Sosede[d]↑[PrvaSoseda[i] + StSosed[i]] := i2;
            Sosede[d]↑[PrvaSoseda[i2] + StSosed[i2]] := i;
          end; {if Prehod = 2}
          StSosed[i] := StSosed[i] + 1; StSosed[i2] := StSosed[i2] + 1;
        end; {if Enaki}
        i2 := Verige[i2]; { naprej po seznamu besed z razpršilno kodo hc }
      end; {while}
      { Dodajmo še besedo i v razpršeno tabelo. }
      Verige[i] := Reze[hc]; Reze[hc] := i;
    end; {for i}
  end; {for d, Izp}
  if Prehod = 1 then begin
    for d := MinDolz to MaxDolz do SkupajSosed[d] := 0;
    for i := 1 to N do begin
      PrvaSoseda[i] := SkupajSosed[Dolzina[i]] + 1;
      SkupajSosed[Dolzina[i]] := SkupajSosed[Dolzina[i]] + StSosed[i];
    end; {for i}
    for d := MinDolz to MaxDolz do if SkupajSosed[d] > 0 then
      GetMem(Sosede[d], SkupajSosed[d] * SizeOf(integer));
  end; {if Prehod = 1}
end; {for Prehod}
{ Pomečimo zdaj vse besede v razpršeno tabelo,
  da jih bomo pri odgovarjanju na poizvedbe lažje našli. }
for i := 0 to RazpMax − 1 do Reze[i] := 0;
for i := 1 to N do begin Verige[i] := 0; Obiskana[i] := 0 end;
for i := 1 to N do
  begin hc := Razprsi(i, 0); Verige[i] := Reze[hc]; Reze[hc] := i end;
{ Začnimo iskati najkrajše poti med danimi pari besed.
  Iskane besede bomo pomožno obravnavali pod številko 0. }
PrvaCrka[0] := PrvaCrka[N] + Dolzina[N];
Assign(TT, 'slovar.out');
Rewrite(TT); ReadLn(T, StPrimerov);
for j := 1 to StPrimerov do begin
  ReadLn(T, S); i := 1; while S[i] <> ' ' do i := i + 1;
  S1 := Copy(S, 1, i − 1); S2 := Copy(S, i + 1, Length(S) − i);
  { Niza S1 in S2 poiščimo v razpršeni tabeli. }
  Dolzina[0] := Length(S1);
  for i := 1 to Length(S1) do Crke[PrvaCrka[0] + i − 1] := S1[i];
  iOd := Reze[Razprsi(0, 0)]; while not Enaki(0, iOd, 0) do iOd := Verige[iOd];
  Dolzina[0] := Length(S2);
  for i := 1 to Length(S2) do Crke[PrvaCrka[0] + i − 1] := S2[i];
  iDo := Reze[Razprsi(0, 0)]; while not Enaki(0, iDo, 0) do iDo := Verige[iDo];
  { Z iskanjem v širino iščemo pot od besede iOd do besede iDo. }
  Obiskana[iOd] := j; Glava := 1; Rep := 2; Vrsta[Glava] := iOd; d := 0; dNasl := 2;
  while (Glava < Rep) and (Obiskana[iDo] < j) do begin
    if Glava = dNasl then begin d := d + 1; dNasl := Rep end;
    i := Vrsta[Glava]; Glava := Glava + 1;
    { Dodajmo v vrsto i-jeve sosede, če jih nismo že kdaj prej. }
    for k := 1 to StSosed[i] do begin
      i2 := Sosede[Dolzina[i]]↑[PrvaSoseda[i] + k − 1];
      if Obiskana[i2] = j then continue; { Točka i2 je bila že obiskana. }
```

```
        Vrsta[Rep] := i2; Rep := Rep + 1; Obiskana[i2] := j;
      end; {for k}
    end; {while}
    { Zdaj smo ali našli pot do iDo ali pa preiskali vse, do česar se je dalo
      iz iOd sploh priti (in ugotovili, da se do iDo ne da priti). }
    if Obiskana[iDo] = j then WriteLn(TT, d + 1)
    else WriteLn(TT, 'Ni prehoda');
  end; {for j}
  Close(T); Close(TT);
  for d := MinDolz to MaxDolz do if SkupajSosed[d] > 0 then
    FreeMem(Sosede[d], SkupajSosed[d] * SizeOf(integer));
end. {Slovar}
```

Gornji program si prizadeva biti varčen pri porabi pomnilnika in ima zato podatkovne strukture mogoče urejene malo bolj zapleteno, kot bi bilo nujno potrebno. V resnici bi bilo gotovo čisto sprejemljivo tudi, če bi pri razpršeni tabeli in seznamih sosedov uporabili dinamično alocirane zapise in jih povezovali s kazalci.

REŠITVE NALOG ČETRTEGA TEKMOVANJA IZ UNIXA

R2002.U.1 Interval števil predstavimo z urejenim parom (*od*, *do*); spodnji program ima funkcijo interval, da predela niz znakov v takšen urejen par. Pri tem moramo niz `"*"` obravnavati posebej in vrniti interval od 0 do 255. Sicer pa dani niz s funkcijo split razcepimo pri znaku `-` in vsak kos predelajmo v celo število; rezultat je seznam L z enim ali dvema elementoma, odvisno od tega, ali je prvotni niz vseboval znak `-` ali ne. V vsakem primeru nam torej prvi element (L[0]) pove spodnjo mejo, zadnji element (L[−1]) pa zgornjo mejo intervala.

Naloga: str. 15

Niz oblike `0-255.*.255.*` bomo s funkcijo split razrezali pri vseh pikah in vsak kos pretvorili v urejen par, kot je to opisano v prejšnjem odstavku. V spodnjem programu to naredi funkcija intervali. (Izraz [f(x) **for** x **in** L] sestavi seznam vrednosti f(x) za vse x iz seznama L.)

Funkcija presek izračuna, koliko števil vsebuje presek dveh intervalov. Presek intervala od a_1 do a_2 in intervala od b_1 do b_2 je interval od $c_1 := \max\{a_1, b_1\}$ do $c_2 := \min\{a_2, b_2\}$, ki vsebuje $c_2 - c_1 + 1$ elementov. Če pa je presek prazen, bo ta vrednost negativna (ker bo $c_2 < c_1$) in moramo vrniti 0.

Produkt več števil lahko elegantno računamo s funkcijo, kot je produkt v spodnjem programu. Pomagamo si s pythonovo vgrajeno funkcijo reduce(f, L, a), ki vrne vrednost f(⋯f(f(a, L[0]), L[1])⋯, L[len(L) − 1]). Če torej hočemo produkt elementov seznama L, mora biti f funkcija, ki sprejme dva argumenta in vrne njun zmnožek; ravno takšno funkcijo pa sestavi izraz **lambda** x, y: x * y.

Glavni del programa pretvori oba dana niza v seznama intervalov; zdaj moramo za vsak par istoležnih intervalov izračunati, koliko števil je v njunem preseku. To lahko elegantno naredimo s pythonovo funkcijo map(f, L1, L2), ki vrne seznam vrednosti f(L1[i], L2[i]) za vse i od 0 do dolžine seznamov − 1.

Število naslovov, ki so skupni obema danima podomrežjema, je kar produkt velikosti presekov po posameznih komponentah. Če je ta produkt enak 0 ali 1, je to že tudi kar vrednost, ki jo mora vrniti naš program; če pa je produkt večji ali enak 2, vrnemo 2.

```
def interval(s):
  if s == "*": return (0, 255)
  L = [int(t) for t in s.split('-')]
  return (L[0], L[−1])
def intervali(s): return [interval(t) for t in s.split('.')]
def presek(a, b): return max(0, min(a[1], b[1]) − max(a[0], b[0]) + 1)
def produkt(faktorji): return reduce(lambda x, y: x * y, faktorji, 1)

import sys
preseki = map(presek, intervali(sys.argv[1]), intervali(sys.argv[2]))
sys.exit(min(2, produkt(preseki)))
```

Naloga: str. 16

R2002.U.2

Primer rešitve z `bash`em in perlom:

```
#!/bin/bash

uporaba() {
  echo "Uporaba: $0 datoteka" 1>&2
  echo " Program na mestu izreže iz podane datoteke vse znake CR" 1>&2
  echo " (predstavljeni desetiško kot 15, kot ^M ali \r)." 1>&2
}

if [ "$#" != 1 ]; then
  uporaba
  exit 1
fi

perl −pi −e "s/\r//g" "$1"
```

Skripta v lupini `bash` vidi prvi parameter ukazne vrstice kot spremenljivko `$1`, število parametrov pa kot `$#`. Ko se prepričamo, da smo dobili točno en parameter, pokličemo `perl`, da res pobriše znake CR iz dane datoteke. Pri tem mu naročimo, naj dani program ponavlja v zanki, po enkrat za vsako vrstico vhodne datoteke, in izpisuje spremenjene vrstice (stikalo −p); na koncu naj dobljene izhodne podatke napiše kar čez vhodno datoteko (stikalo −i). Naš „program“ v perlu je tu dolg eno samo vrstico in ga podamo kar prek stikala −e. Stavek s/.../.../g zamenja vse pojavitve prvega vzorca z drugim; v našem primeru torej zamenja vse pojavitve znaka CR s praznim nizom in jih tako pobriše.

Lahko pa uporabimo tudi program `tr` in mu s stikalom −d naročimo, naj pobriše vse pojavitve določenih znakov (v našem primeru znaka CR). Ker pa dela `tr` le s standardnim vhodom in izhodom, moramo sami poskrbeti za pomožno datoteko. Da ne bi več uporabnikov ali procesov hkrati uporabljalo iste pomožne datoteke, dodajmo v njeno ime tudi številko trenutnega procesa, ki jo v `bash`u dobimo v spremenljivki `$$`. Na koncu s programom `mv` zapišemo pomožno datoteko čez prvotno.

```
tr −d '\r' < "$1" > "/tmp/$1-$$"
mv "/tmp/$1-$$" "$1"
```

Naloga: str. 16

R2002.U.3

Podatke o procesih dobimo od programa `ps`; hočemo podatke o *vseh* procesih (stikali `a` in `x`), brez imen stolpcev v prvi vrstici (stikalo `h`); obliko izpisa mu določimo sami (stikalo `o`), in sicer hočemo za vsak proces njegovo številko (`pid`) in številko njegovega očeta (`ppid`).

Naš program bo bral, kar je ps izpisal; v vsaki vrstici imamo podatke o enem procesu. V razpršeni tabeli otroci bomo za vsak proces vzdrževali seznam njegovih otrok. Procesi tvorijo drevo, čigar koren je proces init s številko 1 (torej prav takšno drevo, kot ga izpiše program pstree in je prikazano pri besedilu naloge na str. 16). Globino lahko računamo rekurzivno: globina drevesa je za eno večja kot globina najglobljega izmed njegovih poddreves. Na koncu vrnemo globina(1), torej globino celotnega drevesa procesov.

```
#!/usr/bin/python
import sys, os

otroci = {}

def globina(proces):
    if not proces in otroci: return 1    # nima otrok
    return 1 + max([globina(otrok) for otrok in otroci[proces]])

for vrstica in os.popen("ps axho pid,ppid", "r"):
    s = vrstica.split()
    proces = s[0]; oce = s[1]
    if not oce in otroci: otroci[oce] = [ ]
    otroci[oce].append(proces)
sys.exit(globina(1))
```

R2002.U.4

Naloga: str. 17

Pomagali si bomo s pogojnimi stavki v lupini bash. V spremenljivkah $1 in $2 dobimo prva dva parametra iz ukazne vrstice, v $# pa število teh parametrov. V primerjalnih izrazih lahko z operatorjem −f preverimo, če je določen niz res ime kakšne datoteke; operator −o deluje kot logični ali, operator ! pa pomeni negacijo. Z operatorjem −ot pa preverimo, če je neka datoteka starejša od druge.

```
#!/bin/bash

if (( $# != 2 )); then
    exit 3
fi

if [ ! -f "$1" -o ! -f "$2" ]; then
    exit 3
fi

if [ "$1" -ot "$2" ]; then
    exit 1
elif [ "$2" -ot "$1" ]; then
    exit 2
else
    exit 0
fi
```

Viri nalog za leto 2002: limuzine — Matija Grabnar; najdaljši cikel, prijatelji in sovražniki, število vsot, Prüferjev kod — Jure Leskovec; TiVo — Mark Martinec; bencin, uvrstitve tekmovalcev — Marjan Šterk; razbijanje kode, razbijanje gesel — Miha Vuk; pristanišče, slovar — Anže Žagar; kodiranje, sestanki — Klemen Žagar; produkt števil — po zgledu ACM SEERC 1996, naloga C, #787 na online-judge.uva.es; Hilbertova krivulja — Janez Brank. TiVo je nek proizvajalec osebnih videorekorderjev (*www.tivo.com*). Zahvale ljudem, ki so implementirali rešitve tujih nalog za 3. skupino: Blažu Fortuni za produkt števil; Blažu Novaku za Prüferja; Marjanu Šterku za Hilberta.

Tekmovanje v poznavanju Unixa 2002 so pripravili: Saša Divjak, Jure Koren, Aleš Košir, Rok Kaver, Rok Papež, Primož Peterlin, Marko Samastur, Andraž Tori in Miha Tomšič.