Tekmovanja v poznavanju Unixa

LETO 1999, TEKMOVANJE V POZNAVANJU UNIXA

Pravila

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

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

1999.U.1 Besedna analiza

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

Rešitev: str. 9

1999.U.2 `vi`

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

Rešitev: str. 11

1999.U.3 Premešaj

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

Rešitev: str. 13

1999.U.4 Številke IP

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

Rešitev: str. 14

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

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

V datoteki razen takih zapisov ni ničesar drugega.

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

LETO 2000, TEKMOVANJE V POZNAVANJU UNIXA

Pravila

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

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

Rešitev: str. 15

2000.U.1

Napiši programček, ki bere vhodno datoteko in jo na standarno izhodno enoto izpiše tako preurejeno, da so besede v vsaki vrstici razvrščene v obratnem vrstnem redu kot v izvorni datoteki.[1] Besede so v vrstici medsebojno ločene s po enim presledkom. Datoteko z vsebino

```
prva druga tretja
alfa beta gama delta
```

naj programček izpiše takole preurejeno:

```
tretja druga prva
delta gama beta alfa
```

Rešitev: str. 16

2000.U.2

Na datotečnem sistemu našega strežnika moramo vsako noč pobrisati vse datoteke `core`, ki so nastale čez dan. Za izvajanje ob določeni uri poskrbi ukaz `cron`, mi pa moramo napisati skripto, ki se pokliče enkrat dnevno in pobriše vse datoteke `core`. Strežnik ima samo en datotečni sistem, zato lahko iščeš od korenskega imenika naprej. Vse datoteke `core` imajo v imenu besedico `core`, zaneseš pa se lahko tudi na to, da program `file` pravilno prepozna vse tipe datotek in da imena datotek ne vsebujejo nenavadnih znakov, samo `A`–`Z`, `a`–`z`, `0`–`9` in `_`.

Rešitev: str. 17

2000.U.3

Opazili ste, da nekateri programi zelo slabo tvorijo datoteke v zapisu HTML, tako da datoteke vsebujejo prazne elemente, kakršen je na primer element `<B></B>`. Sestavi skripto, ki bo iz datoteke `.html` odstranil vse prazne oznake.

Pri tem smeš predpostaviti:

- oznaki za začetek in konec elementa vselej nastopata v isti vrstici;
- znaka `<` in `>` ne nastopata v datoteki nikjer, razen v oznakah elementov;

Tekmovanje v poznavanju Unixa 2000 so pripravili: Aleš Košir, Jure Koren, Roman Maurer, Borut Mrak, Primož Peterlin, Marko Samastur in Boštjan Slivnik.

[1] Podobna naloga je tudi 1998.1.2.

- različne oznake se skladno s standardom HTML ne križajo;
- oznake za začetek in konec elementov so vselej zapisane z enakimi črkami, na primer takole: `<oZnaKa></oZnaKa>`;
- elementi niso nikjer gnezdeni tako, kot kaže primer: `<PRVA><DRUGA></DRUGA></PRVA>`.

2000.U.4

Rešitev: str. 17

Zapiši vse permutacije besed, ki so podane v edini vrstici vhodne datoteke. Besede so medsebojno ločene s po enim presledkom in so različne. Permutacije lahko izpišeš v poljubnem vrstnem redu. Njihovo število je enako $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$, če je n število besed.

Vse permutacije dveh besed `dan` in `noč` so videti takole:

```
dan noč
noč dan
```

Pri treh besedah `jutro`, `dan` in `noč` pa so permutacije:

```
dan jutro noč
dan noč jutro
jutro dan noč
jutro noč dan
noč dan jutro
noč jutro dan
```

LETO 2001, TEKMOVANJE V POZNAVANJU UNIXA

2001.U.1

Rešitev: str. 20

Napiši program, ki bo kot argument vzel imeni dveh obstoječih datotek in ne bo nič izpisal, če bo vsebina datotek popolnoma enaka; če bo vsebina različna, pa bo izpisal na standardni izhod neprazno vrstico. Predpostavi, da argumenta označujeta dve datoteki in da ti zanesljivo obstajata.

Primer:

```
./naloga1 datoteka1 datoteka2
```

2001.U.2

Rešitev: str. 20

Zaradi vse večjega števila uporabnikov internetnih storitev in pomanjkljivih varnostnih ukrepov se je razpaslo kar nekaj virusov, ki se razmnožujejo tako, da se razpošljejo z elektronsko pošto. Uporabnik Janez ima v imeniku pisma shranjena v zapisu `Maildir`, pri katerem je vsako sporočilo shranjeno v svoji datoteki. Vsako sporočilo ima glavo in telo. Glava sporočila je od telesa ločena s prazno vrstico. Naslov sporočila je v glavi sporočila in se prične z nizom `Subject:`, ki je čisto na začetku vrstice. Naslov se ne razteza čez več vrstic.

Napiši program, ki kot argument sprejme ime datoteke, jo pregleda in izpiše znak 1, če je sporočilo okuženo z virusom, sicer pa ne izpiše nič.

Sporočilo pa je okuženo, če se naslov sporočila prične z nizom `I LOVE YOU`.

Rešitev: str. 20

2001.U.3

Napiši program, ki prešteje vse izvedljive datoteke glede na vsebino spremenljivke okolja `$PATH`. Razmisli o tem, da je v poti kateri od imenikov lahko naveden večkrat. Nekatere datoteke morda lahko izvaja samo sistemski skrbnik, mi pa ne; takih ne smemo šteti. V imenikih so poleg navadnih datotek tudi simbolne povezave na druge datoteke, ki jih moramo tudi všteti. Ne smemo pa seveda šteti simbolnih povezav, ki kažejo na neobstoječe datoteke ali na datoteke, za katere nimamo dovoljenja, da bi jih poganjali.

Rešitev: str. 21

2001.U.4

V datoteki `stevila.txt` je $2n$ nenegativnih celih števil, vsako v svoji vrstici.

Napiši program, ki bo iz njih tvoril n parov števil tako, da bo vsota zmnožkov parov števil najmanjša.

V izhodni datoteki morata biti števili v paru ločeni s presledkom, vsak par pa mora biti v novi vrstici. Vrstni red parov v izhodni datoteki ni pomemben.

Vhodna datoteka se imenuje `stevila.txt`. Primer vhodne datoteke:

```
7
4
0
2
2
1
```

Izhodna datoteka (torej tista, ki jo mora narediti tvoj program), naj se imenuje `pari.txt`.

Primer izhodne datoteke (`pari.txt`):

```
0 7
1 4
2 2
```

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.

Rešitev: str. 23

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.

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.

2002.U.2

Rešitev: str. 23

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.

2002.U.3

Rešitev: str. 24

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. 25

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.

LETO 2003, TEKMOVANJE V POZNAVANJU UNIXA

Nasvet

Pri vaših rešitvah bo komisija za ocenjevanje upoštevala poleg pravilnosti in robustnosti tudi njihovo elegantnost in preprostost.

Rešitev: str. 25

2003.U.1

V okolju UNIX je obdelava znakovnih podatkov zelo pomembna. Strežniški programi običajno vodijo datoteko dogodkov — dnevnik (*logfile*).

Napišite program `preglej`, ki mu podamo kot parametre imena treh datotek po vrsti, kot kaže zgled:

```
preglej datoteka1.log regexp-da.dat regexp-ne.dat
```

Prva datoteka je dnevniška z običajnim besedilom (*text file*) in vsebuje zapise dogodkov. Drugi datoteki vsebujeta vsaka po eno vrstico z regularnim izrazom. Dnevnik je potrebno prebrskati in izpisati vse vrstice, ki ustrezajo regularnemu izrazu iz prve datoteke ter ne ustrezajo regularnemu izrazu iz druge datoteke. Če morebiti katera od podanih datotek ne obstaja, naj program izpiše besedilo „`NAPAKA`“ v vrstici na standardni izhod.

Namig: Privzamete lahko, da je dnevniška datoteka vselej dostopna in na voljo, da se med tekom skripte ne dodajajo novi zapisi v dnevnik in da se skripte nikoli ne poganja večkrat hkrati.

Rešitev: str. 25

2003.U.2

Nekatere datoteke z večpredstavnimi vsebinami imajo glavo, ki se začne z nizom „`RIFF`“, temu pa lahko sledijo poljubni podatki, ki se končajo z nizom „`data`“. Za nizom „`data`“ je preostanek datoteke.

Napiši program, ki mu kot parameter podaš ime tako oblikovane datoteke, program pa bo iz nje izrezal vse podatke med nizoma „`RIFF`“ in „`data`“, vključno s tema nizoma. Če na začetku datoteke ni niza „`RIFF`“, naj ne odstani podatkov.

Rešitev: str. 26

2003.U.3

Sistemi Linux v navideznem datotečnem sistemu `/proc` hranijo mnoge podatke o stanju sistema in programov, ki tečejo v njem. Z branjem teh podatkov znajo programi prikazati stanje sistema na različne načine.

Napišite program, ki na standardni izhod izpiše polno pot do izvršilne datoteke očeta in njeno ime. Oče je proces, ki je pognal program, ki ste ga napisali. Če kot zgled v ukazni lupini `bash` poženemo nek program `preskus` s parametri

```
preskus -v test -o izhod.dat
```

in bi ta program hotel izpisati pot do izvršilne datoteke svojega očeta, bi moral izpisati pot do lupine `/bin/bash`. Če pa bi napisali svoj program za izpisovanje poti do izvršilne datoteke očeta in ga pognali iz programa

```
preskus -v test -o izhod.dat
```

bi moral naš program izpisati pot, kjer je na disku shranjena izvršilna datoteka programa `preskus`, denimo `/usr/local/bin/preskus`. V primeru

```
/usr/bin/preskus
```

bi to bil niz

```
/usr/bin/preskus
```

V primeru

```
./preskus
```

pa je to lahko nekaj takega:

```
/home/uporabnik/preskus
```

2003.U.4

Rešitev: str. 28

Nekateri programi so napisani tako, da delujejo kot filtri (berejo podatke s standardnega vhoda in pišejo na standardni izhod), drugim pa moramo podati vhodno in izhodno datoteko.

Program `sort`, denimo, zna delati celo na oba načina.

```
sort datoteka1 -o datoteka2
sort datoteka1 > datoteka2
```

Razliko opazimo takrat, ko sta `datoteka1` in `datoteka2` ista datoteka. Takrat druga oblika ukaza poreže datoteko, preden je urejanje končano in izgubimo njeno vsebino.

Napiši program `prepis`, ki bo prepisal vhodno datoteko na izhodno, podobno kot `cat datoteka1 > datoteka2`, na začetek datoteke pa bo dodal niz „`PREPIS`“.

Če podamo le en parameter, bo program podano datoteko izpisal na standardni izhod. Če mu podamo izhodno datoteko, bo izhod pisal nanjo. V tem primeru se bo tudi pametno odzval, kadar sta podani datoteki ista datoteka, kar pomeni, da bo vhodni datoteki na njen začetek dodal zahtevani niz.

LETO 2004, TEKMOVANJE V POZNAVANJU UNIXA

Rešitev: str. 29

2004.U.1

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

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

Da bi uporabniku pri tem pomagali, sestavite program, ki bo za vneseno besedo izpisal vse možnosti zamenjave dveh sosednjih črk, recimo za *bal*: *abl*, *bal*, *bla*.[2]

Program naj besede prebere iz vhodne datoteke, ki je navedena kot parameter v ukazni vrstici. V vsaki vrstici datoteke je podana ena beseda. Datoteka ne vsebuje presledkov ali nečrkovnih znakov, le črke in znake za konec vrstice. Izpisane besede naj bodo urejene bo abecednem vrstnem redu. Morebitne prazne vrstice v vhodni datoteki preskoči.

Rešitev: str. 30

2004.U.2

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

Dnevniška datoteka je videti takole:

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

Rešitev: str. 31

2004.U.3

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

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

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

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

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

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

2004.U.4

Rešitev: str. 32

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

Sintaksa:

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

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

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

REŠITVE NALOG PRVEGA TEKMOVANJA IZ UNIXA

R1999.U.1

Naloga: str. 1

Pomagali si bomo s programi `tr`, `sort` in `uniq`, ki jih bomo povezali s cevmi (kar pomeni, da ukazna lupina ob poganjanju teh programov poskrbi za to, da se standardni izhod enega programa spelje na standardni vhod naslednjega in tako naprej). Rešitev je lahko takšna:

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

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

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

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

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

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

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

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

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

Sledi primer rešitve z razpršeno tabelo:

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

Za razbijanje vrstice na besede (pri presledkih) smo uporabili pythonovo funkcijo split. Pri dostopu do razpršene tabele moramo posebej obravnavati primer, ko na neko besedo naletimo prvič; takrat je v razpršeni tabeli še ni. Stavek

frekvence[beseda] += 1 sproži takrat izjemo (*exception*), ker bi moral najprej prebrati iz razpršene tabele vrednost, ki pripada tej besedi, da bi jo nato lahko povečal za 1. To izjemo prestrežemo (stavek **try**...**except**) in v tem primeru preprosto vpišemo besedo v razpršeno tabelo z začetno frekvenco 1 (ker smo pravkar videli njeno prvo pojavitev). Namesto te rešitve s **try**...**except** bi lahko tudi eksplicitno preverili, če je beseda že v razpršeni tabeli:

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

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

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

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

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

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

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

R1999.U.2

Naloga: str. 1

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

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

```
#!/usr/bin/perl

use Fcntl ':flock';   # definicija konstant LOCK_*

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

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

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

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

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

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

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

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

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

Pomembno je, da ne gremo najprej preverjat, če datoteka že obstaja, in jo šele nato poskusimo ustvariti; če bi počeli tako, bi nas lahko med našim preverjanjem in ustvarjanjem datoteke prehitel kdo drug, ki bi ravno tako opazil, da še ne obstaja, in jo nato ustvaril, še preden bi jo ustvarili mi. Namesto tega bomo datoteko kar takoj poskusili ustvariti, nato pa bomo samo pogledali, če se je to posrečilo ali ne.

```
#!/bin/bash
if ( umask u=rw,g=,o= ; touch $1.lock > /dev/null 2>&1 )
then
  vi_old $1
```

```
  rm -f $1.lock
else
  echo "Datoteka $1 je trenutno zaklenjena."
fi
```

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

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

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

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

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

Mimogrede omenimo še, da za razliko od prvotne različice programa `vi` nekatere novejše različice, na primer zelo razširjeni `vim` („`vi` improved“), že same po sebi skrbijo tudi za zaklepne datoteke.

R1999.U.3

Naloga: str. 1

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

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

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

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

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

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

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

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

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

Za urejanje uporabimo program `sort`, ki mu s stikalom `−n` naročimo, naj začetke vrstic pri urejanju gleda kot števila (iz enakih razlogov kot pri 1. nalogi, glej str. 9). Na koncu bi radi tista naključna števila z začetkov vrstic pobrisali, kar lahko naredimo s programom `cut`. Ta naj razreže vsako vrstico pri presledkih (`−d " "`) in izpiše vse kose od drugega naprej (`−f 2−`); zavržemo torej le prvi kos, ki vsebuje naključno število, ki smo ga dodali pred urejanjem.

Naloga: str. 1

R1999.U.4

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

```
#!/usr/bin/perl
open(HOSTS_FILE, "/etc/hosts");
while ($line = <HOSTS_FILE>) {
  chomp $line;
  ($ip, $fqdn) = split(/ +/, $line);
  $hostbyip {$ip} = $fqdn;
}
close(HOSTS_FILE);
while (<>) {
  s/\b(\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3})\b/$hostbyip{$1}/g;
  print;
}
```

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

Za iskanje in zamenjevanje številskih naslovov z imeni uporabljamo operator s/*vzorec*$_1$/*vzorec*$_2$/*zastavice*, ki zamenja pojavitve vzorca 1 z vzorcem 2; zastavica `g` zahteva zamenjavo vseh pojavitev, ne pa le prve. Pri regularnem izrazu, ki naj bi odkrival številske naslove IP, upoštevajmo naslednje: pred piko je treba postaviti poševnico \, ker drugače pika sama po sebi deluje kot metaznak, ki se ujame s poljubnim znakom pregledovanega niza; metaznak `\d` se ujame s katero koli številko; metaznak `\b` se ujame z nizom dolžine 0, vendar le na robovih besed. Tisti del regularnega izraza, ki se bo ujel s številskim naslovom, postavimo v oklepaje; tako se bomo lahko na ta del vhodnega niza sklicevali še iz zamenjavnega vzorca (s spremenljivko $1). Zamenjavni vzorec $hostbyip{$1} se torej razširi v niz, ki v razpršeni tabeli hostbyip pripada ključu $1; ker je slednji ravno številski naslov IP, se bo zamenjavni vzorec razširil v pripadajoče ime računalnika.

REŠITVE NALOG DRUGEGA TEKMOVANJA IZ UNIXA

R2000.U.1

Rešitev v `awk`:

Naloga: str. 2

```
{
  for (i = NF; i > 0; i--)
    printf "%s ", $i;
  print "";
}
```

`awk` si misli, da je vhod sestavljen iz zaporedja zapisov (*records*), ločenih z ločilnim znakom; če mu ne povemo drugače, je to znak za konec vrstice, torej so zapisi kar vrstice besedila. Skripta je sestavljena iz pravil (*rules*), vsako pravilo pa iz vzorca (*pattern*) in dejanja (*action*), ki naj se izvede, če trenutni zapis ustreza vzorcu. Naša zgornja skripta ima le eno pravilo, ki pa nima vzorca, zato se izvede na vsakem zapisu. `awk` razdeli vsak zapis na polja (*fields*); če mu ne povemo drugače, razdeli pri presledkih, torej je vsako polje pravzaprav ena beseda. Pri vsakem zapisu postavi vrednost spremenljivke NF na število polj; do posameznih polj lahko pridemo z izrazi $1, $2, ..., $NF. Stavek printf

izpiše dane vrednosti v skladu z danim opisom formata, podobno kot funkcija printf v C/C++. Stavek print pa izpiše svoje argumente, ločene s presledki, na koncu pa izpiše še znak za konec vrstice. Naša „akcija“ gre torej z zanko **for** po vseh besedah v obratnem vrstnem redu in vsako izpiše, med njimi presledke, nazadnje pa še znak za konec vrstice. Ker jo `awk` izvede pri vsaki vrstici posebej, je rezultat ravno to, kar je naloga zahtevala.

Če je vrstica ena sama, gre tudi z ukazom v lupini:

```
tr " " "\n" | tac | tr "\n" " "
```

Program `tr` bere podatke s standardnega vhoda, spreminja nekatere znake in spremenjeno besedilo izpisuje na standardni vhod. Niza, ki ju dobi kot parametra, mu povesta, kako naj spreminja znake (vse pojavitve i-tega znaka prvega niza zamenja z i-tim znakom drugega niza).

Program `tac` prebere vsebino dane datoteke (oz. standardnega vhoda, če mu ne podamo imena datoteke) in si misli, da je sestavljena iz „zapisov“, ki se končajo z določenim ločilnim nizom. Te zapise potem izpiše na standardni izhod v obratnem vrstnem redu. Privzeta vrednost ločilnega niza je `"\n"`, torej `tac` takrat preprosto obrne vrstni red vrstic v vhodnem besedilu.

Naša gornja rešitev torej najprej zamenja presledke z znaki za konec vrstice, tako da vsaka beseda pride na samostojno vrstico; nato s `tac` zamenja vrstni red vrstic; na koncu pa znake za konec vrstice spremeni v presledke, tako da iz vsega skupaj spet nastane ena sama vrstice, med besedami pa so presledki. V primeru, če ima vhodno besedilo več vrstic, ta rešitev seveda ne deluje, saj bi besede z vseh vrstic staknil skupaj v eno samo dolgo vrstico. Malo nerodno je tudi to, da na koncu svojega izpisa ta rešitev ne doda znaka za konec vrstice, pač pa le še en presledek (saj je zadnji klic `tr` vse konce vrstic spremenil v presledke).

Naloga: str. 2

R2000.U.2

Datoteke, ki nas zanimajo, lahko poiščemo s programom `find`. Naročili mu bomo, naj išče vse od korenskega imenika (`/`) navzdol, da nas zanimajo le navadne datoteke (`-type f`) in da mora v imenu biti besedica `core` (`-name "*core*"`). Program `find` izpiše poti do najdenih datotek na standardni izhod, po eno datoteko v vsaki vrstici. To beremo v zanki z lupininim ukazom read; v vsaki iteraciji te zanke preberemo po eno vrstico in jo shranimo v spremenljivko `$f`. Ker ima marsikakšna datoteka v imenu besedo `core`, pa vendarle ne gre za core dump, moramo pred brisanjem še preveriti, če je datoteka res tega tipa. To lahko naredimo s programom `file`, ki izpiše na standardni vhod vrstico oblike „*ime datoteke*`:` *opis*“. Pobrisali bomo le tiste, pri katerih tudi *opis* vsebuje besedo `core`. To preverimo s programom `grep` in primernim regularnim izrazom; stikalo `-q` pa mu naroči, naj ničesar ne izpisuje, saj potrebujemo od njega le povratno vrednost, da jo bomo lahko preverili z lupininim stavkom **if**.

```
find / -type f -name "core*" |
while read f
do
  if file $f | grep -q ^[^:]*:.*core
  then
    rm -f $f
  fi
done
```

Lahko bi uporabili tudi program `cut`, mu naročili, naj razreže vrstico, ki jo je izpisal `file`, pri dvopičjih (−`d :`) in izpiše vse od vključno drugega kosa naprej (−`f 2−`), kajti prvi kos je ime datoteke.

```
if file $f | cut −d : −f 2− | grep −q core
```

R2000.U.3

Naloga: str. 2

Naloga je zelo primerna za reševanje s programom `sed`. Ta bere standardni vhod vrstico za vrstico in izvaja ukaze, ki smo mu jih navedli. Z njimi lahko trenutno vrstico preoblikujemo, na koncu pa `sed` izpiše dobljeni niz in se loti naslednje vrstice. Uporabili bomo ukaz `s:`*vzorec*$_1$`:`*vzorec*$_2$`:`*zastavice*, s katerim zamenjamo pojavitve vzorca 1 z vzorcem 2. Z zastavico `g` zahtevamo, naj se zamenjajo vse pojavitve, ne pa samo prva. Mi bi radi prazne elemente pobrisali, torej bo vzorec 2 kar prazen niz, vzorec 1 pa se mora ujemati z nizi oblike `<ime></ime>`. To, da bo `ime` obakrat enako, lahko zagotovimo tako, da njegovo prvo pojavitev v regularnem izrazu obdamo z oklepaji `\(...\)`, nato pa se nanjo sklicujemo z `\1`. Niz `\1` v regularnem izrazu namreč zahteva, naj se na tem mestu pojavi natančno isti podniz, kakršen se je že ujel s tistim delom regularnega izraza, ki je znotraj prvega para oklepajev `\(...\)`.

```
sed "s:<\([a-zA-Z]*\)></\1>::g"
```

R2000.U.4

Naloga: str. 3

Do vseh permutacij n besed lahko pridemo z naslednjim razmislekom. Naj bo w prva izmed teh besed; vse permutacije naših n besed lahko razdelimo v n skupin glede na to, katero mesto ima v permutaciji beseda w. Če vzamemo eno od teh skupin in zbrišemo besedo w iz vseh permutacij v njej, dobimo ravno vse permutacije preostalih $n-1$ besed, vsako natanko po enkrat. Torej lahko z rekurzivnim klicem pripravimo najprej vse permutacije $n-1$ besed, nato pa besedo w vrinemo v vsako od njih na vseh n možnih položajev (kot prvo, drugo, ..., n-to). Tako bomo dobili ravno vse permutacije vseh n besed. Robni primer, pri katerem se rekurzija ustavi, nastopi takrat, ko imamo le še eno samo besedo in je pri njej možna tudi ena sama permutacija. Pravzaprav bi lahko definirali tudi permutacijo 0 besed (kot prazen seznam) in ustavili rekurzijo šele tam.

Rešitev v perlu:

```
#!/usr/bin/perl -n
permutiraj([split], []);
sub permutiraj {
  my @elementi = @{ $_[0] };
  my @permutacije = @{ $_[1] };
  unless (@elementi) {
    print "@permutacije\n";
  } else {
    my(@noviElementi, @novePermutacije, $i);
    foreach $i (0 .. $#elementi) {
      @noviElementi = @elementi;
      @novePermutacije = @permutacije;
      unshift(@novePermutacije, splice(@noviElementi, $i, 1));
      permutiraj([@noviElementi], [@novePermutacije]);
    }
  }
}
```

Rešitev v pythonu:

```
def permut1(s):
  if s == []: return [s]
  else: return [u[:i] + [s[0]] + u[i:] for u in permut1(s[1:]) for i in range(len(s))]

import sys
for s in permut1(sys.stdin.readline().split()):
  print " ".join(s)
```

Funkcija `permut1(s)` vrne vse permutacije seznama `s`; torej, če je `s` seznam n elementov, vrne `permut1(s)` seznam $n!$ seznamov, od katerih ima vsak po n elementov.

Še nekaj pojasnil o pythonovi sintaksi in uporabljenih funkcijah. Funkcija `range(n)` vrne seznam [0, 1, 2, ..., n − 1]. Izraz `[f(x, y) for x in L1 for y in L2]` sestavi seznam, v katerem je po en element, namreč `f(x, y)`, za vsak par `(x, y)`, pri katerem je `x` iz seznama `L1` in `y` iz seznama `L2`. Izraz `u[:i]` pomeni seznam, v katerem je prvih `i` elementov seznama `u`; izraz `u[i:]` pa seznam, v katerem so vsi elementi seznama `u` razen prvih `i`. Izraz `[]` pomeni prazen seznam, izraz `[x]` pa seznam, ki ima le en sam element, namreč `x`. Sezname lahko stikamo z operatorjem +. Objekt `sys.stdin` predstavlja standardni vhod; metoda `readline` prebere naslednjo vrstico in jo vrne kot niz; metoda `x.split` pa vrne seznam nizov, ki jih dobi tako, da niz `x` razreže pri vseh presledkih. S klicem `x.join(s)` pa dobimo niz, v katerem so staknjeni skupaj vsi nizi iz seznama `s`, med njimi pa je niz `x` (v našem primeru presledek).

Slabost gornje rešitve je, da eksplicitno sestavi seznam vseh $n!$ permutacij danega seznama. To utegne biti potratno s pomnilnikom, klicatelj funkcije `permut1` pa mogoče sploh ne potrebuje seznama vseh permutacij — mogoče mu je dovolj že to, da jih dobiva eno za drugo in lahko z vsako nekaj naredi. V našem primeru je že tako, saj jih moramo le izpisovati in lahko na vsako pozabimo, čim jo izpišemo.

Zato bi bilo lepo, če bi lahko funkcijo `permut1` izvajali „po koščkih“ — vsakič le toliko, da bi nam izračunala naslednjo permutacijo; nato bi njeno izvajanje zamrznili, obdelali trenutno permutacijo, nato z izvajanjem funkcije `permut1` nadaljevali do naslednje permutacije in tako dalje. Podprogramu, ki ga lahko uporabljamo na ta način, pogosto pravimo „korutina“ (*coroutine*). Korutine podpira tudi python, le da se tam imenujejo „generatorji“.

```
def permut2(s):
  if s == []: yield s
  else:
    for u in permut2(s[1:]):
      for i in range(len(s)):
        yield u[:i] + [s[0]] + u[i:]
import sys
for s in permut2(sys.stdin.readline().split()):
  print " ".join(s)
```

Stavek **yield** namesto **return** pythonu pove, da to ni navaden podprogram, pač pa generatorska funkcija. Ko pokličemo `permut2(s)`, se ne začnejo izvajati stavki v funkciji `permut2`, pač pa je rezultat tega klica *generator* — nek objekt, ki hrani podatke o tem, do kod je že prišlo izvajanje podprograma `permut2` in kakšne

so trenutne vrednosti njegovih lokalnih spremenljivk. Generator pa ima tudi metodo next, ki ob vsakem klicu nadaljuje z izvajanjem podprograma permut2 do naslednjega stavka **yield** in vrne vrednost, ki jo je permut2 navedel v tem stavku **yield**. Zato lahko generator uporabimo v stavku **for** in bo na primer **for** u **in** permut2(...) v vsaki iteraciji zanke priredil u-ju naslednjo vrednost, ki jo vrne permut2(...) prek stavka **yield**. Pri našem programu se tako zdaj pravzaprav vedno izvaja kar n vzporednih korutin, ki prek stavka **yield** sestavijo naslednjo permutacijo, nikoli pa ne obstaja v pomnilniku hkrati seznam vseh permutacij. Poraba pomnilnika je zato le še $O(n)$, ne več $O(n!)$.

Še en način, kako priti do vseh permutacij, pa je naslednji: vse permutacije n elementov dobimo tako, da na vse možne načine izberemo enega od njih in ga postavimo na prvo mesto, nato pa ostala mesta zapolnimo z vsemi permutacijami ostalih $n-1$ elementov. To je pravzaprav enak razmislek kot prej, le zapisan na malo drugačen način; je pa dobro izhodišče za naslednjo rekurzivno rešitev.

```
def permut3(zePostavljene, ostale):
  if ostale == []:
    print " ".join(zePostavljene)
  else:
    for i in range(len(ostale)):
      permut3(zePostavljene + [ostale[i]], ostale[:i] + ostale[i + 1:])
import sys
permut3([], sys.stdin.readline().split())
```

Ta rešitev pa porabi veliko časa za rezanje in stikanje seznamov. Zato bo bolje, če imamo ves čas le en sam seznam in elemente samo premeščamo po njem:

```
def permut4(tabela, stZePostavljenih):
  if stZePostavljenih == len(tabela):
    print " ".join(tabela)
  else:
    for i in range(stZePostavljenih, len(tabela)):
      (tabela[i], tabela[stZePostavljenih]) = (tabela[stZePostavljenih], tabela[i])
      permut4(tabela, Postavljenih + 1)
      (tabela[i], tabela[stZePostavljenih]) = (tabela[stZePostavljenih], tabela[i])
import sys
permut4(sys.stdin.readline().split(), 0)
```

Tukaj torej podprogram permut4 predpostavi, da je prvih stZePostavljenih elementov tabele tabela že fiksiranih na svojih mestih, zdaj pa je treba na vse možne načine premešati preostale elemente. To naredimo tako, da z zanko izberemo vsakič po enega od preostalih, ga postavimo na indeks stZePostavljenih in ga razglasimo za fiksiranega; z rekurzivnim klicem potem pripravimo vse permutacije preostalih elementov. V pythonu lahko dve vrednosti zamenjamo s prireditvijo oblike (a, b) = (b, a), ki „hkrati“ priredi staro vrednost b-ja a-ju in staro vrednost a-ja b-ju; na ta način povlečemo enega od elementov na prvo mesto in ga po vrnitvi iz rekurzivnega klica postavimo nazaj.

Za primerjavo smo pognali na istem računalniku vse štiri predstavljene pythonovske rešitve na seznamu desetih besed (izpis pa preusmerili v datoteko). permut1 je porabil 64 s, permut2 31 s, permut3 61 s, permut4 pa 45 s. Videti je torej, da sta permut1 in permut3 počasna zaradi preveč prekladanja seznamov; razlika

v hitrosti med permut2 in permut4 pa je mogoče posledica tega, da je hitreje nadaljevati z izvajanjem generatorja kot pa začeti s povsem novim rekurzivnim klicem (ker je pri slednjem več knjigovodskega dela).

REŠITVE NALOG TRETJEGA TEKMOVANJA IZ UNIXA

Naloga: str. 3

R2001.U.1

Pomagali si bomo s programom `diff`. Ta primerja dve datoteki in na standardni izhod izpiše podatke o tem, kje (v katerih vrsticah) in kako se razlikujeta. Če sta datoteki enaki, ne izpiše ničesar. Njegov izpis lahko pošljemo programu `wc`, ki zna šteti vrstice, besede in znake v svojem standardnem vhodu; s stikalom `-c` mu povemo, naj izpiše le število znakov. Spodnja skripta za lupino `bash` lahko potem to število prebere; če je enako 0, pomeni, da `diff` ni izpisal ničesar in sta datoteki enaki, sicer pa sta različni.

```
#!/bin/bash
diff $1 $2 | wc -c | (
  read dolzina;
  if [ $dolzina -gt 0 ]
  then echo "različni"; fi
)
```

Spremenljivki `$1` in `$2` predstavljata prva dva parametra, ki ju je naš program dobil iz ukazne vrstice. Z internim lupininim ukazom `read` lahko preberemo število, ki ga je izpisal `wc`. V stavku **if** uporabimo operator `-gt`, ki gleda na operanda kot na števili in pove, če je levo večje od desnega. Klic `read` in stavek **if** morata biti skupaj v oklepajih, sicer stavek **if** ne bi videl vrednosti, ki jo je `read` vpisal v spremenljivko `dolzina`. Lahko pa bi namesto tega uporabili obrnjene narekovaje (*backquotes*), ki izvedejo ukaze med narekovaji in izhod teh ukazov shranijo v spremenljivko:

```
#!/bin/bash
dolzina=`diff $1 $2 | wc -c`
if [ $dolzina -gt 0 ]
then echo "različni"; fi
```

Naloga: str. 3

R2001.U.2

Spodnji program v pythonu bere vhodno datoteko po vrsticah; če naleti na prazno vrstico, neha; če pa naleti na vrstico `Subject:`, preveri, če se za tem nizom (in morebitnimi presledki) pojavi besedilo „`I LOVE YOU`“.

```
import sys
for s in file(sys.argv[1], "rt"):
  if s == "\n": break
  if s.startswith("Subject:") and s[8:].strip().startswith("I LOVE YOU"):
    print 1; break
```

Naloga: str. 4

R2001.U.3

V okoljski spremenljivki `$PATH` so našteti imeniki, ločeni z dvopičji; s pythonovo funkcijo `split` lahko razbijemo ta niz v seznam imen posameznih imenikov. Potem se lotimo vsakega imenika posebej; uporabimo funkcijo `realpath`, ki zamenja imena simbolnih povezav s tistim, na

kar te povezave kažejo. Za lažje preverjanje, če smo si nek imenik že ogledali, bomo imena že obdelanih imenikov hranili v razpršeni tabeli pregledaniImeniki. Pri vsakem imeniku potem pregledamo vse datoteke v njej (funkcija os.listdir vrne seznam imen datotek); z realpath spet poskrbimo za simbolne povezave. Potem moramo le še preveriti, če je tista stvar v resnici navadna datoteka (isfile) in če je z našega stališča izvršljiva. Pomagali si bomo s funkcijo os.stat, ki vrne strukturo s koristnimi podatki o datoteki. V polju st_mode so zastavice, ki povedo, kdo lahko datoteko bere, piše in izvaja (prav iste, kot jih lahko spreminjamo s programom `chmod`); v poljih st_uid in st_gid pa sta uporabniška številka lastnika datoteke ter številka skupine, ki ji lastnik pripada. To dvoje lahko primerjamo s svojo številko in številko skupine (getuid, getgid) in tako vidimo, katere zastavice v st_mode veljajo za nas. Da ne bi iste datoteke šteli po večkrat, hranimo imena že odkritih datotek v razpršeni tabeli datoteke.

```
import os, os.path, stat

imeniki = os.environ["PATH"]
pregledaniImeniki = {}    # da ne bi po večkrat pregledovali celih imenikov
datoteke = {}             # množica vseh že odkritih izvršljivih datotek
# Naša uporabniška številka in skupina — to bomo uporabljali
# za preverjanje, če bi lahko neko datoteko izvedli.
uid = os.getuid(); gid = os.getgid()

# Preglejmo vse imenike.
for s in imeniki.split(':'):
  imenik = os.path.realpath(s)              # prava pot do tega imenika
  if imenik in pregledaniImeniki: continue  # tega smo že pregledali
  pregledaniImeniki[imenik] = 1
  if not os.path.isdir(imenik): continue    # to sploh ni imenik

  # Preglejmo vse datoteke v tem imeniku.
  for ime in os.listdir(imenik):
    polnoIme = os.path.realpath(os.path.join(imenik, ime))
    if not os.path.isfile(polnoIme): continue  # najbrž je podimenik
    st = os.stat(polnoIme)
    if polnoIme in datoteke: continue          # to datoteko smo že videli
    # Preverimo zdaj, če smemo to datoteko izvajati.
    izvrsljiva = False
    if st.st_mode & stat.S_IXUSR and st.st_uid == uid: izvrsljiva = True
    if st.st_mode & stat.S_IXGRP and st.st_gid == gid: izvrsljiva = True
    if st.st_mode & stat.S_IXOTH: izvrsljiva = True
    if izvrsljiva: datoteke[polnoIme] = 1

print len(datoteke)
```

V primeru, če kaže na isto datoteko več trdih povezav (ne pa simbolnih), bi gornji program štel vsako povezavo posebej, saj bi realpath pustil imena takih povezav pri miru. Če bi se hoteli izogniti tudi takemu podvajanju, bi lahko v tabeli datoteke namesto imen hranili pare (st.st_dev, st.st_ino), ki enolično identificirajo posamezno datoteko. Pri tem je st_ino številka datoteke znotraj datotečnega sistema (*inode number*), st.st_dev pa pove, v katerem datotečnem sistemu se ta datoteka nahaja.

R2001.U.4

Recimo, da imamo dve majhni števili (x_1, x_2) in dve veliki števili (X_1, X_2). Je bolje vzeti zmnožek obeh majhnih in

Naloga: str. 4

zmnožek obeh velikih ali dva mešana zmnožka s po enim majhnim in enim velikim? Recimo, da je $x_1 = x_2 = x$ in $X_1 = X_2 = kx$; potem nam da prva možnost vsoto $x_1x_2 + X_1X_2 = (k^2+1)x^2$, druga pa $x_1X_1 + x_2X_2 = 2kx^2$. Ker je (če k ni premajhen) k^2+1 precej večje od $2k$, nam bo dala manjši rezultat druga možnost.

Opažanje iz tega primera lahko posplošimo: če hočemo čim manjšo vsoto zmnožkov, je bolje množiti velika števila z majhnimi kot pa posebej velika med sabo in majhna med sabo. Tega načela se bomo najdosledneje držali, če števila kar uredimo naraščajoče in nato zmnožimo najmanjše in največje, pa drugo najmanjše in drugo največje in tako naprej.

Prepričajmo se, da s tem res dobimo najmanjšo vsoto zmnožkov. Označimo naša števila v naraščajočem vrstnem redu z $a_1, \ldots, a_{2n}$, torej tako, da je $a_1 \le a_2 \le \cdots \le a_{2n}$. Naš postopek bi vzel zmnožke

$$a_1a_{2n} + a_2a_{2n-1} + \ldots + a_ia_{2n-i+1} + \ldots + a_na_{n+1}.$$

Recimo pa, da je mogoče z neko drugo razdelitvijo teh števil na pare dobiti manjšo vsoto zmnožkov. Ta razdelitev se z našo mogoče v prvih nekaj parih ujema, prej ali slej pa se mora od nje razlikovati; recimo, da so pri tej drugi razdelitvi tudi prisotni pari $a_1a_{2n}, \ldots, a_{i-1}a_{2n-i+2}$, število a_i pa ni v paru z a_{2n-i+1} (kot pri naši razporeditvi), pač pa z nekim a_j. Ker smo števila $a_1, \ldots, a_{i-1}$ in $a_{2n-i+2}, \ldots, a_{2n}$ že porabili, poleg tega pa tudi ne more biti $j = 2n-i+1$ (saj bi potem to ne bilo nič drugače kot pri naši razporeditvi), mora biti $i < j < 2n-i+1$, poleg tega pa mora biti število a_{2n-i+1} pri tej drugi razporeditvi v paru z nekim a_k, ne pa z a_i kot pri naši; in za k mora iz enakih razlogov kot za j veljati $i < k < 2n-i+1$. V opazovani razporeditvi torej nastopata para a_ia_j in $a_{2n-i+1}a_k$; pa recimo zdaj, da bi elementa a_j in a_k zamenjali. S tem bi vsota zmnožkov izgubila člena a_ia_j in $a_{2n-i+1}a_k$, pridobila pa bi a_ia_{2n-i+1} in a_ja_k. Zato se poveča za

$$a_ia_{2n-i+1} + a_ja_k - a_ia_j - a_{2n-i+1}a_k = (a_{2n-i+1} - a_j)(a_i - a_k).$$

Zaradi $j < 2n-1+1$ je $a_j \le a_{2n-1+1}$, tako da je prvi faktor v tem izrazu nenegativen; zaradi $i < k$ pa je $a_i < a_k$, tako da je drugi faktor nepozitiven; celotna sprememba vsote je torej nepozitivna. Z drugimi besedami, tisto domnevno boljšo razporeditev, ki se je z našo ujemala le v prvih $i-1$ parih, v i-tem paru pa ne, se je dalo predelati tako, da se ujema z našo tudi v i-tem paru, pri tem pa se ji ni vsota zmnožkov nič povečala, ampak je ostala ali enaka ali pa se je celo zmanjšala! S takšnim razmislekom bi lahko nadaljevali in korak za korakom spreminjali tisto razporeditev tako, da bi na koncu postala enaka naši; in ker se ji ni vsota zmnožkov pri tem nikoli povečala, pomeni, da ni naša razporeditev nič slabša od tiste prvotne. Torej je naša razporeditev res najboljša možna.

Zapišimo še program v pythonu:

```
stevila = [int(vrstica) for vrstica in file("stevila.txt")]
stevila.sort()
f = file("pari.txt", "wt")
for i in range(len(stevila) // 2):
  f.write("%d %d\n" % (stevila[i], stevila[-i - 1]))
```

REŠITVE NALOG ČETRTEGA TEKMOVANJA IZ UNIXA

R2002.U.1

Naloga: str. 4

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.

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)))
```

R2002.U.2

Naloga: str. 5

Primer rešitve z bashem 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. 5

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. 5). 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. 6

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
```

REŠITVE NALOG PETEGA TEKMOVANJA IZ UNIXA

R2003.U.1

Naloga: str. 6

Pri tej nalogi nam bo prišel prav program `egrep`, ki prebere neko datoteko in izpiše le tiste njene vrstice, ki ustrezajo določenemu regularnemu izrazu. Pognali ga bomo dvakrat, najprej zato, da bo obdržal vrstice, ki ustrezajo prvemu izrazu, nato pa bomo te vrstice še enkrat pognali skozi `egrep` in obdržali le tiste izmed njih, ki *ne* ustrezajo drugemu izrazu (stikalo `-v`). Programu `egrep` lahko s stikalom `-f` povemo, naj regularni izraz prebere iz določene datoteke.

Naša skripta za `bash` lahko do parametrov, ki jih je dobila iz ukazne vrstice, dostopa prek spremenljivk `$1`, `$2` in `$3`. To, če nek niz res predstavlja ime neke datoteke, lahko preverimo z operatorjem `-f`; vse tri pogoje združimo z operatorjem `-a`, ki pomeni logični in.

```
#!/bin/bash
if [ -f "$1" -a -f "$2" -a -f "$3" ]; then
    egrep -f "$2" "$1" | egrep -v -f "$3"
else
    echo "NAPAKA"
fi
```

R2003.U.2

Naloga: str. 6

Spodnja rešitev (v perlu) prebere kar celo datoteko v pomnilnik (njeno ime je prvi parameter iz ukazne vrstice in do njega pridemo z `$ARGV[0]`) in potem z regularnim izrazom preveri, če je v njej prisotna glava. Operator `s/`*vzorec*$_1$`/`*vzorec*$_2$`/`*zastavice* zamenja pojavitev

vzorca 1 z vzorcem 2; v našem primeru pokrije vzorec 1 celo glavo, vzorec 2 pa je prazen in tako se glave znebimo. Zastavica `s` na koncu pa zahteva, naj obravnava interpreter cel niz kot eno samo dolgo vrstico; to potrebujemo, saj bi se utegnili znotraj glave pojavljati tudi znaki za konec vrstice. Pozorni moramo biti tudi na naslednje: glava se konča že pri prvi pojavitvi niza `data`; znak `*` v regularnem izrazu pa načeloma poskuša pokriti čim več besedila („požrešno ujemanje“, *greedy matching*), torej bi šel tu do zadnje pojavitve niza `data` v opazovanem nizu. Če hočemo, da pokrije čim manj besedila (torej le do prve pojavitve niza `data`), moramo za zvezdico postaviti še `?`.

```
#!/usr/bin/perl
use strict;
use warnings;

open FH, $ARGV[0];
$_= join('', (<FH>));          # Preberemo celo datoteko.
close FH;

if (s/^RIFF.*?data//s) {       # Zbrišimo glavo, če je prisotna.
  open FH, '>', $ARGV[0];      # Če je bila glava prisotna, shranimo
  print FH;                    # preostanek podatkov nazaj v datoteko.
  close FH;
}
```

Naloga pravi, da če se datoteka ne začne na `RIFF`, naj program ne reže ničesar; ne pove pa, kaj storiti v primeru, če se začne na `RIFF`, vendar kasneje ne vsebuje niza `data`. Gornji program bi jo pustil pri miru; verjetno je to vendarle bolje, kot pa če bi pobrisali celo datoteko.

Morebitna slabost gornje rešitve je, da prebere celo datoteko v pomnilnik. To utegne biti nerodno, če je datoteka velika (kar ni pri multimedijskih datotekah nič neobičajnega). Za take primere bi bilo bolje, če bi datoteko brali po koščkih in vsebino, ki sledi nizu `data`, sproti prepisovali na začetek datoteke (podobno kot v rešitvi naloge 2003.U.4), na koncu pa bi datoteko skrajšali s funkcijo truncate.

Naloga: str. 6

R2003.U.3

Za vsak proces obstaja navidezni imenik `/proc/`*pid*, pri čemer je *pid* številka procesa. V tem imeniku je med drugim datoteka z imenom `exe`, ki je simbolna povezava na izvršilno datoteko tistega procesa. V lupini `bash` lahko prek spremenljivke `$PPID` dobimo številko procesa-očeta. Ime prave izvršilne datoteke, kamor kaže naša simbolna povezava, lahko izvemo od programa `ls`, če zahtevamo izčrpnejši izpis (stikalo `-l`). Vrstico, ki jo `ls` izpiše, razbijmo pri vseh presledkih (`cut -d ' '`); izkaže se, da je ime datoteke, kamor kaže simbolna povezava, potem ravno enajsta komponenta vrstice. Ker pa lahko ime vsebuje tudi presledke, je bolje izpisati vse komponente od enajste naprej (stikalo `-f 11-`). Težava je le ta, da `cut` prereže pri vsakem presledku, v izpisu programa `ls` pa je včasih po več presledkov skupaj in bi zato `cut` tam vmes ustvaril še neko število praznih komponent; to število je nepredvidljivo, ker ne vemo, koliko presledkov je `ls` vrinil zaradi poravnavanja stolpcev pri izpisu (odvisno je npr. od tega, koliko števk je porabil za izpis dolžine datoteke). Dobro bi bilo torej spremeniti vsako zaporedje presledkov v en sam presledek. To lahko naredimo s programom `tr`, če uporabimo stikalo `-s`. Druga možnost je, da si izpis programa `ls` shranimo v neko spremenljivko in

jo podamo programu `echo`: iz posameznih komponent bodo nastali posamezni argumenti programu `echo`, interpreterju lupine pa je čisto vseeno, s koliko presledki so ločeni argumenti; `echo` bo med dvema argumentoma vedno izpisal en presledek.

```
#!/bin/bash
povezava=`ls -l /proc/$PPID/exe`
echo $povezava | cut -d ' ' -f 11-
```

ali pa

```
#!/bin/bash
ls -l /proc/$PPID/exe | tr -s ' ' | cut -d ' ' -f 11-
```

Gornja rešitev ima še majhno slabost: če se v imenu datoteke, na katero kaže opazovana simbolna povezava, kdaj pojavlja po več zaporednih presledkov, bo naš program tam izpisal en sam presledek, ker pač v `ls`-jevem izpisu nadomesti vsa zaporedja presledkov s po enim samim. Na srečo pa se v imenih datotek le redko pojavi več zaporednih presledkov.

Lahko bi si pomagali z dejstvom, da v izpisu programa `ls` pred imenom datoteke, na katero kaže simbolna povezava, stoji niz `"-> "`. S `sed`om lahko pobrišemo vse do vključno te puščice in presledka (stikalo `-n` je zato, da ne bo `sed` izpisal še prvotnega niza, kakršen je bil pred brisanjem):

```
#!/bin/bash
ls -l /proc/$PPID/exe | sed -n "s/^.*-> //p"
```

Slabost te rešitve je, da `sed` ujemanje z regularnimi izrazi preverja „požrešno“ (*greedy matching*) — zvezdica poskuša pobrati čim daljši kos niza. Če bi se torej v imenu očetovskega procesa pojavil niz `"-> "`, bi `sed` pobrisal še del tega imena, vse do zadnje pojavitve niza `"-> "`.

Še ena možnost je, da namesto `sed`a uporabimo `awk`; na začetku nastavimo njegovo spremenljivko FS in mu s tem naročimo, naj vrstico, ki jo je izpisal `ls`, razreže pri vseh puščicah. Vrstica tako razpade na NF delov (i-tega dobimo v spremenljivki $i), mi pa moramo izpisati vse razen prvega.

```
#!/bin/bash
ls -l /proc/$PPID/exe | awk '
  BEGIN { FS = "-> " }
  {
    for (i = 2; i < NF; i++)
      printf "%s-> ", $i;
    print $NF;
  }'
```

Še lažje in bolj elegantno gre na primer v perlu, saj imamo funkcijo readlink, ki nam pove, kam kaže simbolna povezava. Očetovo številko dobimo s funkcijo getppid, nize pa stikamo z operatorjem . (pika).

```
#!/usr/bin/perl
print readlink("/proc/" . getppid . "/exe") . "\n";
```

Naloga: str. 7

R2003.U.4

Za stikanje datotek lahko uporabimo program `cat`. S programom `echo` mu pošljemo niz „`PREPIS`“ (brez znaka za konec vrstice, zato stikalo `-n`) in nato programu `cat` naročimo, naj stakne to, kar je prišlo s standardnega vhoda (−), z vsebino vhodne datoteke. Rezultat bi lahko zapisovali kar v izhodno datoteko, vendar pa bi to v primerih, ko sta vhodna in izhodna datoteka ena in ista, pomenilo, da bomo vhodne podatke najbrž izgubili, še preden bomo vse sploh prebrali. Zato raje uporabimo pomožno datoteko in jo potem preimenujmo v ime, ki smo ga dobili kot ime izhodne datoteke. Ime pomožne datoteke pripravimo s programom `mktemp`, ki znake `X` na koncu danega argumenta zamenja z naključnimi števkami in pazi na to, da datoteka s takšnim imenom še ne obstaja; dobljeno ime potem izpiše na svoj standardni izhod.

```
#!/bin/bash
pomozna=`mktemp "$2.XXXXXX"`
echo -n PREPIS | cat - "$1" > "$pomozna"
mv "$pomozna" "$2"
```

Slabost te rešitve je, da je včasih lahko potratna s prostorom. Če sta vhodna in izhodna datoteka različni, bi lahko pisali kar naravnost v izhodno datoteko, tako pa so tik pred klicem `mv` prisotne na disku vse tri: vhodna, pomožna (ki je dolga približno toliko kot vhodna, pravzaprav šest znakov daljša) in še stara izhodna. V primerih, ko sta vhodna in izhodna datoteka ena in ista, pa uporaba pomožne datoteke pomeni, da bosta pred klicem `mv` prisotni na disku dve kopiji vhodne (prvotna in tista pomožna z nizom „`PREPIS`“).

Varčnejša rešitev bi najprej preverila, če sta vhodna in izhodna datoteka ena in ista; če je res tako, naj odpre to datoteko za branje in pisanje obenem, nato pa prebira iz nje podatke po koščkih in se po vsakem branju pomakne po datoteki nazaj ter povozi ravnokar prebrane podatke s tistim, kar bo moralo biti na tem mestu zapisano v izhodni datoteki. Ker je vsebina izhodne datoteke v primerjavi z vsebino vhodne „zamaknjena“ za šest znakov (ker je v izhodni na začetku še niz „`PREPIS`“), nam vedno ostane šest znakov, ki smo jih že prebrali iz vhodne datoteke ter jih pri zadnjem pisanju tudi že povozili z drugimi podatki; te obdrži spodnji program v nizu `buf` in bodo prišli kot prvi na vrsto pri naslednjem pisanju v datoteko (takoj za naslednjim branjem).

```
import sys, os, os.path

def IstaDatoteka(ime1, ime2):
  if ime1 == ime2: return True
  # Mogoče pa sta to trdi povezavi na isto datoteko.
  st1 = os.stat(ime1); st2 = os.stat(ime2)
  return st1.st_dev == st2.st_dev and st1.st_ino == st2.st_ino

vhodIme = os.path.realpath(sys.argv[1])
pazi = False
if len(sys.argv) <= 2:
  # Izpisovali bomo na standardni izhod.
  izhod = sys.stdout
else:
  # Preverimo, če je izhodna datoteka ista kot vhodna.
  izhodIme = os.path.realpath(sys.argv[2])
```

```
  pazi = IstaDatoteka(vhodIme, izhodIme)
  if pazi:
    # Je — odprimo jo le enkrat, za branje in pisanje.
    # „vhod“ in „izhod“ bosta le dve referenci na isti objekt.
    vhod = file(vhodIme, "r+b"); izhod = vhod
  else:
    # Izhodna datoteka ni ista kot vhodna; odprimo izhodno za pisanje
    # (in uničimo morebitno obstoječo datoteko s tem imenom).
    izhod = file(izhodIme, "wb")
if not pazi:
  # Če sta vhodna in izhodna datoteka različni,
  # odprimo zdaj še vhodno, vendar le za branje.
  vhod = file(vhodIme, "rb")

buf = "PREPIS"
bufLen = 1024 * 1024
while len(buf) > 0:
  # Zapomimo si trenutni položaj v datoteki.
  if pazi: pos = izhod.tell()
  # Preberimo nekaj novih podatkov.
  buf = buf + vhod.read(bufLen)
  # Če je vhodna datoteka ista kot izhodna, se postavimo nazaj na položaj
  # „pos“, da bomo pri pisanju povozili pravkar prebrane podatke.
  if pazi: izhod.seek(pos)
  # Zapišimo nekaj podatkov.
  izhod.write(buf[:bufLen])
  # Če mešamo branja in pisanja nad isto datoteko, lahko pride včasih do težav,
  # npr. da vhod.read() vrne tisto, kar smo ravnokar zapisali na stari položaj,
  # namesto da bi prebral nove podatke. Rešitev je, da med pisanjem in branjem
  # pokličemo izhod.flush() ali pa vhod.seek(vhod.tell()).
  if pazi: vhod.seek(vhod.tell())
  # Obdržimo podatke, ki jih še nismo zapisali.
  buf = buf[bufLen:]
```

Za ugotavljanje, če se dani imeni nanašata na eno in isto datoteko, smo uporabili najprej funkcijo realpath, ki sledi simbolnim povezavam; nato pa, če sta imeni tudi po tem različni, pogledamo za vsako ime par (st.st_dev, st.st_ino), ki enolično identificira posamezno datoteko (glej rešitev naloge 2001.U.3). Tako odkrijemo še primere, ko dobimo dve „trdi povezavi“ na isto datoteko.

REŠITVE NALOG ŠESTEGA TEKMOVANJA IZ UNIXA

R2004.U.1

Primer rešitve v pythonu:

Naloga: str. 8

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

```
for s in seznam: print s          # in jih izpišimo.
```

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

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

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

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

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

Naloga: str. 8

R2004.U.2

Primer rešitve v pythonu:

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

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

Če so v vhodni datoteki sami različni IPji, bo naša razpršena tabela na koncu hranila praktično že celotno vsebino vhodne datoteke. Če je vhodna

datoteka zelo dolga, nas torej lahko skrbi, da bo naš program porabil preveč pomnilnika. Podobno kot pri prejšnji nalogi lahko vhodno datoteko tudi tu najprej uredimo; tako pridejo vrstice, ki se nanašajo na isti IP, skupaj in so tudi urejene po naraščajočem času dostopa. Zdaj je za prepoznavanje novih sej dovolj, če primerjamo po dve zaporedni vrstici.

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

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

```
sort -k 1,1 -k 2,2n $1 | ...
```

R2004.U.3

Naloga: str. 8

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

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

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

Naloga: str. 9

R2004.U.4

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

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

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

```
#!/bin/bash

# Določimo velikost vhodne in izhodne slike.
Sirina=`identify "$1" | awk '{print $3}' | sed 's/x/ /g' | awk '{print $1}'`
Visina=`identify "$1" | awk '{print $3}' | sed 's/x/ /g' | awk '{print $2}'`
NovaSirina=$3
let NovaVisina=$Visina*$NovaSirina/$Sirina

# Začnimo z največjo možno kakovostjo.
Q=100
convert $1 -resize "$NovaSirina"x"$NovaVisina" -quality $Q $2

# Zmanjšujmo kakovost v korakih po 10, dokler ne dobimo dovolj majhne slike.
while [ $Q -gt 0  -a  `ls -l $2 | awk '{print $5}'` -gt $4 ]; do
  let Q=$Q-10
  convert $1 -resize "$NovaSirina"x"$NovaVisina" -quality $Q $2
done

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

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