24. državno tekmovanje v znanju računalništva (2000)

NALOGE ZA PRVO SKUPINO

2000.1.1

Starejši si, kot misliš! Koliko sekund je minilo med začetkom (0 h) dneva, v katerem si se rodil, in začetkom današnjega dneva? Na prvi pogled se zdi, da jih je $24 \cdot 60 \cdot 60 \cdot$ *število dni* med obema datumoma. To je skoraj res, ušteli smo se le za kakšen ducat sekund. R: 11

Ozadje naloge: Sekunda (enota za čas) je bila včasih definirana s pomočjo trajanja dneva.Vedno natančnejša astronomska opazovanja so pokazala, da Zemljino vrtenje ni enakomerno, saj nanj vplivajo različne sile, na primer plimovanje, in tudi, da se Zemlja suče vse počasneje; dnevi torej niso vsi enako dolgi. Tako „nenatančna“ definicija sekunde ni več zadoščala znanosti in tehniki, zato so leta 1967 definirali sekundo s pomočjo nihanja cezijevega atoma, ki je mnogo enakomernejše od vrtenja Zemlje. Času, ki ga dobimo s štetjem tako definiranih sekund, pravimo mednarodni atomski čas (s francosko kratico TAI).

Tudi čas UTC, ki ga kažejo naše običajne ure, teče enako hitro kot TAI; vsi uporabljamo isto (novo, atomsko) definicijo trajanja sekunde. Ker pa smo vajeni, da je sonce v najvišji legi točno ob dvanajstih (zanemarimo časovne cone, denimo, da smo v Londonu), se vsako leto sproti izmeri trenutna hitrost in zaostajanje vrtenja Zemlje in po potrebi (približno enkrat letno, a ne vsako leto) vrine v UTC ena dodatna (prestopna) sekunda. Tako je imela na primer zadnja minuta v decembru 1998 po dogovoru eno sekundo več kot ostale minute, torej 61 sekund. Z drugimi besedami: 1. januarja 1999 ob 0 h smo naše ure premaknili za eno sekundo nazaj.

Naloga: Tabela datumov, ko so bile vrinjene prestopne sekunde, je na voljo v datoteki. V vsaki vrstici sta dva podatka: datum (leto-mesec-dan) tik po tem, ko je bila ob polnoči vrinjena prestopna sekunda, poleg njega pa je skupno število prestopnih sekund (torej razlika TAI − UTC), vrinjenih do tega datuma, in ki velja do nadaljnjega (do naslednjega datuma):

```
1972-07-01 11
1973-01-01 12
1974-01-01 13
1975-01-01 14
1976-01-01 15
1977-01-01 16
1978-01-01 17
1979-01-01 18
1980-01-01 19
```

```
1981-07-01 20
1982-07-01 21
1983-07-01 22
1985-07-01 23
1988-01-01 24
1990-01-01 25
1991-01-01 26
1992-07-01 27
1993-07-01 28
1994-07-01 29
1996-01-01 30
1997-07-01 31
1999-01-01 32
```

Napiši program, ki bo prebral rojstni datum (kot niz 10 znakov), potem današnji datum, in s pomočjo tabele v datoteki prestopnih sekund izračunal in izpisal število sekund, ki je minilo med obema datumoma.

Da se ti ne bo treba ukvarjati s poznavanjem koledarja in preračunavanjem datumov v nizih, si lahko pomagaš s podprogramsko funkcijo MJD, ki iz datuma, podanega kot parameter (10-znakovni niz), izračuna zaporedno številko tega dneva od nekega dogovorjenega fiksnega začetka štetja. Tako se poenostavi tudi primerjanje datumov.

```
type DatumT = packed array [1..10] of char;
function MJD(Datum: DatumT): integer; external;
```

R: 12

2000.1.2

Operacijski sistem skrbi za dodeljevanje pomnilnika procesom, ki tečejo na njem. Zaželeno je, da ima ta postopek takšne lastnosti, da je dodeljevanje in sproščanje pomnilnika hitro, njegova razdrobljenost majhna, knjigovodski podatki sistema o kosih pomnilnika pa hitro dosegljivi in neobsežni.

Nekaterim takšnim lastnostnim ustreza metoda dodeljevanja, pri kateri sistem daje pomnilnik na voljo v kosih, velikih po 2^n bajtov: 1 bajt, 2 bajta, 4 bajte, 8 bajtov... **Napiši funkcijski podprogram**, ki bo za dano število m zaprošenih bajtov vrnil velikost dodeljenega pomnilnika kot število, zaokroženo na prvo potenco števila 2, ki ni manjša od m. Pri $m = 0$ pa naj vrne kar 0, saj tisti, ki je zaprosil za 0 bajtov, pomnilnika očitno v resnici sploh ne potrebuje. Pojasni prednosti in slabosti svojega funkcijskega podprograma.

Tabela za nekaj m prikazuje vrednost te funkcije:

m	0	1	2	3	4	5	6	7	8	9	10	...
$f(m)$	0	1	2	4	4	8	8	8	8	16	16	...

R: 18

2000.1.3

Napiši program, ki prebere ocene za 10 predmetov in izpiše, ali je uspeh negativen ali pozitiven; če je uspeh pozitiven, naj izpiše tudi povprečno oceno. Za vsak predmet je podana ena ocena,

ki je celo število med 1 in 5. Uspeh je negativen, če je vsaj en predmet ocenjen z 1, sicer pa pozitiven.

2000.1.4

R: 19

Janez ima na svojem računalniku podatkovno zbirko s celotnim seznamom svojih CDjev. Za vsak CD ima podatke o izvajalcu, naslov CDja in seznam vseh skladb na CDju. Zdaj želi svoj seznam, ki je že urejen po izvajalcu in imenu albuma, izpisati na lepši način, tako da se pri izpisu ne bodo po nepotrebnem ponavljala imena izvajalcev in naslovi albumov.

Na primer, namesto:

```
Blesavi bend, 3 lahki komadi, Moja prva ljubezen
Blesavi bend, 3 lahki komadi, Moja druga ljubezen
Blesavi bend, 3 lahki komadi, Moja zadnja ljubezen
Blesavi bend, Singl, Tristo kosmatih
Nori fantje, Najvecji neuspehi, Spet si sla k drugemu
```

naj **program** izpiše:

```
Blesavi bend  3 lahki komadi     Moja prva ljubezen
                                 Moja druga ljubezen
                                 Moja zadnja ljubezen
              Singl              Tristo kosmatih
Nori fantje   Najvecji neuspehi  Spet si sla k drugemu
```

Pri tem naj program pazi, da je med imenom izvajalca in imenom albuma ter med imenom albuma in naslovom skladbe vedno vsaj za dva presledka prostora, stolpci pa naj bodo vsi levo poravnani. Vrstica je lahko poljubno dolga, naj pa ne bo daljša, kot je potrebno.

Na voljo imaš dva podprograma:

ZacniSeznam povzroči, da se seznam podatkov bere od začetka.

Komad(**var** Izvajalec, Album, Skladba: string): boolean vrne false, če ni več podatkov, sicer vrne true, v parametrih pa vrne polne podatke o naslednji skladbi v vrstnem redu.

NALOGE ZA DRUGO SKUPINO

2000.2.1

R: 20

Nekatera opravila na računalniku so take narave, da jih je treba ponavljati ob določenih časih. Tako na primer lahko želimo, da vsako uro ob polni uri zapiska zvonček; da se vsako minuto požene nek program za merjenje obremenjenosti računalnika; da se varnostno arhiviranje datotek požene vsak dan natanko dvakrat: točno opoldne in

ponoči ob 15 minut čez drugo uro po lokalnem času; Omejimo se le na opravila, ki niso vezana na datum in za katera zadošča točnost ene minute.

Za opis časov, ob katerih se mora zgoditi neko opravilo, nam tako zadoščata dve polji: podatek za uro (0..23) in podatek za minuto (0..59). Za periodična opravila (*vsako* minuto oziroma *vsako* uro) se dogovorimo, da vrednost -1 pomeni „vsako“.

Tako lahko za primer zapišemo prej našteta opravila:

ura	minuta	opravilo
-1	0	zvonček
-1	-1	vsakominutna meritev
12	0	backup
2	15	backup

Glede na to, da je ura opravila podana kot lokalni čas, moramo biti previdni dvakrat na leto: ko se spomladi lokalni čas prestavi za eno uro naprej na poletni čas, in jeseni, ko skočimo za eno uro nazaj ponovno na normalni čas. Premik je vedno izveden ob polni uri za polno uro in na štetje minut ne vpliva.

Da v računalniku ni prevelike zmede, ki bi jo povzročile nenadne prestavitve ure, njegova notranja ura teče enakomerno in meri standardni čas (UTC ali po starem GMT). Krajevni čas v Sloveniji dobimo tako, da času UTC prištejemo odmik v urah, ki je pozimi $+1$ in poleti $+2$.

Po sprejetem dogovoru se uveljavitev ali razveljavitev poletnega časa vedno izvede na določen dan ponoči ob 1 h po UTC. Spomladi ob tej uri se torej urni odmik v Sloveniji spremeni iz $+1$ na $+2$, jeseni pa nazaj na $+1$.

Tudi v takem dnevu, ki ima le 23 oziroma 25 ur, se morajo opravila izvajati po načelu „najmanjšega presenečenja“: na opravila, ki se izvajajo vsako uro, prestavitev lokalnega časa ne sme vplivati; nočno arhiviranje, za katero je predpisana določena ura, pa se mora opraviti natanko enkrat tisti dan: ko se lokalna ura prvič ta dan ujame z zahtevano uro (jeseni), ali, če te ure v tem dnevu ni (pomladi), ob času ki bi veljal, če se lokalna ura ne bi še prestavila.

Napiši podprogram:

```
procedure PolnaMinuta(UraUTC, Minuta, Odmik: integer);
```

za katerega operacijski sistem jamči, da bo pognan ob vsaki polni minuti. Kot parametre dobi standardni čas (UraUTC med 0 in 23 ter Minuta med 0 in 59) ter Odmik, to je celo število ur, ki jih je treba prišteti k UraUTC, da dobimo lokalni čas. V Sloveniji je odmik pozimi vedno $+1$ in poleti $+2$. Primer: spomladi v zaporednih minutah okrog uvedbe poletnega časa bo naš podprogram klican z naslednjimi trojicami parametrov: $(0, 58, +1)$, $(0, 59, +1)$, $(1, 0, +2)$, $(1, 1, +2)$, $(1, 2, +2)$, lokalna ura pa v teh trenutkih kaže: 1:58, 1:59, 3:00, 3:01, 3:03.

Program naj vsakokrat prebere datoteko z opravili (kje se nahaja, ni predpisano, izberi po želji; v vsaki vrstici sta po dve števili, preostanek vrstice je naziv opravila) in naj pokliče:

```
procedure Opravi(Opravilo: NizT); external;
```

z nazivom opravila za vsako tako opravilo, ki se mora v tej minuti začeti izvajati. Podprogram Opravi se vrne takoj in ne čaka, da se bo opravilo tudi zaključilo. Definiraš lahko svoje globalne spremenljivke in predpostaviš, da je njihova začetna vrednost 0.

2000.2.2

R: 21

Na ravnini so podani pari točk, ki predstavljajo leva spodnja in desna zgornja oglišča pravokotnikov s stranicami, vzporednimi s koordinatnima osema. **Napiši podprogram**, ki poišče presek vseh pravokotnikov in izpiše koordinati spodnjega levega oglišča in zgornjega desnega oglišča tako določenega pravokotnika.

2000.2.3

R: 22

Na svoj ročni računalnik (dlančnik, palmtop) želiš redno prenašati nekatere spletne vsebine (novice, kino spored, ipd.), ki so že objavljene na različnih strežnikih po Sloveniji. Ker pa imajo ročni računalniki nekatere omejitve, zaradi katerih originalne HTML strani niso povsem primerne za direktni prikaz (majhen zaslon, nekateri imajo tudi črnobel zaslon), moraš napisati program, ki bo originalne HTML strani „oskubil“ tako, da bodo primerne za pregledovanje na ročnem računalniku.

Zapis HTML uporablja posebne oznake za označevanje delov teksta in strukture, vse pa se začnejo z znakom „<“ in končajo z „>“, vmes pa je ime elementa in njegovi atributi. Primer:

```
<HTML>
  <HEAD>
    <TITLE>Primer HTML strani</TITLE>
  </HEAD>
  <BODY BGCOLOR="white">
    <H1>Naslov: primer HTML strani</H1>
    Privzet font, <FONT FACE="Helvetica" SIZE="1" COLOR="blue">
    spremenjen font</FONT>.
  </BODY>
</HTML>
```

Pri predelavi HTML strani želimo doseči dvoje: odstraniti želimo vse odvečne atribute, tako recimo želimo namesto `<BODY BGCOLOR="white">` imeti v rezultatu samo `<BODY>`. Nato pa se želimo znebiti še odvečnih (imenujmo jih kar „prepovedanih“) HTML oznak, ki nimajo vpliva na prikaz na ročnem računalniku ali pa ga celo kvarijo. V zgornjem primeru se želimo znebiti oznake `<FONT FACE=...>` in seveda tudi `</FONT>` (če bi se samo prve, bi izhodni HTML zapis vseboval napako: konec HTML elementa, ki nima svojega začetka). Če programu torej navedemo, da se želimo znebiti le oznak `FONT`, mora biti izhod programa pri zgornjem primeru naslednji:

```
<HTML>
  <HEAD>
    <TITLE>Primer HTML strani</TITLE>
  </HEAD>
  <BODY>
    <H1>Naslov: primer HTML strani</H1>
    Privzet font,
    spremenjen font.
  </BODY>
</HTML>
```

Vsebina med odstranjenima oznakama `FONT` je seveda ostala nespremenjena.

Na voljo imaš seznam znakovnih nizov, ki predstavljajo „prepovedane“ oznake, oziroma tiste, ki jih želimo popolnoma odstraniti. Seznam je dolg n elementov:

- v Pascalu:
  const n = 10;
  var Prepovedana: **array** [1..n] **of** string;
- v C-ju:
  #define SIZE 256
  #define N 10
  char Prepovedana[N][SIZE];

Oznake HTML so sestavljene iz alfanumeričnih znakov (črke in številke).

Napiši proceduro ali program, ki odstranjuje HTML atribute iz oznak in v celoti tiste HTML oznake, ki so navedene v seznamu Prepovedana. Bral boš s standardnega vhoda (Input v pascalu, stdin v C) in rezultat izpisal na standardni izhod (Output v pascalu, stdout v C).

R: 23

2000.2.4

Slovenska nogometna reprezentanca je bila v pretekli sezoni zelo uspešna. Dobila je celo nagrado svetovne nogometne zveze za največji napredek v sezoni. Nas pa zanima, ali je slovenska reprezentanca naredila tudi največji skok na lestvici reprezentanc. **Napiši algoritem**, ki bo ugotovil, katera reprezentanca je naredila največji skok na lestvici.

Podatki o vrstem redu reprezentanc se nahajajo v dveh datotekah. V datoteki `stanje98.txt` so reprezentance naštete v vrstnem redu, kakršen je bil ob koncu leta 1998, v datoteki `stanje99.txt` pa so reprezentance naštete v vrstnem redu s konca leta 1999. Vsaka reprezentanca je v svoji vrstici, ta pa vsebuje ime reprezentance in njeno identifikacijsko številko.

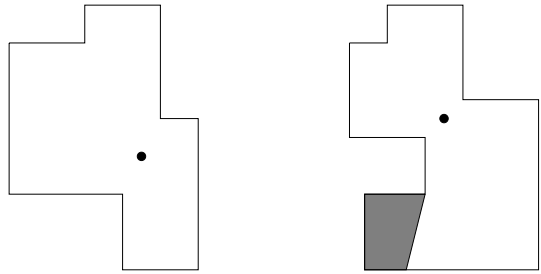
NALOGE ZA TRETJO SKUPINO

2000.3.1

R: 23

Imaš prijatelja, ki je glavni varnostnik v veliki trgovini. Ena od njegovih nalog je, da mora zagotoviti stalen video nadzor v vseh nadstropjih trgovine. Trgovina ima omejen proračun, zato želi uporabiti samo eno kamero za vsako nadstropje. Kamere lahko gledajo v vse smeri.

Prvi problem je določiti mesto, kamor bi lahko postavili kamero za posamezno nadstropje. Edina zahteva je, da mora biti s tega mesta vidno celotno nadstropje. Na spodnji sliki je levo nadstropje take oblike, da ga je možno opazovati samo z eno kamero, medtem ko v desnem primeru nikakor ne moremo postaviti kamere tako, da bi videla celoten prostor.



Preden bodo poskusili postaviti kamere, hoče tvoj prijatelj vedeti, ali je v nekem nadstropju sploh mogoče najti primerno mesto, kamor bi lahko postavili kamero. Tukaj pride na vrsto tvoja naloga. Podan imaš načrt nadstopja, ugotoviti pa moraš, ali je sploh možno postaviti kamero tako, da bi videla vsak del nadstropja.

Napiši funkcijo, ki bo ob danih vhodnih podatkih vrnila `true`, če je kamero možno postaviti tako, da bo videla celoten prostor in `false`, če tega ni možno narediti.

Stene so vedno vzporedne koordinatnima osema. Vhodni podatek je seznam točk (x, y), ki med seboj povezane v smeri urinega kazalca, določajo stene nadstropja.[1]

2000.3.2

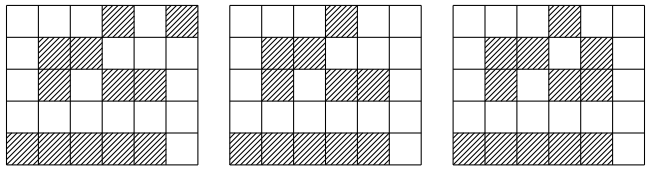
R: 25

Dan je labirint, ki temelji na pravokotni karirasti mreži (vsako polje je lahko prosto ali pa zazidano). Po labirintu se lahko premikamo, seveda le po prostih poljih; z enega gremo lahko na drugo polje le v primeru, če imata skupno eno od stranic. Sprehod po labirintu, pri katerem nobenega polja ne obiščemo po večkrat, imenujemo *pot*. Dolžino poti definiramo kot število polj, ki jih pri tej poti obiščemo, vključno z začetnim

[1] To je naloga D z ACMovega jugozahodnoevropskega študentskega tekmovanja v programiranju (SWERC 1997, Ulm, 23. nov. 1997); #588 v zbirki na `online-judge.uva.es`.

in končnim poljem. O labirintu vemo, da je zgrajen tako, da za vsak par polj obstaja natanko ena pot med tema dvema poljema.

Primer: od treh labirintov na sliki ustreza temu pogoju le skrajni levi labirint.



Radi bi ugotovili, kako dolga je najdaljša pot, ki jo lahko opravimo v tem labirintu. **Opiši** (čim hitrejši) **postopek**, s katerim bi to ugotovil. Lahko si misliš, da je labirint podan z matriko:

```
type
  Polje = (Prosto, Zid);
const
  Visina = ...;
  Sirina = ...;
var
  Labirint: array [1..Visina, 1..Sirina] of Polje;
```

Primer: za skrajni levi labirint na zgornji sliki je rešitev 15 (tako dolga je npr. pot med najbolj desnima prostima poljema v prvi vrstici).[2]

R: 30

2000.3.3

Varčni napredni kmetovalec Janez je opazil, da mu vsako leto otroci pojedo vse češnje v njegovem sadovnjaku, in se odločil, da sadovnjak ogradi z bodečo žico. Ker pa je varčen, želi postaviti natanko toliko ograje, da bo z njo obdal vsa drevesa. Janez je premeril svoj vrt in vsakemu drevesu določil koordinate njegove lege. S svojim novim računalnikom želi izračunati dolžino ograje, da bo zajela ravno vsa drevesa, a se mu je zataknilo pri postopku, s katerim želi ugotoviti, katerih dreves se bo ograja dotikala.

Janezu prišepni **postopek**, ki bo iz dvojic drevesnih koordinat poiskal tiste koordinate, ki se jih ograja dotika. Te koordinate naj postopek izpiše.

R: 33

2000.3.4

Prvo, na kar pomislimo ob besedi „Internet“, je takorekoč neomejen dostop do brezštevilne množice podatkov. Druga misel pa je: „Zakaj je moja povezava tako počasna?“ Tako za prvo kot za drugo so v veliki meri „krivi“ usmerjevalniki (*router*), preko katerih dostopamo do svetovnega medmrežja. V tej nalogi si bomo ogledali, kako (naj bi) usmerjevalniki reševali težave s počasnostjo povezav.

[2] To je naloga E z ACMovega srednjeevropskega študentskega tekmovanja v programiranju (CERC 1999, Praga, 12.–13. nov. 1999).

Zelo poenostavljeno, a smiselno za to nalogo, si lahko usmerjevalnik predstavljamo kot škatlo z n vhodnimi cevmi (imenovanimi tudi „vhodni tokovi“, *streams*) in eno izhodno cevjo. Sam promet po ceveh sestoji iz paketkov različnih dolžin, kjer dolžina paketka predstavlja *količino podatkov*. V nekem trenutku se lahko zgodi, da je količina prihajajočih podatkov večja, kot jih lahko takrat zapusti usmerjevalnik in zato jih usmerjevalnik začasno skladišči (*buffering*). Za potrebe te naloge bomo predpostavili, da usmerjevalnik *vse* prispele podatke tudi slej ko prej odpošlje.

Vrstni red, kako naj bodo paketki odposlani iz usmerjevalnika, je pomemben in je stvar usmerjevalnika — vrstni red odhajajočih podatkov določa *politiko razvrščanja* (*scheduling policy*). Idealna politika je, da tok podatkov iz vsake vhodne cevi dobi v določenem časovnem obdobju enako količino izhodne cevi — se pravi, da je količina odposlanih podatkov (*ne* število paketkov!) približno enaka za vsako vhodno neprazno cev. (V resničnih usmerjevalnikih običajno dodatno utežimo posamezno vhodno cev — tista cev, ki prihaja od vira, kateri je pripravljen več plačati, dobi večji delež izhodne cevi.) Kako učinkovita je ta politika in kako pravična je, določa *kakovost postrežbe* (*quality of service, QoS*).

Opišite in naredite (implementirajte) podatkovno strukturo s pripadajočimi operacijami, ki bo zagotavljala čim bolj pravično odpošiljanje prispelih paketkov. Vaša rešitev naj ima operacije: Pripravi(str), ki postavi strukturo str v začetno stanje, Vstavi(str, pkt, cev), ki bo poklicana, ko se bo na cevi cev pojavil paketek pkt, in Naslednji(str, pkt), ki bo poklicana, ko bo izhodna cev pripravljena za oddajo naslednjega paketka. Slednja operacija je funkcija in vrne true, če je paketek na voljo; če pa ga ni, vrne false. Paketek, ki je na voljo za pošiljanje, dobi v parametru pkt. Predpostavite lahko, da bo funkcija Naslednji samodejno poklicana, ko se bo v strukturi pojavil prvi paketek.

Predpostavite lahko tudi, da bo vedno veljalo $0 \leq \mathsf{cev} < n$.

Pri svoji rešitvi imate na voljo funkcijo Velikost(pkt), ki vrne velikost paketa pkt. Zopet, bolj učinkovita in bolj pravična bo vaša rešitev, več točk boste dobili.

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.

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

R: 36

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.[3] 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
```

R: 37

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 `_`.

R: 38

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;
- 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>`.

R: 38

2000.U.4

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:

[3] Podobna naloga je tudi 1998.1.2.

```
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
```

REŠITVE NALOG ZA PRVO SKUPINO

R2000.1.1

N: 1

Z branjem datumov prestopnih sekund lahko ugotovimo, koliko prestopnih sekund je bilo vrinjenih do posameznega datuma. To je število sekund v tisti vrstici datoteke, ki ima najkasnejši datum, manjši ali enak iskanemu datumu.

Med rojstnim in današnjim dnem je torej poleg ustreznega števila dni (kar dobimo kot razliko vrednosti, ki ju za ta dva datuma vrne funkcija MJD) minilo še nekaj sekund — natančneje, tiste prestopne sekunde, ki so se nabrale v dnevih od rojstnega do pred današnjim.

```
program Starost(Input, Output, Prestopne);
type
  DatumT = packed array [1..10] of char;
var
  Prestopne: text;
  Rojstvo, Danes, Kdaj: DatumT;
  Starost, PrestopnihSek: integer;
  PrestopnihSekObRojstvu, PrestopnihSekDanes: integer;

  function MJD(Datum: DatumT): integer; external;

begin
  PrestopnihSekObRojstvu := 0; PrestopnihSekDanes := 0;
  ReadLn(Rojstvo); ReadLn(Danes);
  while not Eof(Prestopne) do begin
    ReadLn(Prestopne, Kdaj, PrestopnihSek);
    if MJD(Kdaj) <= MJD(Rojstvo) then
      PrestopnihSekObRojstvu := PrestopnihSek;
    if MJD(Kdaj) <= MJD(Danes) then
      PrestopnihSekDanes := PrestopnihSek;
  end; {while}
  Starost := (MJD(Danes) - MJD(Rojstvo)) * 24 * 60 * 60 +
             (PrestopnihSekDanes - PrestopnihSekObRojstvu);
```

```
  WriteLn('Od rojstnega do današnjega datuma (ob polnoči) je minilo ',
          Starost, ' sekund.');
end. {Starost}
```

Mimogrede, MJD pomeni *modified Julian day* in je definiran kot MJD := JD − 2400000,5. JD (*Julian day*) šteje čas v dnevih od poldneva, 1. januarja 4713 pr. n. š., MJD pa zato od polnoči, 17. novembra 1858. Glej npr. `http://tycho.usno.navy.mil/mjd.html` in `http://en.wikipedia.org/wiki/Julian_day`.

N: 2

R2000.1.2

Nalogo lahko rešimo na veliko različnih načinov. Eden od najpreprostejših je, da v zanki pregledujemo vse večje potence števila 2, dokler ne naletimo na prvo tako, ki je vsaj tolikšna kot število, ki smo ga dobili kot parameter.

```
function Zaokrozi1(x: integer): integer;
var y: integer;
begin
  if x > 0 then y := 1 else y := 0;
  while y < x do y := y * 2;
  Zaokrozi1 := y;
end; {Zaokrozi1}
```

Če za vhodni parameter x velja $2^{k-1} < x \le 2^k$, se bo zanka izvedla k-krat.

Če zapišemo x v dvojiškem sestavu in ugasnemo vse prižgane bite razen najvišjega, dobimo največjo potenco števila 2, ki je manjša ali enaka x. Recimo tej vrednosti $g(x)$. Na primer: iz $29 = 11101_2$ dobimo $10000_2 = 16$. Nas pa zanima najmanjša potenca števila 2, ki je večja ali enaka x; recimo tej vrednosti $f(x)$. Če je x potenca števila 2, je $g(x) = x = f(x)$, sicer pa je $f(x) = 2g(x)$; tako torej, če izračunamo $g(x)$, ne bo težko dobiti tudi $f(x)$. Lahko pa $f(x)$ izračunamo tudi tako, da upoštevamo, da je $f(x) = 2g(x-1)$, ker je največja potenca števila 2, manjša ali enaka $x-1$, zanesljivo manjša od x in jo moramo zato pomnožiti z 2, da dobimo najmanjšo potenco števila 2, večjo ali enako x.

Oglejmo si zdaj, kako se lahko lotimo ugašanja prižganih bitov. Število x je v računalniku verjetno predstavljeno v dvojiški obliki; na najnižjih mestih je mogoče nekaj ničel, prej ali slej pa nastopi prva enica (če x ni kar enak 0). Če x zmanjšamo za 1, se tiste ničle spremenijo v enice, enica pred njimi pa v ničlo:

$$\begin{aligned} x &= \ldots 1000 \ldots 00, \\ x-1 &= \ldots 0111 \ldots 11, \\ \text{torej } x \textbf{ and } (x-1) &= \ldots 0000 \ldots 00. \end{aligned}$$

Operacija x := x **and** (x − 1) bi torej ugasnila najnižjo enico v številu x. Tako lahko ugašujemo enice eno za drugo; tik preden postane x enak 0, je prižgana le še najvišja enica iz prvotne vrednosti — takrat imamo torej vrednost 2^k, če je bil $2^k \le x < 2^{k+1}$. Zdaj torej število ponovitev zanke ni nujno enako k,

ampak je enako številu enic v dvojiškem zapisu vrednosti x, to pa je vedno manj ali enako k, razen pri $x = 2^{k+1} - 1$, ki ima $k + 1$ enic.

```
function Zaokrozi2(x: integer): integer;
var xPrvotni, y: integer;
begin
  xPrvotni := x; if x <= 0 then y := 0;
  while x > 0 do begin
    y := x;
    x := x and (x − 1);
  end; {while}
  if xPrvotni > y then y := y * 2;
  Zaokrozi2 := y;
end; {Zaokrozi2}
```

Ko se zanka konča, vsebuje y vrednost, ki jo je imel x, tik preden smo v njem ugasnili še zadnjo (najvišjo) enico. Torej, če je bila prvotna vrednost x na intervalu $2^k \le x < 2^{k+1}$, bo y po koncu zanke enak 2^k. Ravno to pa tudi potrebujemo.

Še boljši postopek dobimo, če takoj za stavek xPrvotni := x dodamo še x := x **and not** (x **div** 2) (tako dopolnjeni različici podprograma Zaokrozi2 recimo Zaokrozi2a). To namreč pusti od vsake skupine zaporednih enic goreti le najvišjo:

$$\begin{aligned} x &= \ldots 00\,11111\,00000\,11111\,00000\,11111 \\ x \textbf{ div } 2 &= \ldots 00\,01111\,10000\,01111\,10000\,01111 \\ \textbf{not } (x \textbf{ div } 2) &= \ldots 11\,10000\,01111\,10000\,01111\,10000 \\ x \textbf{ and not } (x \textbf{ div } 2) &= \ldots 00\,10000\,00000\,10000\,00000\,10000. \end{aligned}$$

Tako lahko precej zmanjšamo število prižganih bitov, preden se začne izvajati zanka **while**. Ker mora priti za vsako enico, razen na najnižjem bitu, še vsaj ena ničla, je lahko prižganih največ $\lfloor k/2 \rfloor + 1$ bitov.

Koliko smo s temi izboljšavami pridobili v primerjavi s prvotno rešitvijo? Lahko poskusimo prešteti, koliko iteracij zanke **while** se vsega skupaj izvede, če poženemo zgornje podprograme na vseh številih x iz neke množice. Z nekaj telovadbe pridemo do rezultatov iz spodnje tabele. Vidimo lahko, da porabi Zaokrozi2a približno pol manj iteracij kot Zaokrozi2, ta pa pol manj kot Zaokrozi1.

	Zaokrozi1	Zaokrozi2	Zaokrozi2a
Skupno število iteracij na vseh n-bitnih številih $(2^{n-1}, \ldots, 2^n - 1)$			
	$n2^{n-1} - 1$	$(n+1)2^{n-2}$	$(n+2)2^{n-3}$ (izjema: 1 pri $n = 1$)
OEIS	A001787	A001792	A045623
Skupno število iteracij na vseh številih z največ n biti $(1, \ldots, 2^n - 1)$			
	$(n-1)(2^n - 1)$	$n2^{n-1}$	$(n+1)2^{n-2}$
OEIS	A059672	A001787	A001792

Namesto z ugašanjem bitov lahko rešimo nalogo tudi s prižiganjem. Recimo, da za začetno vrednost x velja $2^k \le x < 2^{k+1}$. Najvišji prižgani bit v dvojiškem zapisu števila x je torej bit k. Če bi prižgali še vse nižje ležeče bite, bi dobili vrednost $1 + 2 + 4 + \ldots + 2^k = 2^{k+1} - 1 = 2g(x) - 1$. Tako ni težko priti do $g(x)$, od tam pa do $f(x)$ z enakim razmislekom kot zgoraj. Za učinkovito prižiganje bitov si pomagajmo z zamikanjem: če je v x že prižganih r najvišjih bitov (torej od $k - r + 1$ do k), bodo v x **shr** r prižgani biti od $k - 2r + 1$ do $k - r$; če števili x in x **shr** r združimo z operatorjem **or**, bodo prižgani vsi biti od $k - 2r + 1$ do k, torej že $2r$ najvišjih bitov. Število zagotovo prižganih bitov se torej v vsakem koraku podvoji, zato bo treba to izvesti največ petkrat, če delamo z 32-bitnimi celimi števili.

```
function Zaokrozi3(x: integer): integer;
begin
  x := x − 1;
  x := x or (x shr 1);
  x := x or (x shr 2);
  x := x or (x shr 4);
  x := x or (x shr 8);
  x := x or (x shr 16);
  Zaokrozi3 := x + 1;
end; {Zaokrozi3}
```

Če pa ne bi vedeli, koliko bitov imajo lahko števila tipa integer, bi lahko naredili zanko in se ustavili, čim bi bili v x prižgani že vsi biti. To prepoznamo tako, da primerjamo x in $x + 1$; če so v x prižgani vsi biti, so v $x + 1$ vsi ti biti ugasnjeni, naslednji bit pa je prižgan. Primer:

$$\begin{aligned} x &= 111111 \\ x+1 &= 1000000 \\ x \textbf{ and } (x+1) &= 0. \end{aligned}$$

```
function Zaokrozi3a(x: integer): integer;
var r: integer;
begin
  x := x − 1; r := 1;
  while x and (x + 1) <> 0 do begin
    x := x or (x shr r);
    r := r shl 1;
  end; {while}
  Zaokrozi3a := x + 1;
end; {Zaokrozi3a}
```

Lepo pri tej različici rešitve je tudi to, da izvede le toliko iteracij, kot je potrebno — na primer, če je $x = 85 = 1010101_2$, bomo že po prvi iteraciji dobili

1111111_2 in se takoj ustavili. V splošnem se izkaže, da potrebujemo $1+\lfloor \log_2 k \rfloor$ iteracij, če je k dolžina najdaljše strnjene skupine ničel v dvojiškem zapisu števila x. Povprečna vrednost k po vseh n-bitnih številih (torej $2^{n-1} \le x < 2^n$) je približno $\log_2 n - 0{,}67$, povprečno število iteracij po vseh n-bitnih številih pa narašča še malo počasneje kot $O(\log \log n)$. Na primer: pri 30-bitnih številih porabi Zaokrozi2a povprečno 8 iteracij, Zaokrozi3a pa 2,67; pri 64-bitnih številih porabi Zaokrozi2a povprečno 16,5 iteracij, Zaokrozi3a pa le 3.[4]

Nalogo lahko rešimo tudi z uporabo števil s plavajočo vejico. Če je $2^{k-1} < x \le 2^k$, bo za dvojiški logaritem veljalo $k - 1 < \log_2 x \le k$, torej $k = \lceil \log_2 x \rceil$. Ko to izračunamo, moramo vrniti vrednost 2^k. Ker običajno nimamo na razpolago posebne funkcije za računanje dvojiškega logaritma, bomo uporabili naravne algoritme in formulo $\log_2 x = (\ln x)/(\ln 2)$.

```
function Zaokrozi4(x: integer): integer;
var y: integer;
begin
  if x <= 0 then Zaokrozi4 := 0
  else begin
    { V pascalu žal nimamo funkcije za zaokrožanje navzgor,
      imamo pa Trunc, ki zaokroža proti 0. }
    y := 1 shl Trunc(Ln(x) / Ln(2));
    if y < x then y := 2 * y;
    Zaokrozi4 := y;
  end; {if}
end; {Zaokrozi4}
```

Slabost tega postopka je, da je računanje naravnega logaritma pogosto precej počasno in je zato celoten postopek počasnejši od prej omenjenih rešitev.

Recimo, da v našem narečju pascala obstaja tip double, ki hrani 64-bitna števila s plavajočo vejico, predstavljena po standardu IEEE 754. Takšno število je sestavljeno iz treh delov: mantise (spodnjih 52 bitov), eksponenta e (naslednjih 11 bitov) in predznaka s (najvišji bit). Če odmislimo posebne primere, kot so neskončnosti, vrednost NaN in denormalizirana števila, je vrednost takega števila (recimo ji x) naslednja: pred 52-bitno mantiso v mislih postavimo enico in „decimalno“ vejico ter dobljeno število preberimo kot neko (mogoče ne-celo) število y, zapisano v dvojiškem sistemu. Nato ga še pomnožimo z 2^{e-1023}; če je predznak $s = 1$, ga pomnožimo še z -1 (pri naši nalogi bomo delali le z nenegativnimi števili in lahko na predznak pozabimo). Ker je $1 \le y < 2$, je $2^{e-1023} \le x < 2^{e-1022}$, torej je $\lfloor \log_2 x \rfloor = e - 1023$. Tako nam logaritma ne bo

[4] Pri 2048-bitnih številih porabi Zaokrozi2a povprečno 512,5 iteracij, Zaokrozi3a pa le 3,998! Žal pa v tem primeru naša analiza tako ali tako ni več pretirano realistična, ker samo šteje iteracije glavne zanke in zanemarja dejstvo, da pri delu z zelo velikimi n-bitnimi števili mnoge operacije (npr. **or** in **and**) ne trajajo konstantno mnogo časa, pač pa $O(n)$ časa. Zato postopek, kot je Zaokrozi3a, takrat ne bi bil posebej primeren.

treba računati s funkcijo Ln, ampak lahko kar izluščimo vrednost e iz števila tipa double. Predpostavili bomo še, da imamo na voljo 64-bitni celoštevilski tip int64. Na spremenljivko tipa double lahko pogledamo, kot da je tipa int64, in jo zamaknemo za 52 bitov v desno („**shr** 52“) ter izključimo vse bite razen spodnjih 11 („**and** 2047“); kar ostane, je ravno iskani eksponent.

```
function Zaokrozi4a(x: integer): integer;
var xx: double; y: integer;
begin
  if x <= 0 then Zaokrozi4a := 0
  else begin
    xx := x;
    y := 1 shl (((int64(xx) shr 52) and 2047) - 1023);
    if y < x then y := 2 * y;
    Zaokrozi4a := y;
  end; {if}
end; {Zaokrozi4a}
```

Tip double smo uporabili zato, ker ima dovolj veliko mantiso, da lahko brez napake hrani poljubno 32-bitno celo število. Če nas zanimajo le manjša števila, bi bil dovolj dober že tip single, ki ima 23-bitno mantiso.

Nalogo lahko rešimo tudi z bisekcijo. Našli bi radi vrednost k, za katero je $2^k \le x < 2^{k+1}$. Med iskanjem vzdržujmo dve števili, k_L in k_D, tako da je $2^{k_L} \le x < 2^{k_D}$; v vsaki iteraciji zanke bomo eno od teh dveh števil spremenili in to tako, da bo ta pogoj še naprej veljal, razlika $k_D - k_L$ pa se bo razpolovila (od tod ime „bisekcija“ — razdeljevanje na dvoje). Tako že v nekaj korakih pridemo do $k_D - k_L = 1$; natančneje povedano, če delamo z največ n-bitnimi števili, postavimo na začetku $k_L = 0$, $k_D = n$ in ker se razdalja med njima v vsaki iteraciji prepolovi, bomo izvedli največ $\lceil \lg n \rceil$ iteracij. Pri 32-bitnih številih pomeni to pet iteracij, pri 64-bitnih pa šest.

```
function Zaokrozi5(x: integer): integer;
var k1, k2, k: integer;
begin
  if x <= 0 then begin Zaokrozi6a := 0; exitend;
  k1 := 0; k2 := 32;
  while k2 - k1 > 1 do begin
    { Na tem mestu velja: 2^k1 <= x < 2^k2. }
    k := (k1 + k2) div 2;
    if x < 1 shl k then k2 := k else k1 := k;
  end; {while}
  { Na tem mestu velja: 2^k1 <= x < 2^(k1+1). }
  k := 1 shl k1;
  if x > k then k := k * 2;
  Zaokrozi5 := k;
end; {Zaokrozi5}
```

Zanko lahko tudi „razvijemo“ v skupino drevesasto gnezdenih pogojnih stavkov. Spodaj je različica, ki predpostavlja, da je vhodni parameter x neko osembitno število, torej $x \leq 256$; če bi hoteli podpreti vsa števila do 2^{16}, bi bil podprogram dvakrat daljši, za vsa 32-bitna števila pa bi bil štirikrat daljši. Lepo pri tej rešitvi je, da se ni več treba ukvarjati s spremenljivkami k, k1 in k2, zato bo podprogram malo hitrejši.

```
function Zaokrozi5a(x: integer): integer;
begin
  if x <= 16 then
    if x <= 4 then
      if x <= 2 then
        if x <= 0 then Zaokrozi5a := 0
        else Zaokrozi5a := x
      else Zaokrozi5a := 4
    else
      if x <= 8 then Zaokrozi5a := 8
      else Zaokrozi5a := 16
  else
    if x <= 64 then
      if x <= 32 then Zaokrozi5a := 32
      else Zaokrozi5a := 64
    else
      if x <= 128 then Zaokrozi5a := 128
      else Zaokrozi5a := 256;
end; {Zaokrozi5a}
```

Recimo, da nas zanimajo le števila z največ n biti, torej $0 \leq x < 2^n$, in da se pojavljajo vsa enako pogosto. Med temi števili je kar polovica n-bitnih, četrtina je $(n-1)$-bitnih in tako naprej; velika števila so torej pogostejša in če hočemo rešitev s čim manjšo povprečno časovno zahtevnostjo, je pametno narediti takšno, ki teče na velikih številih čim hitreje, četudi je na majhnih zato mogoče malo počasnejša. Uporabimo lahko na primer postopek, ki dela podobno kot Zaokrozi1, le da pregleduje potence števila 2 od večjih proti manjšim. Če je $x > 2^{n-1}$, mora naša funkcija vrniti 2^n; če je $x > 2^{n-2}$, mora vrniti 2^{n-1} in tako naprej.

```
function Zaokrozi6(x: integer): integer;
var r: integer;
begin
  if x <= 0 then begin Zaokrozi6 := 0; exit end;
  r := 1 shl (n - 1);
  while r >= x do r := r shr 1;
  if x = 1 then Zaokrozi8 := 1
  else Zaokrozi8 := r * 2;
end; {Zaokrozi8}
```

Če poženemo ta podprogram na vseh x od 0 do $2^n - 1$, se izvede vsega skupaj $2^n - 1$ iteracij zanke **while** — torej v povprečju manj kot ena na vsak x! Če se torej v povprečju res pojavljajo vsi x enako pogosto, bo ta rešitev zelo hitra. Največ časa pa porabi pri majhnih x, zato je manj ugodna, če jo mislimo uporabljati v razmerah, ko bodo prevladovali majhni x, možni pa bodo tudi veliki x, tako da pri inicializaciji spremenljivke r ne smemo uporabiti kakšnega majhnega n. Še hitrejšo različico Zaokrozi6 dobimo, če zanko razvijemo v zaporedje stavkov **if**. Spodaj je različica, ki deluje pravilno za števila do 256; če bi hoteli podpirati tudi večje x, pa bi morali pač dodati na začetku še nekaj takih pogojnih stavkov.

```
function Zaokrozi6a(x: integer): integer;
begin
  if x > 128 then Zaokrozi6a := 256
  else if x > 64 then Zaokrozi6a := 128
  else if x > 32 then Zaokrozi6a := 64
  else if x > 16 then Zaokrozi6a := 32
  else if x > 8 then Zaokrozi6a := 16
  else if x > 4 then Zaokrozi6a := 8
  else if x > 2 then Zaokrozi6a := 4
  else if x > 1 then Zaokrozi6a := 2
  else if x > 0 then Zaokrozi6a := 1
  else Zaokrozi6a := 0;
end; {Zaokrozi6a}
```

Težave s počasnostjo pri majhnih x bi lahko omilili tako, da bi rezultate zanje hranili kar v neki tabeli (ki bi jo inicializirali ob zagonu programa).

Primerjavo hitrosti delovanja različnih tu opisanih rešitev kaže tabela na str. 19.

N: 2

R2000.1.3

Pri tej nalogi ni treba drugega kot slediti navodilom. Ko beremo ocene, jih sproti seštevamo (spremenljivka Vsota), v spremenljivki Negativen pa si zapomnimo, če smo naleteli na kakšno negativno oceno. Na koncu povprečno oceno izračunamo tako, da vsoto delimo z deset.

```
program Spricevalo(Input, Output);
var j, Ocena, Vsota: integer;
    Negativen: boolean;
begin
  Negativen := false; Vsota := 0;
  for j := 1 to 10 do begin
    ReadLn(Ocena);
    Vsota := Vsota + Ocena;
    if Ocena = 1 then Negativen := true;
  end; {for}
```

Funkcija	Opis	Povprečni čas izvajanja [ns] za $x = 1, 2, \ldots, 2^n$ $n = 20$	24	27	30
Zaokrozi1	primerja x z 1, 2, 4, 8, ...	116	136	152	167
Zaokrozi2	ugaša prižgane bite	70	80	87	95
Zaokrozi2a	ugaša skupine prižganih bitov	52	57	61	65
Zaokrozi3	prižiga bite (5 korakov)	39	39	39	39
Zaokrozi3a	prižiga bite (zanka)	35	36	37	37
Zaokrozi4	uporabi Ln	219	219	219	250
Zaokrozi4a	uporabi eksponent v tipu double	60	60	60	60
Zaokrozi5	bisekcija po eksponentih	78	78	75	73
Zaokrozi5a	bisekcija, razvita v stavke **if**	15	14	14	14
Zaokrozi6	primerja x z 2^{n-1}, 2^{n-2}, 2^{n-3}, ...	19	17	17	17
Zaokrozi6a	kot Zaokrozi6 z razvito zanko	13	12	12	12

Vsako od rešitev naloge 2000.1.2 smo pognali na vseh x od 1 do 2^n in skupni procesorjev čas delili z 2^n. Tabela prikazuje dobljene povprečne čase v nanosekundah.

```
  if Negativen then
    WriteLn('Uspeh je negativen.')
  else
    WriteLn('Uspeh je pozitiven. Povprečna ocena je ', Vsota/10:2:1);
end. {Spricevalo}
```

R2000.1.4

N: 3

V prvem prehodu skozi vse podatke si le zapomnimo dolžino najdaljšega imena izvajalca, naslova albuma in naslova skladbe, da bomo kasneje lahko podatke pri izpisu lepo poravnali v stolpce. V drugem prehodu beremo zapise enega za drugim in če ima nek zapis istega izvajalca in album kot prejšnji, moramo izpisati le naslov skladbe (pred tem pa dovolj presledkov); če ima istega izvajalca, a ne album, moramo izpisati album in skladbo; če pa ima tudi drugega izvajalca, izpišemo vse troje. V spremenljivkah PrejIzv in PrejAlbum si zapomnimo izvajalca in album iz prejšnjega zapisa, da ju lahko primerjamo s trenutnim.

```
program Izpisi(Input, Output);

var SirIzv, SirAlb: integer; { širine stolpcev }
    Izvajalec, Album, Skladba: string;
    PrejIzv, PrejAlbum: string;
begin
  SirIzv := 0; SirAlb := 0;
  ZacniSeznam;
  while Komad(Izvajalec, Album, Skladba) do begin
    if Length(Izvajalec) > SirIzv then SirIzv := Length(Izvajalec);
    if Length(Album) > SirAlb then SirAlb := Length(Album);
  end; {while}
  SirIzv := SirIzv + 2; SirAlb := SirAlb + 2); { po dva presledka med stolpci }
```

```
  PrejIzv := ' '; { da bomo prvega izvajalca izpisali v celoti }
  ZacniSeznam;
  while Komad(Izvajalec, Album, Skladba) do begin
    if Izvajalec <> PrejIzv then begin { nov izvajalec }
      WriteLn(Izvajalec, ' ':(SirIzv - Length(Izvajalec)),
              Album, ' ':(SirAlb - Length(Album)), Skladba);
      PrejIzv := Izvajalec; PrejAlbum := Album;
    end else if Album <> PrejAlbum then begin { nov album }
      WriteLn(' ':SirIzv, Album, ' ':(SirAlb - Length(Album)), Skladba);
      PrejAlbum := Album;
    end else begin
      WriteLn(' ':SirIzv, ' ':SirAlb, Skladba);
    end; {if}
  end; {while}
end. {Izpisi}
```

REŠITVE NALOG ZA DRUGO SKUPINO

N: 3

R2000.2.1

V neki globalni spremenljivki si zapomnimo odmik (v urah) ob prejšnjem klicu. Ob vsaki polni uri (torej ko je Minuta = 0) primerjajmo prejšnji odmik s sedanjim, pa bomo zaznali primere, ko se ista ura še enkrat ponavlja ali pa je bila ena ura izpuščena (to si zapomnimo v spremenljivkah IzpustiliUro in PonavljamoUro). Opravila, ki se izvajajo na vsako uro, moramo v tem primeru izvajati kot običajno; v primeru ponavljanja ene ure tistih, ki se morajo izvesti na to uro, zdaj ne smemo izvesti še drugič; v primeru izpuščene ure moramo izvesti tudi tista opravila, ki bi se morala izvesti v izpuščeni prejšnji uri. Ob naslednji polni uri se bosta IzpustiliUro in PonavljamoUro postavili nazaj na false in program bo spet deloval po starem. Glede seznama opravil bomo predpostavili, da ga lahko beremo kar s standardnega vhoda.

```
type
  NizT = packed array [1..64] of char;
var
  PrejsnjiOdmikDefiniran: integer value 0;
  PrejsnjiOdmik: integer;
  IzpustiliUro, PonavljamoUro: boolean;

  procedure Opravi(Opravilo: NizT); external;

procedure PolnaMinuta(UraUTC, Minuta, Odmik: integer);
var
  h, m: integer;
  Opravilo: NizT;
  Stori: boolean;
```

```
begin
  if PrejsnjiOdmikDefiniran = 0 then
    begin PrejsnjiOdmik := Odmik; PrejsnjiOdmikDefiniran := 1 end;
  if Minuta = 0 then begin
    IzpustiliUro := Odmik > PrejsnjiOdmik;
    PonavljamoUro := Odmik < PrejsnjiOdmik;
    PrejsnjiOdmik := Odmik;
  end; {if}
  while not Eof(Input) do begin
    ReadLn(h, m, Opravilo);
    if (m < 0) or (m = Minuta) then begin
      if PonavljamoUro then  { prehod na zimski čas }
        Stori := (h < 0)
      else if IzpustiliUro then { prehod na poletni čas }
        Stori := (h < 0) or (h = UraUTC + Odmik) or (h = UraUTC + Odmik - 1)
      else
        Stori := (h < 0) or (h = UraUTC + Odmik);
      if Stori then Opravi(Opravilo);
    end; {if}
  end; {while}
end; {PolnaMinuta}
```

R2000.2.2

Presek pravokotnika s spodnjim levim ogliščem (x_1, y_1) in zgornjim desnim (x_2, y_2) ter pravokotnika z ogliščema (x_1', y_1') in (x_2', y_2') ima oglišči

N: 5

$$(\max\{x_1, x_1'\}, \max\{y_1, y_1'\}) \quad \text{in} \quad (\min\{x_2, x_2'\}, \min\{y_2, y_2'\}).$$

Potem lahko izračunamo presek med njim in nekim tretjim pravokotnikom in imamo zdaj presek vseh treh. Tako nadaljujemo, pri tem pa lahko še sproti preverjamo, če je presek postal prazen (če desni rob ne leži več desno od levega ali pa zgornji ne več nad spodnjim).

```
const N = ...;                  { število kvadratov }
type Kvadrat = record           { vsak kvadrat je podan z dvema točkama: }
       x1, y1, x2, y2: integer; { spodnjim levim in zgornjim desnim ogliščem }
     end;
var Kvadrati: Kvadrat[1..N];

function PresekObstaja: boolean;
var Presek: Kvadrat; i: integer;
begin
  PresekObstaja := False;
  Presek := Kvadrati[1];
  for i := 2 to N do begin
    Presek.x1 := max(Presek.x1, Kvadrati[i].x1);
```

```
      Presek.y1 := max(Presek.y1, Kvadrati[i].y1);
      Presek.x2 := min(Presek.x1, Kvadrati[i].x1);
      Presek.y2 := min(Presek.y1, Kvadrati[i].y1);
      if (Presek.x1 > Presek.x2) or (Presek.y1 > Presek.y2) then
        exit; { presek ne obstaja oz. je prazna množica }
    end; {for}
    PresekObstaja := true;
end; {PresekObstaja}
```

N: 5

R2000.2.3

Vhodno besedilo beremo znak po znak; ko naletimo na < (ki mu mogoče sledi še /), preberemo ime oznake in nato preskočimo atribute (do znaka >). Če je oznaka na seznamu prepovedanih, ne izpišemo nič, sicer pa samo ime oznake, brez atributov. Besedilo med oznakami izpisujemo nespremenjeno.

```
const N = 10;
var Prepovedana: array [1..N] of string;
var C: char; S: string; EndTag, Ok, Gt: boolean; i: integer;
begin
  while not Eof(Input) do begin
    Read(C);
    if C <> '<' then begin Write(C); continue end;
    { Preberimo ime oznake. }
    S := ''; EndTag := false; Gt := false;
    while not Eof(Input) do begin
      Read(C);
      if C = '/' then EndTag := true { To je oznaka za konec elementa. }
      else if not (C in ['A'..'Z', 'a'..'z', '0'..'9']) then
        begin Gt := C = '>'; break end { Konec imena oznake. }
      else S := S + C;
    end; {while}
    { Berimo do konca oznake (do znaka '>'), da preskočimo atribute.
      Mogoče smo znak '>' celo že prebrali — to pove spremenljivka Gt. }
    if not Gt then while not Eof(Input) do
      begin Read(C); if C = '>' then break end;
    { Preverimo, če je oznaka prepovedana. }
    Ok := true; i := 1;
    while (i <= N) and Ok do
      begin Ok := UpCase(S) <> UpCase(Prepovedana[i]); i := i + 1 end;
    if Ok then begin { Izpišimo oznako. }
      Write('<');
      if EndTag then Write('/');
      Write(S, '>');
    end; {if}
  end; {while}
end.
```

Funkcija UpCase naj bi vrnila niz, v katerem je vsaka mala črka zamenjana z ustrezno veliko. Pri prevajalnikih, ki take funkcije nimajo (ali pa podpira le posamične znake, ne pa nizov), bi si pač lahko pomagali s svojim podprogramom.

R2000.2.4

N: 6

Predpostavili bomo, da je koda reprezentance neko majhno število, ki ga lahko uporabimo kot indeks v tabelo (drugače bi si lahko pomagali z razpršeno tabelo), in da se v razvrstitvi leta 1999 ne pojavlja nobena taka reprezentanca, ki se ne bi že leta 1998. Imeli bomo tabelo imen in položajev v lanski razvrstitvi. Ko beremo letošnjo razvrstitev, prek te tabele poiščemo lanski položaj, izračunamo premik na lestvici in sproti vzdržujemo podatek o tem, katera od doslej prebranih je naredila največji skok (Najboljsa) in kolikšen je ta skok bil (NajvecjiSkok). Na koncu le še izpišemo rezultat.

```
Uvrstitev := 0;
while not Eof(Stanje98) do begin
  Uvrstitev := Uvrstitev + 1;
  PreberiPodatek(Stanje98, ImeReprezentance, KodaReprezentance);
  Ime[KodaReprezentance] := ImeReprezentance;
  Skok[KodaReprezentance] := Uvrstitev;
end; {while}
Uvrstitev := 0;
NajvecjiSkok := -1;
while not Eof(Stanje99) do begin
  Uvrstitev := Uvrstitev + 1;
  PreberiPodatek(Stanje99, ImeReprezentance, KodaReprezentance);
  Skok[KodaReprezentance] := Skok[KodaReprezentance] - Uvrstitev;
  if Skok[KodaReprezentance] > NajvecjiSkok then begin
    NajvecjiSkok := Skok[KodaReprezentance];
    Najboljsa := KodaReprezentance;
  end; {if}
end; {while}
WriteLn('Najboljša je reprezentanca ', Ime[Najboljsa],
        ', ki je naredila skok za ', Skok[Najboljsa], ' mest.');
```

REŠITVE NALOG ZA TRETJO SKUPINO

R2000.3.1

N: 7

Opazujmo orientacijo sten. Dve steni (obe vodoravni ali navpični) morata imeti orientacijo v smeri urinega kazalca; torej, če je neka navpična stena levo od neke druge, mora leva kazati navzgor, desna pa navzdol; podobno velja tudi za vodoravne stene. Ker so točke podane v smeri urinega kazalca, smeri stene ni težko določiti.

nerešljiv rešljiv

Spodnji podprogram zaradi lažjega dostopa do tabele predpostavlja, da je v px[0] in py[0] še ena kopija koordinat px[n] in py[n]. Ker je vsaka druga stena navpična, vse vmes pa vodoravne, je stena, ki se začne pri točki $i \bmod 2$, gotovo vzporedna tisti, ki se začne pri i, tako da lahko gre j od i **mod** 2 naprej s korakom 2.

```
const n = ...;
var px, py: array [0..n] of integer;

function LahkoPostavimoKamero: boolean;
var i, j: integer;
begin
  LahkoPostavimoKamero := false;
  for i := 0 to n - 1 do begin
    j := i mod 2; while j < n do begin
      if px[i] = px[i + 1] then begin  { navpično }
        if (px[i] < px[j]) and (py[i] > py[i + 1]) and (py[j] < py[j + 1]) then exit;
      end else                         { vodoravno }
        if (py[i] > py[j]) and (px[i] > px[i + 1]) and (px[j] < px[j + 1]) then exit;
      j := j + 2;
    end; {while}
  end; {for}
  LahkoPostavimoKamero := true;
end; {LahkoPostavimoKamero}
```

Ta rešitev je časovno precej zahtevna, saj bi pri n stenah kar $(n^2/2)$-krat primerjala orientacijo dveh sten (vsako vodoravno z vsemi $n/2$ vodoravnimi in vsako navpično z vsemi $n/2$ navpičnimi). Obstaja pa tudi učinkovitejši postopek. Označimo položaj kamere s koordinatama (x_k, y_k). Vsaka stena prestavlja neko omejitev glede tega, kje sme biti kamera, da bo to steno videla. Če obstaja neka vodoravna stena od (x_1, y) do (x_2, y) in je $x_1 < x_2$, mora biti kamera pod to steno, da jo bo videla z notranje strani; imamo torej pogoj $y_k < y$, ki je obenem potreben in zadosten. Podobno, če je $x_1 > x_2$, dobimo pogoj $y_k > y$. Za navpične stene od (x, y_1) do (x, y_2) pa pri $y_1 < y_2$ dobimo pogoj $x_k > x$ in pri $y_1 > y_2$ pogoj $x_k < x$. Kamero lahko postavimo na poljubno točko (x_k, y_k), ki ustreza vsem dobljenim pogojem; moramo torej le preveriti, ali sploh obstaja kakšna taka točka.

```
const n = ...;
var px, py: array [1..n] of integer;

function LahkoPostavimoKamero: boolean;
```

```
var i, j, xMin, yMin, xMax, yMax: integer;
begin
  { Na začetku vzemimo, da je sprejemljiv vsak položaj kamere
    v očrtanem pravokotniku (bounding box) cele sobe. }
  xMin := px[1]; xMax := px[1]; yMin := py[1]; yMax := py[1];
  for i := 2 to n do begin
    if px[i] < xMin then xMin := px[i];
    if px[i] > xMax then xMax := px[i];
    if py[i] < yMin then yMin := py[i];
    if py[i] > yMax then yMax := py[i];
  end; {for}
  { Upoštevajmo omejitve, ki jih nalagajo posamezne stene. }
  j := n;
  for i := 1 to n do begin          { Oglejmo si steno od j do i. }
    if px[i] = px[j] then begin     { navpična }
      if (py[i] > py[j]) and (px[i] > xMin) then xMin := px[i];
      if (py[i] < py[j]) and (px[i] < xMax) then xMax := px[i];
    end else begin                  { vodoravna }
      if (px[i] < px[j]) and (py[i] > yMin) then yMin := py[i];
      if (px[i] > px[j]) and (py[i] < yMax) then yMax := py[i];
    end;
    j := i;
  end; {for}
  { Sprejemljivi položaji kamere so vsi, ki ustrezajo pogojema
    xMin < xKamere < xMax in yMin < yKamere < yMax.
    Torej mora biti xMin < xMax in yMin < yMax. }
  LahkoPostavimoKamero := (xMin < xMax) and (yMin < yMax);
end; {LahkoPostavimoKamero}
```

R2000.3.2

N: 7

Mislimo si graf, kjer vsakemu prostemu polju labirinta pripada po ena točka in sta dve točki povezani natanko tedaj, ko imata njuni polji skupno stranico. Ker je v labirintu med vsakim parom polj natanko ena pot, je ta graf pravzaprav drevo. Eno od polj (vseeno je, katero) si izberimo za koren drevesa. Čim imamo koren, lahko v drevo vpeljemo relacije tipa starši–potomci: od korena do poljubne druge točke u obstaja natanko ena pot in u-jev oče je zadnja točka pred u na tej poti. Ko imamo v drevesu takšno strukturo, lahko govorimo tudi o poddrevesih, ta pa nam omogočajo, kot bomo videli, precej elegantno določiti najdaljšo pot v drevesu.

Za koren lahko vzamemo na primer kar prvo prosto polje v matriki Labirint. Preostanek drevesa lahko potem preiščemo z iskanjem v globino. To je eden od znanih postopkov za sistematično pregledovanje grafov. Na začetku si mislimo, da je vsaka točka še nepregledana. Poženimo naslednji podprogram in mu kot parameter podajmo koren drevesa:

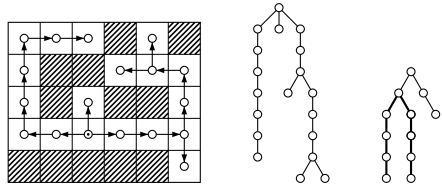
```
procedure IskanjeVGlobino(a: Tocka);
```

```
begin
  Pregledana[a] := true;
  za vsako a-jevo sosedo b:
    if not Pregledana[b] then
      { Točka a je roditelj točke b. Preglejmo zdaj poddrevo, ki se prične pri b. }
      IskanjeVGlobino(b);
end; {IskanjeVGlobino}
```

Ko se torej ta postopek pokliče za neko točko a, se bo zakopal v a-jeva poddrevesa in se ne bo vrnil, dokler ne bo rekurzivno pregledal vsega, kar se v drevesu nahaja pod točko a, pa ne glede na to, kako globoko bo treba iti. Odtod tudi izraz „iskanje v globino“.

Ker naloga zagotavlja, da naši labirinti ne vsebujejo cikličnih poti, je med a-jevimi sosedami že pregledana le tista, iz katere smo prišli v a. Zato ni nujno imeti posebne tabele Pregledana, ampak si lahko zgolj zapomnimo, od kod smo prišli v trenutno točko.

Primer labirinta in drevesa, ki ga odkrijemo v njem, kažeta sliki $(1a)$ in $(1b)$ spodaj.



$(1a)$ $(1b)$ (2)

$(1a)$ Primer odkrivanja drevesa v labirintu. Polje, pri katerem smo začeli, je označeno z $\odot$. $(1b)$ Dobljeno drevo, če bi ga narisali na bolj tradicionalen način. (2) Primer drevesa, pri katerem najdaljša pot (označena z debelimi črtami) ne poteka skozi koren drevesa.

Zdaj lahko najbolj oddaljeni točki v drevesu poiščemo tako, da začnemo pri listih drevesa in napredujemo proti korenu, pri tem pa upoštevamo naslednji rekurzivni razmislek: najdaljša pot v drevesu lahko koren drevesa obišče ali pa ne. (i) Če pot gre skozi koren, jo koren razdeli na dva kosa, vsak od njiju pa gre lahko le še navzdol po drevesu (saj iz korena ne moremo narediti koraka navzgor, torej gre prvi korak navzdol; koraku navzdol pa v drevesu ne more slediti korak navzgor, saj bi se tako le vrnili nazaj v točko, iz katere smo ravnokar prišli, takih poti pa naloga ne dovoli). Zato je najdaljša tovrstna pot tista, pri kateri se ta dva konca poti spustita v najgloblji poddrevesi. (ii) Če pa najdaljša pot ne gre skozi koren drevesa (glej npr. drevo (2) na sliki), pomeni, da v celoti leži znotraj enega od poddreves (kajti pot ne more iti iz enega poddrevesa v drugo, ne da bi šla pri tem skozi koren); torej mora biti to najdaljša pot tistega poddrevesa.

Najdaljšo pot bomo torej dobili tako, da bomo od opisanih dveh možnosti vedno vzeli tisto, ki nam da daljšo pot. Preden lahko določimo najdaljšo pot (in globino) nekega drevesa, moramo poznati najdaljše poti in globine njegovih poddreves. (Ta rekurzija se konča pri listih drevesa, torej takrat, ko trenutna točka sploh nima poddreves. Takrat je edina in najdaljša pot ta, ki obišče dani list in nič drugega.) Zato lahko računanje najdaljših poti in globin lepo vključimo v zgoraj opisani postopek iskanja v globino: primeren trenutek za izračun najdaljše poti in globine za poddrevo s korenom v točki a je takoj po tistem, ko se vrnemo iz rekurzivnih klicev za a-jeve otroke (in preden se vrnemo iz rekurzivnega klica za samo točko a); takrat imamo že pripravljene vse potrebne podatke o a-jevih poddrevesih.

```
program NajdaljsaPot;
const MaxVisina = ...; MaxSirina = ...;
      DX: array [1..4] of longint = (0, 0, −1, 1);
      DY: array [1..4] of longint = (1, −1, 0, 0);
type Polje = (Prosto, Zid);
var Labirint: array [1..MaxVisina, 1..MaxSirina] of Polje;
    Sirina, Visina: LongInt;

  { Pregleda poddrevo, ki se začne pri (X, Y); vrne globino poddrevesa in dolžino
    najdaljše poti v njem. (PX, PY) sta koordinati starša polja (X, Y) v drevesu. }
  procedure Drevo(X, Y, PX, PY: longint; var Glob, Dolz: longint);
  var Smer, XC, YC, GC, DC, Glob2: longint;
  begin
    Glob := 0; Glob2 := 0; Dolz := 0;
    for Smer := 1 to 4 do begin
      { Oglejmo si poddrevo, ki se začne pri (XC, YC). }
      XC := X + DX[Smer]; YC := Y + DY[Smer];
      if (XC = PX) and (YC = PY) then continue;
      if (XC < 1) or (XC > Sirina) or (YC < 1) or (YC > Visina) then continue;
      if Labirint[YC, XC] = Zid then continue;
      Drevo(XC, YC, X, Y, GC, DC);
      if GC > Glob then begin Glob2 := Glob; Glob := GC end
      else if GC > Glob2 then Glob2 := GC;
      if DC > Dolz then Dolz := DC;
    end; {for}
    { Zdaj imamo v Glob in Glob2 globini dveh najglobljih poddreves. }
    DC := Glob + Glob2 + 1; if DC > Dolz then Dolz := DC;
    Glob := Glob + 1; { Globina drevesa = globina najglobljega poddrevesa + 1. }
  end; {Drevo}

var X, Y, XR, YR, Glob, Dolz: longint; C: char;
begin
  { (XR, YR) bo koren drevesa. To je lahko poljubno prosto polje. }
  ReadLn(Sirina, Visina); XR := −1; YR := −1;
```

```
  for Y := 1 to Visina do begin
    for X := 1 to Sirina do begin
      Read(C);
      if C = '#' then Labirint[Y, X] := Zid
      else begin Labirint[Y, X] := Prosto; XR := X; YR := Y end;
    end; {for X}
    ReadLn;
  end; {for Y}
  Drevo(XR, YR, -1, -1, Glob, Dolz);
  WriteLn('Dolžina najdaljše poti: ', Dolz, '.');
end. {NajdaljsaPot}
```

Ta rešitev je lahko nekoliko potratna s pomnilnikom, če gredo rekurzivni klici zelo globoko. Načeloma bi lahko prosta polja tvorila eno samo dolgo kačasto pot, ki bi obsegala približno polovico vseh polj v labirintu. Rekurzivni klici podprograma Drevo se lahko tedaj gnezdijo do globine približno $\mathsf{Sirina} \cdot \mathsf{Visina}/2$, pri vsakem klicu pa se na sklad naložijo vsi parametri, lokalne spremenljivke in še kakšni knjigovodski podatki (npr. naslov za vrnitev iz klica). V takih primerih bi lahko prihranili precej pomnilnika, če bi rekurzivne klice simulirali sami; dovolj bi bilo, če bi za vsako celico hranili vrednosti Glob (globina poddrevesa, ki se začenja pri tej celici), Dolz (dolžina najdaljše poti v tem poddrevesu) in podatek o tem, katera od sosednjih točk je njen roditelj v drevesu (slednje bi lahko hranili kar v tabeli Labirint). Vendar pa je zdaj poraba pomnilnika za te dodatne podatke (poleg samega labirinta) vedno sorazmerna s številom vseh polj v labirintu (ker moramo pač rezervirati toliko pomnilnika za obe tabeli), medtem ko je bila prej sorazmerna z globino najglobljega drevesa (toliko je bilo namreč največ gnezdenih rekurzivnih klicev). Zato je ta prijem koristen le, če bomo imeli opravka z zelo izrojenimi drevesi. Na testnih podatkih z ACMovega tekmovanja (CERC 1999) je bil spodnji program nekaj počasnejši od gornjega (1,75 s namesto 1,2 s).

```
program NajdaljsaPot;
type PoljeSmer = (Prosto, Zid, Gor, Dol, Levo, Desno);
const MaxVisina = ...; MaxSirina = ...;
      DX: array [Gor..Desno] of longint = (0, 0, -1, 1);
      DY: array [Gor..Desno] of longint = (1, -1, 0, 0);
var Labirint: array [1..MaxVisina, 1..MaxSirina] of PoljeSmer;
    Glob, Dolz: array [1..MaxVisina, 1..MaxSirina] of longint;
    Sirina, Visina: longint;
    X, Y, XC, YC, XR, YR, Dolz1, DC, Glob1, Glob2, GC: longint;
    C: char; Smer: PoljeSmer; Nasel: boolean;
begin
    { Preberimo labirint. Za koren (XR, YR) izberimo poljubno prosto polje. }
    ReadLn(W, H); XR := -1; YR := -1;
    for Y := 1 to H do begin
```

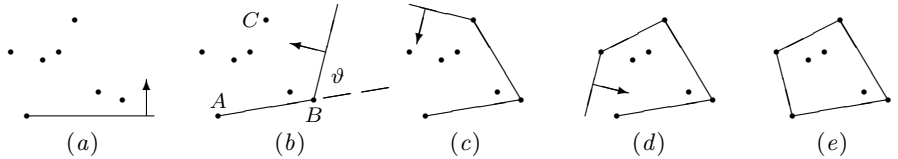
```
      for X := 1 to W do begin
        Read(C);
        if C = '#' then Labirint[Y, X] := Zid
        else begin Labirint[Y, X] := Prosto; XR := X; YR := Y end;
      end; {for X}
      ReadLn;
    end; {for Y}
    if XR < 0 then begin WriteLn('Dolžina najdaljše poti: ', 0, '.'); exit end;

    Labirint[YR, XR] := Zid; X := XR; Y := YR; { Koren drevesa. }
    Smer := Pred(Gor);
    while true do begin
      { Poskusimo najti naslednjega otroka polja (X, Y). }
      Nasel := false;
      while Ord(Smer) < Ord(Desno) do begin
        Smer := Succ(Smer); XC := X + DX[Smer]; YC := Y + DY[Smer];
        if (XC <= 1) or (XC > W) or (YC <= 1) or (YC > H) then continue;
        if Labirint[YC, XC] <> Prosto then continue; { Zid ali pa naš roditelj. }
        { V Labirint[YC, XC] si zapomnimo, v kakšni smeri smo se premaknili. }
        X := XC; Y := YC; Labirint[Y, X] := Smer; Nasel := true; break;
      end; {while}
      if Nasel then begin Smer := Pred(Gor); continue end; { „rekurzivni klic“ }

      { Izračunajmo Glob in Dolz pri (X, Y) s pomočjo vrednosti pri otrocih. }
      Glob1 := 0; Glob2 := 0; Dolz1 := 0;
      for Smer := Gor to Desno do begin
        XC := X + DX[Smer]; YC := Y + DY[Smer];
        { Preverimo, če ni (XC, YC) slučajno naš roditelj. }
        if (X <> XR) or (Y <> YR) then if (XC = X - DX[Labirint[Y, X]])
          and (YC = Y - DY[Labirint[Y, X]]) then continue;
        if (XC <= 1) or (XC > W) or (YC <= 1) or (YC > H) then continue;
        if Labirint[YC, XC] = Zid then continue;
        GC := Glob[YC, XC]; DC := Dolz[YC, XC];
        if GC > Glob1 then begin Glob2 := Glob1; Glob1 := GC end
        else if GC > Glob2 then Glob2 := GC;
        if DC > Dolz1 then Dolz1 := DC;
      end; {for}
      DC := Glob1 + Glob2 + 1; if DC > Dolz1 then Dolz1 := DC;
      Dolz[Y, X] := Dolz1; Glob[Y, X] := Glob1 + 1;

      { Premaknimo se nazaj na starša in si zapomnimo tudi smer,
        od katere bo treba nadaljevati s pregledovanjem otrok. }
      if (X = XR) and (Y = YR) then break; { Preiskali smo že celo drevo. }
      Smer := Labirint[Y, X]; X := X - DX[Smer]; Y := Y - DY[Smer];

    end; {while}
    WriteLn('Dolžina najdaljše poti: ', Dolz[YR, XR], '.');
end. {NajdaljsaPot}
```

N: 8

R2000.3.3

Problem, ki ga rešujemo pri tej nalogi, je v literaturi o računski geometriji znan kot problem iskanja konveksne ovojnice (*convex hull*). Konveskna ovojnica neke množice točk je najmanjši tak konveksni lik (ali telo, če smo v več dimenzijah), ki vsebuje vse te točke. Lik je konveksen takrat, ko za vsak par točk, ki ju vsebuje, vsebuje tudi celo daljico med njima; bolj preprosto si lahko to predstavljamo tako, da si mislimo, da je lik vsepovsod izbočen (ali pa raven), nikjer pa vbočen. Pri mnogokotniku, torej liku z ravnimi stranicami, je ta pogoj enakovreden zahtevi, naj bo notranji kot pri vsakem oglišču manjši od 180°. Ograja, ki jo hočemo postaviti okoli sadovnjaka, bo morala iti ravno po robu konveksne ovojnice, da bo najkrajša.

Naloge se lahko lotimo na več načinov. Če poiščemo najnižje ležečo točko (recimo ji A), torej tisto z najmanjšo koordinato y, vemo, da bo gotovo ležala na robu ovojnice. Mislimo si, da bi na to točko privezali en konec elastične vrvice, nato pa bi šli z drugim koncem okoli sadovnjaka (glej sliko); elastika bi se pri tem ovijala okoli dreves na robu ovojnice in na koncu, ko bi prišli naokoli in privezali drugi konec za drevo, kjer smo začeli, bi tekla elastika ravno po robu celotne konveksne ovojnice.



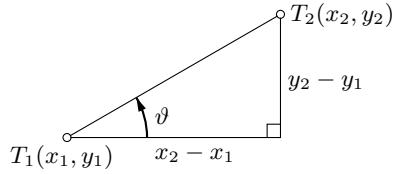
Vidimo, da bo naslednja točka na robu ovojnice (gledano v pozitivni smeri, torej nasproti smeri urinega kazalca) tista B, za katero bo kot med smerjo AB in vodoravno smerjo desno od A najmanjši. V nadaljevanju se elastika ovija okoli točke B in prva točka, ob katero zadene, je tista C, za katero je kot ϑ med staro smerjo AB in novo smerjo BC najmanjši. Potem ovijamo elastiko okoli C in tako naprej, dokler se ne vrnemo nazaj na začetek, v točko A.

```
type Tocka = record x, y: real end;
const N = ...;
var p: array [1..N + 1] of Tocka;

{ Vrne smerni kot daljice od t1 do t2 v stopinjah. }
function Theta(t1, t2: Tocka): real;
const eps = 1e−5; { Majhno pozitivno število. }
var dx, dy, Kot: real;
begin
  dx := t2.x − t1.x; dy := t2.y − t1.y;
  if Abs(dx) < eps then begin { navpičen vektor }
    if dy >= eps then Kot := Pi * 0.5 else Kot := Pi * 1.5
  end else begin
    Kot := ArcTan(dy / dx);
```

```
    if dx < 0 then Kot := Kot + Pi              { (−π/2, π/2) → (π/2, 3π/2) }
    else if dy < 0 then Kot := Kot + 2 * Pi;    { (−π/2, 0) → (3π/2, 2π) }
  end; {if}
  Theta := Kot * 180.0 / Pi; { Preračunajmo radiane v stopinje. }
end; {Theta}

procedure Ovojnica;
var i, Min, M: integer;
    MinKot, v: real;
    t: Tocka;
begin
  Min := 1;
  for i := 2 to N do
    if p[i].y < p[Min].y then Min := i;
  M := 0;
  p[N + 1] := p[Min];
  MinKot := 0.0;
  repeat
    { Na ovojnici že imamo točke p[1..M] v tem vrstnem redu,
      vemo pa tudi, da bo p[Min] naslednja. Torej jo premaknimo v p[M + 1]. }
    M := M + 1; t := p[M]; p[M] := p[Min]; p[Min] := t;
    Min := N + 1; v := MinKot; MinKot := 360.0;
    { Poglejmo zdaj, katera od preostalih točk, torej p[M..N + 1] (na mestu N + 1
      imamo kopijo prve točke z ovojnice, ker bomo morali ovojnico še speljati
      nazaj do nje), oklepa najmanjši kot s staro smerjo. Obenem vemo, da bo
      naklon nove stranice večji kot naklon prejšnje, ki je v stopinj. }
    for i := M + 1 to N + 1 do
      if Theta(p[M], p[i]) > v then
        if Theta(p[M], p[i]) < MinKot then
          begin Min := i; MinKot := Theta(p[m], p[Min]) end;
  until Min = N + 1;
end; {Ovojnica}
```

Funkcija Theta si pri računanju kotov pomaga s funkcijo arc tg (glej sliko na str. 32), vendar pa mora posebej paziti na (skoraj) navpične vektorje (pri katerih bi lahko prišlo do deljenja z 0) in na primere, ko je sprememba x-koordinate negativna (ker arc tg vedno vrne vrednost z intervala $(-\pi/2, \pi/2)$). Če hočemo na koncu vračati kote z intervala $[0, 2\pi)$ namesto $(-\pi/2, 3\pi/2]$, moramo posebej paziti še na to.

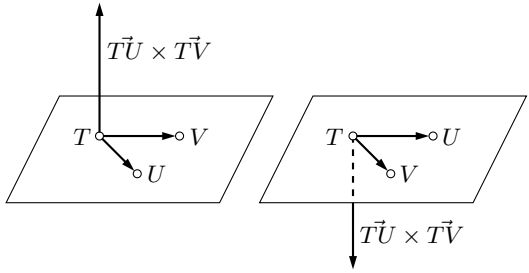
Pogoja v notranji zanki (**for** i) bi lahko še malo poenostavili in se izognili potrebi po funkciji Theta, če bi uporabili vektorski produkt. Naj bo T prejšnja točka na ovojnici, U doslej najobetavnejša kandidatka za naslednjo točko, V pa še neka nova točka, za katero moramo zdaj preveriti, če ni mogoče še boljša od U. V bomo sprejeli kot novo najboljšo kandidatko, če je kot med smerjo



Smerni kót daljice od T_1 do T_2 lahko izračunamo s pomočjo funkcije arc tg. Za pravokotni trikotnik na sliki velja $\operatorname{tg}\vartheta = \dfrac{y_2 - y_1}{x_2 - x_1}$, torej je $\vartheta = \operatorname{arc\,tg}\dfrac{y_2 - y_1}{x_2 - x_1}$.

prejšnje stranice in smerjo TV manjši kot med smerjo prejšnje stranice in TU; toda ta pogoj je enakovreden temu, da je TV desno od TU (oz. natančneje: če se hočemo iz smeri TU zasukati v smer TV in pri tem napraviti kot, manjši od 180°, se moramo zasukati nasproti smeri urinega kazalca). Recimo, da smer TU opisuje vektor $\vec{TU} = (x_1, y_1)$, smer TV pa $\vec{TV} = (x_2, y_2)$. Delajmo se, da sta to 3-d vektorja z z-koordinato 0 in izračunajmo njun vektorski produkt $\vec{TU} \times \vec{TV}$.

Spomnimo se, da je vektorski produkt definiran tako, da je pravokoten na $\vec{TU}$ in $\vec{TV}$ in da za njegovo smer velja pravilo desne roke: če pokažemo s palcem desne roke v smer $\vec{TU}$, s kazalcem pa v smer $\vec{TV}$, lahko pokažemo s sredincem v smer $\vec{TU} \times \vec{TV}$, v nasprotno smer pa ne (oz. težko).



Če imamo v neki ravnini podane tri točke, T, U in V, lahko iz smeri vektorskega produkta $\vec{TU} \times \vec{TV}$ ugotovimo, ali gre pri premiku iz smeri TU v smer TV za premik v levo ali v desno.

Torej, če je zasuk iz smeri TU v smer TV pozitiven, kaže vektorski produkt $\vec{TU} \times \vec{TV}$ nad ravnino, v kateri ležijo točke T, U in V, sicer pa pod njo. V našem primeru si lahko mislimo, da ležijo vse tri točke na ravnini $z = 0$, torej lahko to, ali kaže $\vec{TU} \times \vec{TV}$ nad ali pod njo, preverimo preprosto tako, da pogledamo predznak njegove z-koordinate. Iz običajne formule za vektorski produkt sledi, da je ta v našem primeru enak $(x_1, y_1, 0) \times (x_2, y_2, 0) = (0, 0, x_1 y_2 - x_2 y_1)$. Torej lahko pogoj, da mora biti smer TV desno od smeri TU, zamenjamo s pogojem $x_1 y_2 - x_2 y_1 < 0$. Tako nam ne bo treba računati s koti ali kakšnimi drugimi nezaželenimi operacijami s plavajočo vejico.

```
Min := M + 1;
for i := M + 2 to N + 1 do then
  if (p[i].x − p[M].x) * (p[Min].y − p[M].y) <
     (p[i].y − p[M].y) * (p[Min].x − p[M].x) then Min := i;
```

Pomembno je, da imamo v pogoju $<$ in ne $\leq$; drugače bi pri iskanju druge točke ovojnice (torej pri $M = 1$), ko bi i prišel do $N + 1$, računali vektorski

produkt med vektorjem od p[M] do p[Min] ter vektorjem od p[M] do p[N + 1]; ker pa je $M = 1$ in je v p[N + 1] kopija začetne točke, je drugi od teh dveh vektorjev ničelni, zato bi dobili vektorski produkt 0 in pogoj $\leq$ bi bil zagotovo izpolnjen; tako bi naš program takoj speljal ovojnico nazaj v začetno točko in končal. Pogoj $<$ pa v takem primeru zanesljivo ni izpolnjen.

Slabost opisanega postopka (ki mu običajno pravijo „zavijanje darila“ ali paketa, pa tudi „Jarvisov obhod“) je, da mora preiskati vse točke (razen tistih, ki so že na robu ovojnice), da dobi naslednjo stranico ovojnice. Če ima ovojnica h stranic, je časovna zahtevnost tega postopka $O(nh)$; in v najslabšem primeru so lahko na robu ovojnice kar vse točke (velja $h = n$). Obstajajo tudi učinkovitejši postopki s časovno zahtevnostjo $O(n \lg n)$ in celo $O(n \lg h)$.[5]

Preden začnemo računati konveksno ovojnico, lahko z naslednjim preprostim razmislekom zavržemo še nekaj točk, da se nam kasneje ne bo treba ukvarjati z njimi: če neki množici točk dodamo neko novo točko, ki leži znotraj konveksne ovojnice dosedanje množice, se konveksna ovojnica nič ne spremeni. Zato velja tudi obratno: če izberemo nekaj točk iz naše množice, lahko potem iz nje zavržemo vse, ki ležijo v njihovi konveksni ovojnici, pa se konveksna ovojnica cele množice ne bo zato nič spremenila. Lahko bi na primer izbrali najbolj levo, najnižjo, najbolj desno in najvišjo točko cele množice; konveksna ovojnica teh štirih je kar štirikotnik, ki jih obišče vse štiri v navedenem vrstnem redu (lahko je tudi izrojen v trikotnik ali celo daljico, če tiste štiri točke niso vse različne); zdaj ni težko za vsako od preostalih točk preveriti, če slučajno leži v tem štirikotniku, in če je to res, jo lahko zavržemo. Seveda pa ni nujno, da se bomo na ta način res znebili kakšne točke (npr. če so bile razporejene po neki krožnici).

R2000.3.4

N: 8

Če bi naloga zahtevala, da je *število* odposlanih podatkov za posamezno cev enako, bi bila pravilna rešitev preprosta vrsta (*queue*, FIFO). Ker pa naloga zahteva, da je enaka *količina* odposlanih podatkov, je prava rešitev *prioritetna vrsta* (*priority queue*). Ker lahko uporabimo poljubno rešitev prioritetne vrste iz literature, bomo v tej rešitvi predpostavili, da le-ta že obstaja in z njo tudi operacije PripraviPV(PV), VstaviPV(PV; Prioriteta; Paket) in IzlociNajmPV(PV; **var** Prioriteta; **var** Paket). Vloga prvih dveh operacij je očitna, medtem ko slednja izloči iz prioritetne vrste PV element z najmanjšo prioriteto. Pri tem predpostavljamo, da operacija vrne false, če ni v prioritetni vrsti PV nobenega elementa, in true, če je. V slednjem primeru je izločeni element paket Paket s prioriteto Prioriteta.

Prioriteta elementa (paketka) določa, kako zgodaj bo paketek zapustil usmerjevalnik; nižja ko je, prej bo odšel. Zato za vsako vhodno cev vodimo

[5] Literatura: kaj o računski geometriji, dovolj bo že ustrezno poglavje v Cormen *et al.*, *Introduction to Algorithms* (35. pogl. v prvi izdaji, 33. v drugi). Preparata in Shamos (*Computational Geometry: An Introduction*, 2nd ed., Springer, 1988) pa imata o konveksnih ovojnicah dve precej zajetni poglavji.

trenutno prioriteto, ki pove, koliko podatkov je doslej prišlo iz te cevi.

Takšna rešitev je pravilna, če iz vseh vhodnih cevi kar naprej prihajajo paketki. Ko pa iz neke cevi nekaj časa ni paketkov, bi bili naslednji iz nje prispeli paketki takoj odposlani, kar pa ni pravično — saj bi v tistem trenutku ta cev dobila večjo količino izhodne cevi od vseh ostalih. (Predstavljajmo si na primer, da imamo dve vhodni cevi in je en dan le ena pošiljala podatke, druga pa nič. Če hočeta naslednji dan pošiljati obe, mi pa bi gledali le količino poslanih podatkov, bi imela zdaj tista cev, ki včeraj ni poslala ničesar, danes ves dan vso izhodno pasovno širino zase, dokler ne bi poslala toliko podatkov, kolikor jih je prva cev poslala včeraj; šele potem bi začeli deliti izhodno pasovno širino enakomerno med obe cevi.)

Zato poleg količine podatkov, prispelih iz vsake cevi, vodimo še podatek, koliko podatkov je doslej imela vsaka cev možnost poslati. Namreč, da bi bilo porazdeljevanje pravično, moramo predpostaviti, da je vsaka cev (tudi tista, ki je trenutno prazna) prejela (in zatorej tudi odposlala naprej) vsaj toliko podatkov.

Najprej definicije vseh tipov, ki jih bomo potrebovali v rešitvi:

```
const N = 100;                          { Število vhodnih cevi. }
type
  PaketekT = ...;                       { En paketek. }
  PrioritetnaVrstaT = ...;              { Prioritetna vrsta. }
  StrukturaT = record
    PV: PrioritetnaVrstaT;              { Prioritetna vrsta. }
    Prejeto: array [1..N] of integer;   { Količina prejetih podatkov po vsaki cevi. }
    Poslano: integer;                   { Koliko podatkov je doslej imela možnost
                                          poslati vsaka cev. }
  end; {StrukturaT}
```

V resničnih okoljih bi bil PaketekT definiran kot **struct** mbuf v BSD Unixih in **struct** sk_buff v Linux-ih.

In sedaj posamezne operacije. Najprej Pripravi:

```
procedure Pripravi(var Str: StrukturaT);
var Cev: integer;
begin
  PripraviPV(Str.PV);       { Pripravimo prioritetno vrsto. }
  for Cev := 1 to N do      { Iz nobene cevi ni bilo še nič prejetega... }
    Str.Prejeto[Cev] := 0;
  Str.Poslano := 0;         { ... in nič poslanega. }
end; {Pripravi}
```

Nato Vstavi, ki se pokliče, ko pride paketek:

```
procedure Vstavi(var Str: StrukturaT; Paket: PaketekT; Cev: integer);
```

```
var Prioriteta: integer;
begin
  { Če je cev doslej prejela manj podatkov, kot jih je imela možnost poslati,
    je to njen problem in zato predpostavimo, da je v resnici prejela
    vsaj Str.Poslano podatkov. }
  if Str.Prejeto[Cev] < Str.Poslano
  then Prioriteta := Str.Poslano
  else Prioriteta := Str.Prejeto[Cev];
  Prioriteta := Prioriteta + Velikost(Paket);    { Tole bo prioriteta tega paketka, }
  VstaviPV(Str.PV, Prioriteta, Paket);  { ki ga zdaj vstavimo v prioritetno vrsto. }
  { Popravimo še količino prejetih podatkov za to cev. }
  Str.Prejeto[Cev] := Prioriteta;
end; {Vstavi}
```

Na koncu še Naslednji, ki je klicana, ko izhodna cev želi poslati naslednji paketek:

```
function Naslednji(var Str: StrukturaT; Paket: PaketekT): boolean;
var Cev, Prioriteta: integer;
begin
  if not IzlociNajmPV(Str.PV, Prioriteta, Paket) then
    Naslednji := false      { Nobenega paketka ni na voljo. }
  else begin
    { Funkcija Vstavi pripiše vsakemu novemu paketu prioriteto tako, da vzame
      količino podatkov, ki jih je oz. bo tista cev že poslala naprej
      (ali pa vsaj imela možnost poslati), preden bo prišel na vrsto tudi
      trenutni paket; temu številu pa prišteje še dolžino trenutnega paketa.
      Če ima paket, ki ga zdajle odpošiljamo, prioriteto Prioriteta, pomeni,
      da je doslej vsaka vhodna cev poslala ali pa vsaj imela možnost poslati
      naprej Prioriteta podatkov. Zato vpišimo to trenutno prioriteto v polje
      Str.Poslano, ki je namenjeno shranjevanju prav tega podatka. }
    Str.Poslano := Prioriteta;      { Največ toliko podatkov je bilo }
                                    { doslej poslanih iz vsake cevi. }
    Naslednji := true;
  end; {if}
end; {Naslednji}
```

Oglejmo si primer delovanja opisane rešitve. Recimo, da imamo dve vhodni cevi in je nekaj časa pošiljala podatke samo prva cev, nato pa začnejo prihajati podatki po obeh ceveh:

Dogodek	Stanje strukture po tem dogodku		
	Prioritetna vrsta	Prejeto	Poslano
	prazna	(100, 0)	100
iz 1. cevi pride paket A dolžine 10	$(A, 110)$	(110, 0)	100
iz 2. cevi pride paket B dolžine 5	$(B, 105), (A, 110)$	(110, 105)	100
iz 2. cevi pride paket C dolžine 10	$(B, 105), (A, 110), (C, 115)$	(110, 115)	100
izhodna cev odpošlje paket B	$(A, 110), (C, 115)$	(110, 115)	105
izhodna cev odpošlje paket A	$(C, 115)$	(110, 115)	110
iz 2. cevi pride paket D dolžine 10	$(C, 115), (D, 125)$	(110, 125)	110
izhodna cev odpošlje paket C	$(D, 125)$	(110, 125)	115
iz 1. cevi pride paket E dolžine 20	$(D, 125), (E, 135)$	(135, 125)	115

Tukaj opisano rešitev najdemo v visoko zmogljivih usmerjevalnikih novejše generacije. Edina kritična točka zgornje rešitve je učinkovita izvedba prioritetne vrste. Običajno se pri slednji tudi zalomi, saj klasične rešitve „iz knjig“ zahtevajo $O(\log m)$ časa za vsako od operacij (m je tukaj število čakajočih paketkov). V večini primerov je to preveč in zato se izdelovalci poslužujejo drugih (približnih) rešitev.

REŠITVE NALOG DRUGEGA TEKMOVANJA IZ UNIXA

N: 10

R2000.U.1

Rešitev v `awk`:

```
{
  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).

R2000.U.2

N: 10

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
```

N: 10

R2000.U.3

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"
```

N: 10

R2000.U.4

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).