Dodatne naloge

Včasih se ob pripravljanju nalog za tekmovanje nabere več nalog, kot jih tisto leto potrebujemo. Nekaj nalog torej ostane neuporabljenih, kar pa še ne pomeni, da so slabe ali nezanimive. Nekaj takšnih nalog je predstavljenih v tem razdelku. Opisi nalog, še posebej pa rešitve, praviloma niso tako dodelani kot pri nalogah, ki so bile uporabljene na tekmovanjih.

2002.X.1 Mobilni milijonar

Konec lanskega leta je veliko navdušenje povzročila Mobitelova igra Mobilni milijonar. Po zgledu Jonasovega Milijonarja na POP TV je bilo treba prek sporočil SMS pravilno in čim hitreje odgovoriti na zaporedje desetih vprašanj. Ker je pisanje sporočil SMS na mobilni telefon zamudno, so spretni programerji napisali program, ki je krmilil njihov mobilni telefon in odgovarjal na vprašanja. Pri tem jim je šlo na roko dejstvo, da je bilo vprašanj končno mnogo, ter da je sistem sporočal pravilni odgovor, če je igralec poslal napačnega (seveda je bilo v tem primeru igre konec in je bilo potrebno začeti znova). Program se je tako lahko, če je bil dovolj vztrajen, sam naučil pravilne odgovore na vprašanja in v mnogih poizkusih celo uspel pravilno odgovoriti na zaporedje desetih vprašanj. Če je pri tem imel še malo sreče, da je bil najhitrejši v odgovarjanju na zaporedje vprašanj (Mobitelovo omreže je bilo v tistem času prezasedeno s sporočili SMS za Mobilnega milijonarja, zato igralec ni mogel vplivati na hitrost vračanja odgovorov), je bila nagrada (milijon) njegova.

R: 13

Opiši postopek, ki se bo igral igro Mobilni milijonar, ne da bi vedel odgovor na eno samo vprašanje. Na voljo imaš naslednje funkcije:

```
{ Začne novo igro in prebere prvo vprašanje. }
procedure ZacniIgro; external;

{ Naslednje funkcije vračajo podatke o trenutnem vprašanju. }
function TrenutnoVprasanje: string; external; { Vrne prazen niz, če je igre konec. }
function StMoznihOdgovorov: integer; external;
function MozniOdgovor(StOdgovora: integer): string; external;

{ Pošlje odgovor; če je pravilen, prebere naslednje vprašanje in vrne
  prazen niz, sicer je igre konec, funkcija pa vrne pravilni odgovor. }
function PosljiOdgovor(Odgovor: string): string; external;
```

2002.X.2 Pretvarjanje znakov Unicode

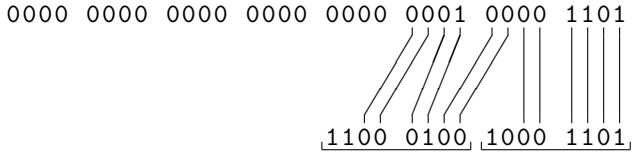
R: 15 Računalniki že desetletja uporabljajo osembitne znake za prikaz črk, kar omogoča prikaz 256 znakov. To pa je seveda premalo, da bi lahko ponazorili vse znake vseh obstoječih svetovnih jezikov. Problem rešuje standard Unicode, ki določa nabor 32-bitnih znakov. Za lažje prikazovanje znakov Unicode na računalnikih pa se vedno pogosteje uporablja pretvorba iz 32-bitnega nabora Unicode v niz enega ali več 8-bitnih znakov, imenovana UTF-8. Pretvorba UTF-8 deluje različno na različnih območjih 32-bitnega nabora znakov Unicode:

Območje Unicode (šestnajstiško)	Preslikava bitov (dvojiško)
00000000–0000007F	0xxxxxxx
00000080–000007FF	110xxxxx 10xxxxxx
00000800–0000FFFF	1110xxxx 10xxxxxx 10xxxxxx
00010000–001FFFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx
00200000–03FFFFFF	111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
04000000–7FFFFFFF	1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx
80000000–FFFFFFFF	pretvorba ni definirana

Oznaka „x“ pomeni en bit iz znaka Unicode; vedno se uporabi le potrebne spodnje bite.

Če je vrednost znaka med 0 in 127, se izpiše le en byte, ki vsebuje vseh spodnjih sedem bitov originalnega 32-bitnega znaka. V naslednjem območju (128 do 2047) se najprej izpiše byte, ki ima vedno nastavljene zgornje tri bite na 110, njegovih spodnjih pet bitov pa so biti 10 do 6 prvotnega znaka; v naslednjem bytu pa sta zgornja dva bita enaka 10, spodaj pa so še preostali biti prvotnega znaka (od 5 do 0).

Primer: črka „č“ ima po Unicode kodo 269 (šestnajstiško 0000010D) in se torej nahaja v drugem območju po zgornji tabeli. Preslika se v dva 8-bitna znaka:



Znak se torej preslika v zaporedje dveh bytov 11000100 10001101 (dvojiško) oziroma znaka s kodama C4 in 8D (šestnajstiško; kar je 196 in 141 v desetiškem zapisu).

Napiši podprogram UTF8, ki dobi za parameter 32-bitno število (lahko je kar tipa integer oz. **int**), ki predstavlja poljuben znak iz nabora Unicode, ter

izpiše zaporedje potrebnih osembitnih znakov glede na zgornjo pretvorbeno tabelo (če dobi znak iz območja 80000000–FFFFFFFF, naj ne izpiše ničesar). Za izpis imaš na voljo podprogram PutChar, ki izpiše en sam osembitni znak.

```
procedure PutChar(C: char); external;
extern int putchar(int c);
```

2003.X.1 Ražnjič

Aci in Buba sta šla na gasilsko veselico, na kateri je bilo obilo bučne zabave, prodajali pa so tudi slastne ražnjiče. Veselica ju je pošteno zlakotila in zato jima je mama kupila en ražnjič, ki si ga bosta razdelila. Ražnjič je lesena palčka, na kateri so nabodeni kosi mesa in zelenjave. R: 17

Aci, ki bi rad zrastel v močnega fanta, ne mara drugega kot meso, Buba pa je vegetarijanka in jé le zelenjavo.

Zastavlja se jima vprašanje, kje naj razlomita paličico ražnjiča, tako da bo vsak od njiju dobil svoj konec, obenem pa hoče na njem čim več hrane, ki jo ima rad, in čim manj hrane, ki je ne mara. Aci in Buba sta pridna, zato vedno vse pojesta, prav tako pa bi bilo neolikano, če bi si med seboj menjala kose hrane.

Tako bo Aci s svojega kosa ražnjiča pojedel vse meso in se pošteno potrudil, da bo zmogel tudi zelenjavo. Buba bo z veseljem pojedla vso zelenjavo, le mesa, ki ga ne mara, bi rada pojedla čim manj.

Pomagaj Aciju in Bubi ter napiši program, ki jima bo povedal, kje naj razdelita ražnjič, tako da bosta oba skupaj dobila čim večjo količino (težo) hrane, ki jo imata rada, obenem pa bosta morala pojesti čim manj hrane, ki je ne marata.

Napiši program, ki najde mesto razreza ražnjiča, pri katerem bo vrednost izraza

$$\mathit{Meso}(\mathit{Aci}) - \mathit{Zelenjava}(\mathit{Aci}) + \mathit{Zelenjava}(\mathit{Buba}) - \mathit{Meso}(\mathit{Buba})$$

čim večja. $\mathit{Meso}(x)$ pomeni količino mesa, $\mathit{Zelenjava}(x)$ pa količino zelenjave na x-ovem koncu ražnjiča.

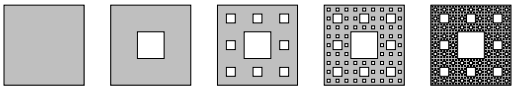
Na voljo imaš tabelo:

```
var Raznjic: array [0..N − 1] of integer;
int Raznjic[N];
```

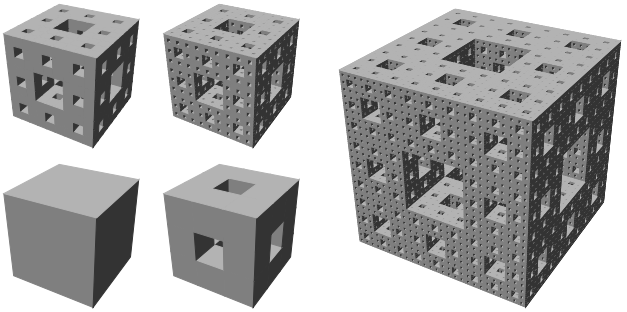
ki opisuje ražnjič, na katerem je nanizanih N kosov hrane. Če je Raznjic[i] $>$ 0, to pomeni, da je na i-tem mestu na ražnjiču kos mesa s težo Raznjic[i]. Če pa je Raznjic[i] $<$ 0, to pomeni, da je na i-tem mestu na ražnjiču kos zelenjave s težo $-$Raznjic[i].

2003.X.2 Kocka Sierpińskega

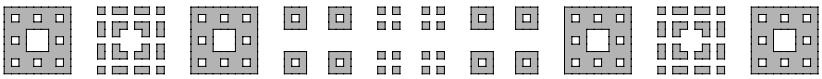
R: 18 Kvadrat Sierpińskega dobimo, če kvadrat v mislih razdelimo na 9 enakih delov in srednjega izrežemo. Vsakega izmed preostalih osmih kvadratov zopet razdelimo na 9 enakih delov in srednjega izrežemo, ... ter tako do neskončnosti. No, če smo iskreni, po neskončno korakih od začetnega kvadrata ne ostane kaj prida[1] in bi nam tudi najboljša lupa ne pomagala preveč, zato nas bodo zanimali le taki kvadrati, ki jih lahko dobimo po končno mnogo korakih. Primeri na sliki kažejo kvadrate, ki jih dobimo po nič, enem, dveh, treh in štirih korakih.



Podobno kot s kvadratom lahko naredimo tudi s kocko. Razdelimo jo na 27 kockic in zavržemo tiste, ki se ne dotikajo nobene od stranic prvotne kocke. Ostane nam dvajset od prvotnih 27 kockic in zdaj lahko na enak način obdelamo vsako od njih. Telo, ki se mu s ponavljanjem te operacije postopoma približujemo, se imenuje „kocka Sierpińskega“ ali tudi „Mengerjeva spužva“. Naslednja slika prikazuje telesa, ki jih dobimo po prvih nekaj korakih. Označimo s K_n telo, ki ga dobimo po n korakih.



Če želimo kocko, dobljeno po n korakih, narisati v dveh dimenzijah, jo lahko razrežemo na 3^n rezin in narišemo vsako posebej. Spodaj je primer razreza za kocko, kjer je $n = 2$. Opaziš lahko, da se vzorec s posamezne rezine pojavi večkrat. Tako so si prva, tretja, sedma in deveta rezina enake, prav tako druga in osma ter četrta in šesta rezina. Peta rezina ni enaka nobeni drugi.



[1] Ostane pravzaprav kar veliko točk (neštevno mnogo), le zelo so razpršene, tako da ima ta „lik“ (če lahko takšni množici točk rečemo lik) ploščino 0.

Napiši program, ki iz datoteke `kocka.in` prebere števili n ($0 \le n \le 7$) in r ($1 \le r \le 3^n$) in v datoteko `kocka.out` izpiše, pri koliko rezinah kocke K_n se pojavi isti vzorec kot pri r-ti rezini, nato pa še nariše r-to rezino kocke K_n. Prazna polja nariši s piko („`.`“), polna pa z zvezdico („`*`“).[2]

Primer vhodne datoteke:

```
2 4
```

Pripadajoča izhodna datoteka:

```
Vzorec 4. rezine se pojavi 2-krat.
***...***
*.*...*.*
***...***
.........
.........
.........
***...***
*.*...*.*
***...***
```

2003.X.3 Ugibanje nizov

Dva igralca se igrata naslednjo igro: prvi si je zamislil nek niz znakov (recimo mu s) in povedal drugemu le dolžino tega niza, drugi igralec pa bi zdaj rad ta niz uganil. Drugi igralec ugiba tako, da predlaga nek niz t, prvi igralec pa mu nato pove, na koliko mestih se istoležna znaka nizov s in t ujemata ter koliko izmed preostalih znakov t-ja se pojavlja na preostalih mestih s-ja. Na primer: če je $s =$ `abcabde` in $t =$ `ebaabda`, se niza ujemata na štirih mestih (`.b.abd.`), od ostalih znakov t-ja (to so znaki `e.a...a`) pa se na ostalih mestih s-ja pojavljata dva (namreč `e` in eden od `a`-jev). R: 22

Napiši podprogram, ki bo pomagal prvemu igralcu primerjati niza. Podprogram naj kot parametra dobi niza s in t in izpiše, na koliko mestih se ujemata in koliko izmed ostalih črk t-je se pojavlja na ostalih mestih t-ja. Predpostaviš lahko, da so nizi sestavljeni le iz malih črk angleške abecede (od `a` do `z`).

2003.X.4 Palindromi

Nek niz je *palindrom*, če se od konca proti začetku bere enako kot od začetka proti koncu (primeri: „radar“, „vodovodov“, „omejujemo“, pa v angleščini npr. “deified”, “reviver”, “rotator”). Znotraj nekega danega niza bi radi poiskali podnize, ki so palindromi. (Zanimali nas bodo le strnjeni podnizi, torej taki, pri katerih si črke sledijo neposredno druga za drugo.) Zanima nas taka skupina palindromnih podnizov, ki se ne prekrivajo, skupaj pa pokrijejo pa čim več črk prvotnega niza. Pri tem pa nočemo uporabljati palindromnih podnizov dolžine R: 23

[2] Zanimivo vprašanje je tudi, iz koliko ločenih koščkov je sestavljena r-ta rezina. Iz slike za kocko K_2 na str. 4 vidimo, da so za rezine kocke K_2 odgovori po vrsti takšni: 1, 16, 1, 4, 16, 4, 1, 16, 1.

1, saj bi z njimi lahko trivialno pokrili celoten niz. **Opiši postopek**, ki za dani niz poišče najboljšo takšno skupino palindromnih podnizov.

Primer: pri nizu *abcbaedeabc* bi lahko pokrili osem črk z dvema palindromoma, npr. *abcba* in *ede* ali pa *bcb* in *aedea*, še boljšo rešitev pa dobimo, če uporabimo zgolj palindrom *cbaedeabc*, ki pokrije kar devet črk.

2003.X.5 Seštevanje ulomkov

R: 25 **Napiši podprogram**, ki za dana cela števila a, b, c, d izračuna vsoto

$$\frac{a}{b} + \frac{c}{d}$$

in jo zapiše kot okrajšan ulomek.

2003.X.6 Urejanje logičnih vrednosti

R: 26 Imamo tabelo logičnih vrednosti:

```
const N = ...;
var Tabela: array [1..N] of boolean;

#define n ...
bool tabela[n];
```

Napiši podprogram, ki to tabelo čim hitreje uredi. Vrstni red je pri tipu boolean tak, da je vrednost false manjša od vrednosti true.

2003.X.7 Stikala

R: 28 V uporabniškem vmesniku nekega programa nastopa skupina n stikal (*checkboxes*), pri čemer jih sme biti naenkrat odkljukanih največ m. **Napiši podprogram**, ki ga bo sistem klical ob vsakem kliku na kakšno stikalo. Tvoj podprogram naj poskrbi, da ne bo nikoli odkljukanih več kot m stikal. Na voljo so podprogrami:

```
function StStikal: integer; external;
function MaxHkratiOdkljukanih: integer; external;
function JeOdkljukano(StStikala: integer): boolean; external;
procedure PostaviStikalo(StStikala: integer; Odkljukano: boolean); external;
```

Tvoj podprogram naj bo oblike:

```
procedure KlicanObSpremembi(StStikala: integer);
```

Parameter StStikala pove, na katero stikalo je uporabnik pravkar kliknil in mu s tem spremenil stanje.

2003.X.8 Nabori znakov

Recimo, da imamo nek niz znakov, v katerem je vsak znak predstavljen z nekim 32-bitnim celim številom (iz nekega velikega nabora znakov, na primer Unicode). Recimo tudi, da naš računalnik premore za izpisovanje nizov le pisave, ki ne podpirajo vseh možnih znakov, pač pa vsaka le neko podmnožico vseh možnih znakov. Zato se lahko zgodi, da našega niza ne moremo v celoti prikazati z eno pisavo, pač pa ga bo treba prikazati po koščkih in vmes preklapljati med pisavami. **Opiši postopek**, ki za dani niz in podatke o pisavah, ki so na voljo (za vsako pisavo poznamo številke vseh znakov, ki jih ta pisava vsebuje), ugotovi, kako ga je treba razdeliti na strnjene podnize, tako da bo teh podnizov čim manj in da se bo dalo vsak podniz v celoti izpisati z eno samo pisavo.[3] Opiši tudi podatkovne strukture, ki bi jih tvoj postopek uporabljal pri svojem delu. Predpostaviš lahko, da so številke znakov med 0 in 10^6 in da za vsak znak obstaja vsaj ena pisava, s katero ga lahko izpišemo. R: 29

2003.X.9 Hiperkocka in mreža

Graf je struktura, sestavljena iz množice točk in množice povezav. Vsaka povezava povezuje dve točki. Pri tej nalogi so povezave vedno neusmerjene. R: 30

Hiperkocka H_n je graf, ki ima 2^n točk, označenih z n-bitnimi celimi števili od 0 do $2^n - 1$. Povezava med točkama u in v obstaja natanko v primeru, ko dvojiški zapis števil u in v razlikuje v natanko enem bitu.

Mreža $M_{w,h}$ je graf, ki ima $w \cdot h$ točk, označenih s pari celih števil (x, y), $0 \le x < w$, $0 \le y < h$. Povezava med (x, y) in (x', y') obstaja natanko v primeru, če se točki v eni od koordinat ujemata, v drugi pa se razlikujeta za natanko 1.

Če v hiperkocki H_n spremenimo oznake točk in pobrišemo nekaj povezav, lahko iz nje dobimo mrežo $M_{2^m,2^{n-m}}$. **Opiši postopek**, ki za dani celi števili n in m, $0 \le m \le n$, pove, katere povezave hiperkocke H_n je treba pobrisati in kako je treba preimenovati ostale točke, da nam ostane ravno mreža $M_{2^m,2^{n-m}}$.

2004.X.1 Graf signala

Program za digitalno obdelavo signalov obdeluje tabelo vrednosti signala skozi čas: R: 33

```
const StVrednosti = ...;
var Vrednosti: array [0..StVrednosti − 1] of integer;
```

[3] Malo drugačen problem pa dobimo, če se vprašamo, kako izpisati niz, tako da bomo pri tem porabili čim manj različnih pisav (četudi bo treba zato niz mogoče razsekati v večje število podnizov).

Tipično število vrednosti je nekaj sto tisoč. Program mora izrisovati tudi graf signala, česar ni smiselno početi bolj natančno, kot je ločljivost zaslona — približek signala sestavimo iz daljic, katerih začetna in končna točka se po x-osi razlikujeta za najmanj 1. (Recimo, da nas ne moti, če zaradi izpuščanja vrednosti na grafu manjkajo kakšni zanimivi pojavi, „špice“ ali kaj podobnega.) **Napiši del programa**, ki izriše približek signala. Poleg konstante StVrednosti in tabele Vrednosti uporabi še naslednje deklaracije:

```
const ZaslonX = ...;
  { Vodoravna ločljivost zaslona; lahko je manj ali več od StVrednosti. }

procedure NarisiDaljico(x1, y1, x2, y2: integer); external;
  { Nariše daljico od točke (x1, y1) do (x2, y2). }
```

Predpostaviš lahko, da se vrednosti gibljejo med 0 in navpično ločljivostjo zaslona, zato po y-osi signala ni treba premikati ali raztegovati.

Še deklaracije v C/C++:

```
#define StVrednosti ...
#define ZaslonX ...
int Vrednosti[StVrednosti];
extern void NarisiDaljico(int x1, int y1, int x2, int y2);
```

2004.X.2 Ribe

R: 34 Imamo zaporedje n rib. Za vsako ribo poznamo njeno velikost; število a_i je velikost i-te ribe ($i = 1, \ldots, n$). Izbrati si smemo eno od rib in nato zaporedoma početi naslednje: če sta obe sosedi trenutne ribe manjši od nje, se ustavimo, sicer pa ena od večjih sosed požre trenutno ribo (če sta obe sosedi večji, lahko sami izberemo, katera) in sama postane trenutna. Na koncu seštejemo velikosti vseh požrtih rib in vsoti prištejemo še velikost ribe, pri kateri se je postopek ustavil. **Opiši postopek**, ki ugotovi, kakšna je največja vsota, ki jo lahko na ta način dobimo. Če riba požre neko drugo, se ji velikost pri tem nič ne spremeni.

2004.X.3 Cezarjev kod

R: 38 Težava varne izmenjave podatkov ni le problem moderne civilizacije, temveč se z njo človeštvo sooča že od nekdaj. Pred dvema tisočletjema je Julij Cezar svojim vojskovodjem pošiljal sporočila, kodirana s preprosto zamenjavo, ki ji danes pravimo Cezarjev kod. Deluje tako, da vsako črko zamenjamo s črko, ki je v abecedi nekaj mest za njo. „Ključ“ tega kodiranja je torej število mest. Pri ključu 3 ($k = 3$) bi tako denimo črko `d`, ki se v angleški abecedi s 26 znaki

($n = 26$) nahaja na četrtem ($a = 4$) mestu, zamenjali z `g`, ki je sedma črka abecede: $a + k = 3 + 4 = 7$. Če kodiramo eno izmed zadnjih k črk abecede, je seveda ne moremo zamenjati s črko, ki je v abecedi k mest za njo, saj take črke sploh ni; zato v tem primeru jemljemo črke spet z začetka abecede. Če je torej $a + k > n$, bomo a zakodirali v črko $a + k - n$.

Če vemo, da je neko besedilo zakodirano s Cezarjevo šifro, ga lahko zelo enostavno dekodiramo, saj obstaja le $n - 1 = 25$ možnih vrednosti ključa k. **Napiši podprogram**, ki na osnovi znanega (podanega) dela besedila razbere in vrne ključ, po katerem je kodirano celotno sporočilo. Sporočilo sestavljajo le velike črke angleške abecede brez presledkov (`ABCDEFGHIJKLMNOPQRSTUVWXYZ`).

Tvoj podprogram naj bo takšne oblike:

```
function Dekodiraj(Kodirano, Znano: string): integer;
```

ali, v C/C++:

```
int Dekodiraj(char* Kodirano, char* Znano);
```

Kot vhod dobi dva niza: kodirano sporočilo (Kodirano), in nek strnjen podniz nekodiranega sporočila (Znano). Vrne naj vrednost ključa k. Lahko se zgodi, da je podniz nekodiranega sporočila izbran tako nesrečno, da je možnih več ključev k; v tem primeru lahko vrneš poljubnega med njimi.

Če hočeš, lahko predpostaviš, da imaš na voljo naslednji dve funkciji za pretvarjanje črk v zaporedne številke in nazaj:

```
const n = ...;

function StevilkaCrke(Crka: char): integer;
  { Črka mora biti ena od 'A', ..., 'Z'.
    Funkcija vrne njen položaj v abecedi (število med vključno 1 in n). }

function CrkaIzStevilke(Stevilka: integer): char;
  { Številka mora biti med vključno 1 in n. Vrne ustrezno izmed črk 'A', ..., 'Z'. }
```

Primer: če je nekodirano sporočilo

`DOBRODOSLINATEKMOVANJUIZZNANJARACUNALNISTVA`

in ga kodiramo po ključu $k = 3$, se črke preslikajo v skladu z naslednjo tabelo:

iz	`ABCDEFGHIJKLMNOPQRSTUVWXYZ`
v	`DEFGHIJKLMNOPQRSTUVWXYZABC`

Kodirano sporočilo bi v tem primeru bilo:

`GREURGRVOLQDWHNPRYDQMXLCCQDQMDUDFXQDOQLVWYD`

Če bi podprogram Dekodiraj dobil to sporočilo kot prvi parameter, kot drugega pa na primer niz `VANJUIZZ` (torej nek primeren podniz nekodiranega sporočila), bi moral vrniti vrednost 3.

2004.X.4 Števila zveri

R: 40 V spletni enciklopediji celoštevilskih zaporedij najdemo pod geslom A051003 naslednje zaporedje števil, ki vsebujejo (v desetiškem zapisu) 666 kot podniz:

$$666, 1666, 2666, 3666, 4666, 5666, 6660, 6661, 6662, 6663,$$
$$6664, 6665, 6666, 6667, 6668, 6669, 7666, 8666, 9666, 10666, \ldots$$

Recimo, da bi nas namesto števila 666 zanimalo kot podniz neko drugo naravno število, recimo b. Da nam pomagaš pri računanju zaporedja števil, ki vsebujejo b, nam **opiši postopek**, ki pri danih naravnih številih a in b izračuna, katero je najmanjše tako naravno število, ki je večje ali enako a in vsebuje b kot podniz (če ju zapišemo v desetiškem zapisu). Da bo lažje, lahko predpostaviš, da je a večji ali enak b.

Primer: pri $a = 6500$ in $b = 666$ je prvo primerno število 6666, pri $a = 5436934$ in $b = 705$ pa 5437050.

Tvoj postopek naj bo **čim bolj učinkovit**. Če se le da, naj deluje dovolj hitro, da bo uporaben tudi v primerih, ko sta števili a in b zelo veliki (recimo, da ima b v desetiškem zapisu več sto števk, a pa več tisoč števk). Opiši tudi, kako bi takšna števila predstavil v računalniku, ni pa ti treba pisati celotne implementacije v kakšnem konkretnem programskem jeziku.

2004.X.5 Zamenjave

R: 49 Ko napišemo predlogo besedila, je potrebno posamezne dele prilagoditi ciljnemu uporabniku. V predlogi označimo mesta, kamor naj se vpišejo ime, priimek, naslov itd. Včasih pa to ni dovolj in potrebne so bolj zakomplicirane zamenjave.

Napiši program, ki bo zamenjal gesla v besedilu z novimi vrednostmi in pri tem upošteval dano tabelo zamenjav.

Na začetku *vhodne datoteke* program dobi v vsaki vrstici po en par ⟨*geslo, nova vrednost*⟩ (vmes je presledek; geslo torej nikoli ne vsebuje presledka). Nato sledi prazna vrstica in pa besedilo. Gesla se v tabeli ne ponavljajo.

V besedilu se lahko pojavljajo nizi oblike `$(geslo)`, katere moraš zamenjati z novimi vrednostmi. Ti nizi so lahko tudi gnezdeni, na primer `$(geslo$(drugo_geslo))`; nikoli pa se ne raztezajo čez več vrstic. Če se oblika `$(nekaj)` pojavi v ravnokar obdelanem besedilu (po neki zamenjavi), je ne smeš zamenjati, saj bi take zamenjave lahko vodile v neskončno zanko. Če določenega gesla ni v tabeli, ne naredi pri njem nobene spremembe. Če sumiš, da je v besedilu napaka (manjka zaklepaj), ga prav tako pusti nespremenjenega.

Vsakič, ko program naleti na znaka `$$` v *originalnem* besedilu, naj izpiše namesto njiju le en `$`. Znak `$` ne sme voditi v novo zamenjavo (recimo v

primeru `$$(geslo)`, kjer je rezultat `$(geslo)`). Lahko pa je del gnezdene zamenjave (npr. `$(ge$$lo)`, kjer je rezultat nova vrednost pod geslom `ge$lo`). Podobno naj, če v originalnem besedilu naleti na znaka `$)`, izpiše namesto njiju le zaklepaj `)`.

V naslednjih primerih pa v originalnem besedilu ne naredi sprememb: če določenega gesla ni v tabeli (pustiš nedotaknjena tudi `$(` in `)`); če je v vrstici napaka (pri nizu `$(` manjka ustrezni zaklepaj); če znaku `$` sledi nek znak, ki ni niti `$` niti `(` niti `)`.

V *izhodno datoteko* izpiši besedilo, ki nastane po opisanih zamenjavah.

Omejitve. Dolžina gesel in novih vrednosti je manjša od 256 znakov. Število vseh zamenjav v tabeli je manjše od 1000. Vsaka vrstica besedila je krajša od 256 zankov. Globina gnezdenja gesel je največ 50.

Primer vhodne in izhodne datoteke je na str. 12.

2004.X.6 Vžigalice

Na mizi je nekaj vžigalic, razporejenih v v vrstic. V prvi vrstici je a_1 vžigalic, v drugi a_2 in tako naprej. Dva igralca izmenično vlečeta poteze. Vsaka poteza je sestavljena iz tega, da trenutni igralec izbere eno od vrstic in vzame iz nje eno ali več vžigalic. Igro izgubi tisti, ki pobere z mize zadnjo vžigalico. Pri vsakem začetnem razporedu vžigalic se izkaže, da obstaja natanko za enega od igralcev zmagovalna strategija — torej lahko igra tako, da bo zanesljivo zmagal, pa karkoli bo že počel drugi igralec. **Opiši postopek**, ki za dani začetni razpored vžigalic na mizo ugotovi, kateri igralec ima takšno zmagovalno strategijo.

R: 51

2004.X.7 Trikotniki

Raztreseni profesor Galerkin preučuje trdnost letalskih kril z metodo končnih elementov. Ta metoda lik, ki predstavlja obliko krila, razdeli na trikotnike, nato pa izvaja trdnostne izračune. Iznašel je novo, boljšo obliko krila, žal pa je izgubil opis njene razdelitve na trikotnike, brez katerega so rezultati neuporabni. Na srečo mu je ostal seznam daljic, ki so trikotnike sestavljali. Pomagaj mu s **programom**, ki bo izpisal seznam prvotnih trikotnikov. (Trikotnikov, ki so znotraj razdeljeni na manjše trikotnike, ne izpiši. Daljice so razpostavljene tako, da ne tvorijo nobenega mnogokotnika z več kot tremi oglišči, ki ne bi bil navznoter z dodatnimi daljicami razdrobljen na trikotnike. Daljice se ne sekajo in ne prekrivajo, pa tudi dotikajo se ne, razen v tem smislu, da imata lahko dve daljici skupno krajišče.)

R: 58

Daljice so podane v *vhodni datoteki*, katere prva vrstica vsebuje eno celo število — število daljic (največ 10 000), v vsaki nadaljnji vrstici pa je četverka realnih stevil x_1, y_1, x_2, y_2, ki predstavljajo daljico od točke (x_1, y_1) do (x_2, y_2). Dve števili imaš lahko za enaki, če se razlikujeta za manj kot 10^{-6}.

Vhodna datoteka („␣“ označuje enega od presledkov, ki se ga drugače ne bi videlo):

```
geslo niz
drugo_geslo drug_niz
gnezdeno_geslo drugo_
gnezdeno_geslo2␣
$(negnezdeno)geslo trik
$(gesloZoklepaji) nizGeslaZOklepaji
geslo4 $(zamenjavaZoklepaji)
geslo41 $(zamenjava
geslo42 Zoklepaji)
zamenjavaZoklepaji ne_pride_v_postev
dolar $
zaklepaj )

$(geslo) $(drugo_geslo)
$($(gnezdeno_geslo)geslo)
$($(gnezdeno_geslo2)geslo)
$($(negnezdeno)geslo)
$($(gesloZoklepaji))
$(geslo4)
$(geslo41)$(geslo42)
$(nedefinirano_geslo)
Primer napake:
  $(geslo$(gnezdeno_geslo)
Odvečen zaklepaj: $(gnezdeno_geslo))
Ubežne sekvence: $$(geslo$)
Trik brez uporabe ubežnih sekvenc:
  $(dolar)(geslo$(zaklepaj)
In z gnezdenjem:
  $($(dolar)(geslo$(zaklepaj))
  ali $($$(geslo$))
$(dolar)$(geslo) in $$(geslo$)
  ali $$$$(geslo$)
  in $(dolar)$(dolar)(geslo)
$ $a$b$c$d
```

Pripadajoča izhodna datoteka:

```
niz drug_niz
drug_niz
niz
trik
nizGeslaZOklepaji
$(zamenjavaZoklepaji)
$(zamenjavaZoklepaji)
$(nedefinirano_geslo)
Primer napake:
  $(geslodrugo_
Odvečen zaklepaj: drugo_)
Ubežne sekvence: $(geslo)
Trik brez uporabe ubežnih sekvenc:
  $(geslo)
In z gnezdenjem:
  $($(geslo))
  ali $($(geslo))
$niz in $(geslo)
  ali $$(geslo)
  in $$(geslo)
$ $a$b$c$d
```

Primer vhodne in izhodne datoteke za nalogo 2004.X.5.

Izhodna datoteka naj vsebuje v prvi vrstici število trikotnikov. V nadaljnjih vrsticah naj bo seznam trikotnikov, vsak v petih vrsticah. Trikotnik z oglišči (x_1, y_1), (x_2, y_2) in (x_3, y_3) naj bo zapisan kot:

(*prazna vrstica*)
x_1 y_1
x_2 y_2
x_3 y_3
x_1 y_1

Opis trikotnika se lahko začne s poljubnim od njegovih treh oglišč, nadalje-

vati pa se mora v pozitivni smeri (obratni od urinega kazalca). Vrstni red trikotnikov ni pomemben.

Primer dveh vhodnih datotek:

```
3
0.0 0.0 1.0 0.0
1.0 0.0 0.0 1.0
0.0 0.0 0.0 1.0
```

```
6
0.0 0.0 2.0 0.0
1.0 0.8 0.0 0.0
1.0 0.8 1.0 2.0
1.0 2.0 0.0 0.0
2.0 0.0 1.0 0.8
2.0 0.0 1.0 2.0
```

Možni pripadajoči izhodni datoteki:

```
1

0.0 0.0
1.0 0.0
0.0 1.0
0.0 0.0
```

```
3

0.0 0.0
2.0 0.0
1.0 0.8
0.0 0.0

0.0 0.0
1.0 0.8
1.0 2.0
0.0 0.0

1.0 0.8
2.0 0.0
1.0 2.0
1.0 0.8
```

REŠITVE DODATNIH NALOG

R2002.X.1 Mobilni milijonar

N: 1

Ko odgovarjamo na vprašanja, se nam počasi nabira znanje o njih. Če uspemo na neko vprašanje pravilno odgovoriti, si zapomnimo, kateri odgovor je bil pravilen, in v bodoče na to vprašanje vedno odgovarjajmo s tem odgovorom. Dokler pravilnega odgovora še ne poznamo, pa si shranjujmo vsaj odgovore, ki smo jih pri tem vprašanju že preizkusili in ugotovili, da so napačni. Pri odgovarjanju potem vedno izberimo kakšnega izmed odgovorov, za katere še nismo ugotovili, če so napačni ali ne. Spodnji program shranjuje podatke o znanih odgovorih v zapise tipa OdgovoriT in predpostavlja, da ima na voljo podprogram Poisci, ki poišče zapis za dano vprašanje (oz. vrne false, če takega zapisa še ni), in Shrani, ki shrani zapis za dano vprašanje (in pri tem povozi morebitni že obstoječi zapis). V praksi bi lahko Poisci in Shrani uporabljala tabelo parov ⟨*vprašanje*, *zapis tipa* OdgovoriT⟩, še učinkoviteje pa bi bilo

te zapise shranjevati v razpršeno tabelo. Tabelo znanih napačnih odgovorov (OdgovoriT.Napacni) bi bilo mogoče koristno preurediti v seznam (npr. če ne poznamo neke primerne zgornje meje za število možnih napačnih odgovorov, kot je MaxStNapacnih v spodnjem programu) ali pa kar v razpršeno tabelo (to bi bilo še posebej koristno v primeru, da nam lahko sistem predlaga veliko različnih napačnih odgovorov; z razpršeno tabelo bi lahko hitreje preverjali, kateri izmed trenutno ponujenih odgovorov so že znani kot napačni).

```
program MobilniMilijonar;

  procedure ZacniIgro; external;
  function TrenutnoVprasanje: string; external;
  function StMoznihOdgovorov: integer; external;
  function MozniOdgovor(StOdgovora: integer): string; external;
  function PosljiOdgovor(Odgovor: string): string; external;

  const MaxStNapacnih = ...;
  type OdgovoriT = record
    Pravilni: string;
    Napacni: array [1..MaxStNapacnih] of string;
    nNapacnih: integer;
  end; {OdgovoriT}

  function Poisci(Vprasanje: string; var Odg: OdgovoriT): boolean; external;
  procedure Shrani(Vprasanje: string; Odg: OdgovoriT); external;

var Vpr, Odg: string; Odgovori: OdgovoriT; i, j: integer; Zmaga: boolean;
begin
  Zmaga := false;
  repeat
    ZacniIgro;
    while true do begin

      Vpr := TrenutnoVprasanje;
      if Vpr = '' then begin Zmaga := true; break end;
      if not Poisci(Vpr, Odgovori) then { To vprašanje vidimo prvič. }
        begin Odgovori.Pravilni := ''; Odgovori.nNapacnih := 0 end;

      if Odgovori.Pravilni <> 0 then begin
        { Pravilni odgovor že poznamo; pošljimo ga... }
        Odg := PosljiOdgovor(Odgovori.Pravilni); Assert(Odg = '');
        continue; { ... in pojdimo na naslednje vprašanje. }
      end; {if}

      { Poiščimo kak odgovor, za katerega še ne vemo, ali je napačen. }
      i := 1;
      while i <= StMoznihOdgovorov do begin
        Odg := MozniOdgovor(i); j := 1;
        while j <= Odgovori.nNapacnih do
          if Odgovori.Napacni[j] = Odg then break else j := j + 1;
        if j > Odgovori.nNapacnih then break else i := i + 1;
```

```
      end; {while}
      Assert(i <= StMoznihOdgovorov);

      { Pošljimo ta odgovor. }
      Odg := PosljiOdgovor(MozniOdgovor(i));
      if Odg = '' then { Odgovor je bil pravilen! }
        Odgovori.Pravilni := MozniOdgovor(i)
      else begin { Odgovor je bil napačen. }
        Assert(Odgovori.nNapacnih < MaxStNapacnih);
        Odgovori.nNapacnih := Odgovori.nNapacnih + 1;
        Odgovori.Napacni[Odgovori.nNapacnih] := MozniOdgovor(i);
      end; {if}
      Shrani(Vpr, Odgovori);

      if Odg <> '' then break; { Napačen odgovor, konec igre. }
    end; {while}
  until Zmaga;
end. {MobilniMilijonar}
```

R2002.X.2 Pretvarjanje znakov Unicode

N: 2

Z nekaj stavki **if** lahko ugotovimo, v katero območje spada naše vhodno število c, in nato vsako območje obravnavamo posebej ter skonstruiramo ustrezno zaporedje bytov, ki ga bo treba izpisati. Da izpulimo iz c-ja primerne skupine bitov, uporabimo operatorja **shr** (zamikanje desno) in **and** (logični „in“ nad istoležnimi biti). Z operatorjem **or** (logični „ali“ nad istoležnimi biti) poskrbimo še za prižiganje enic na mestih, ki jih zahteva standard za pretvorbo. Zaradi berljivosti je v spodnjem podprogramu več celoštevilskih konstant zapisanih v šestnajstiškem sistemu; spoznamo jih po znaku $ na začetku. Takšnih šestnajstiških konstant standardni pascal sicer ne predvideva, podpirajo pa jih mnogi prevajalniki. (Nestandardna, vendar v praksi široko podprta sta tudi operator **shr** in raba **and** in **or** nad celimi števili.)

```
procedure UTF8_1(c: integer);
begin
  if c < $80 then
    PutChar(Chr(c))
  end else if c < $800 then begin
    PutChar(Chr($C0 or (c shr 6)));
    PutChar(Chr($80 or (c and $3F)));
  end else if c < $10000 then begin
    PutChar(Chr($E0 or (c shr 12)));
    PutChar(Chr($80 or ((c shr 6) and $3F)));
    PutChar(Chr($80 or (c and $3F)));
  end else if c < $200000 then begin
    PutChar(Chr($F0 or (c shr 18)));
    PutChar(Chr($80 or ((c shr 12) and $3F)));
```

```
      PutChar(Chr($80 or ((c shr 6) and $3F)));
      PutChar(Chr($80 or (c and $3F)));
    end else if c < $4000000 then begin
      PutChar(Chr($F8 or (c shr 24)));
      PutChar(Chr($80 or ((c shr 18) and $3F)));
      PutChar(Chr($80 or ((c shr 12) and $3F)));
      PutChar(Chr($80 or ((c shr 6) and $3F)));
      PutChar(Chr($80 or (c and $3F)));
    end else begin
      PutChar(Chr($FC or (c shr 30)));
      PutChar(Chr($80 or ((c shr 24) and $3F)));
      PutChar(Chr($80 or ((c shr 18) and $3F)));
      PutChar(Chr($80 or ((c shr 12) and $3F)));
      PutChar(Chr($80 or ((c shr 6) and $3F)));
      PutChar(Chr($80 or (c and $3F)));
    end; {if}
end; {UTF8_1}
```

Če hoteli preveriti še, da c ni z intervala 80000000–FFFFFFFF (kjer pretvorba ni definirana), bi bilo pametno to storiti že na začetku podprograma. Drugače bi namreč lahko (če pomeni našemu prevajalniku tip integer predznačena cela števila) takšna števila naš podprogram videl kot negativna, pogoj **if** c < $80 bi bil izpolnjen in izpisal bi se nek nesmiseln znak. Neveljavne c-je lahko polovimo s takšnim pogojem na začetku podprograma:

```
  if (c and not $7FFFFFFF) <> 0 then SporociNapako;
```

Eden od razlogov, zakaj leta 2002 te naloge nismo uvrstili na tekmovanje, je bilo prav dejstvo, da ima očitno, puščobno in prav nič domiselno rešitev, ki jo vidimo zgoraj. Lahko pa smo pri reševanju te naloge tudi malo bolj ustvarjalni. Števila, manjša od 128, obravnavajmo kot poseben primer, ostala števila pa čisto sistematično. Če ugotovimo, kateri je najvišji prižgani bit števila c, lahko izračunamo, koliko bytov bo treba izpisati (recimo k). V prvem bytu mora biti prižganih najvišjih k bitov, v ostalih pa le najvišji bit; prvi byte vsebuje najvišjih nekaj bitov števila c, vsak od ostalih bytov pa po šest bitov c-ja.

```
procedure UTF8_2(c: integer);
var n, cc, m, k: integer;
begin
  if c < 128 then begin PutChar(chr(c)); exit end;
  { n naj bo indeks najvišjega prižganega bita (0..31) v c-ju. }
  cc := c; n := 0;
  while cc <> 1 do begin n := n + 1; cc := cc shr 1 end;
  if n > 30 then begin WriteLn('Napaka!'); exit end;
  { Koliko bytov bo treba izpisati? }
  k := (n + 4) div 5;
```

```
  { V prvem izpisanem bytu mora biti prižganih vrhnjih k bitov. }
  PutChar(Chr((((1 shl k) − 1) shl (8 − k)) or (c shr (6 * (k − 1)))));
  { V vsakem nadaljnjem bytu izpišemo po 6 bitov c-ja. }
  while k > 1 do begin
    k := k − 1;
    PutChar(Chr(128 or ((c shr (6 * (k − 1))) and 63)));
  end; {while}
end; {UTF8_2}
```

Slabost te rešitve pa je, da je lahko precej počasnejša od prve; polna je zank in pogojnih stavkov, ki se izvajajo precej dlje kot preproste bitne operacije v prvi rešitvi. Koliko ta razlika pomeni v praksi, je sicer odvisno od tega, kako dolgo se izvaja podprogram PutChar, ki ga kličeta obe rešitvi enako mnogokrat. Pri naših poskusih, ko je PutChar le zapisoval znake v neko tabelo v pomnilniku, je bila druga rešitev pet- do šestkrat počasnejša od prve (razen pri znakih od 0 do 127, kjer sta obe praktično enako hitri).

R2003.X.1 Ražnjič

Naj bo m_A (oz. z_A) količina mesa (oz. zelenjave), ki jo pri določeni razdelitvi ražnjiča poje Aci, m_B (oz. z_B) pa enako za Bubo. Naloga pravi, da moramo maksimizirati $m_A - z_A + z_B - m_B$. N: 3

Naj bo m skupna teža vseh kosov mesa, z pa vseh kosov zelenjave. Potem je $m_B = m - m_A$ in $z_B = z - z_A$; če vstavimo to v izraz, ki ga maksimiziramo, dobimo $m_A - z_A + z - z_A - m + m_A$, kar je $z - m + 2(m_A - z_A)$. Ker sta z in m neodvisna od tega, kako prelomimo ražnjič, bo torej za optimalno rešitev zadostovalo že maksimiziranje razlike $m_A - z_A$. V naši tabeli so predstavljeni kosi mesa s pozitivnimi, kosi zelenjave pa z negativnimi števili, zato je $m_A - z_A$ kar vsota tistih elementov naše tabele, ki predstavljajo Acijeve kose hrane. Zaradi načina delitve ražnjiča dobi Aci bodisi prvih nekaj kosov ali pa zadnjih nekaj kosov. Če označimo s s_i vsoto prvih i števil v tabeli, s t_i pa vsoto zadnjih i števil, vidimo, da moramo poiskati vrednost $\max\{s_0, \ldots, s_n, t_0, \ldots, t_n\}$. Ker je vsota vseh števil v tabeli enaka $m - z$, je $t_i = m - z - s_{n-i}$. Naš maksimum je zato naprej enak

$$\begin{aligned} & \max\{\max\{s_0, \ldots, s_n\}, \max\{m - z - s_n, \ldots, m - z - s_0\}\} \\ = \; & \max\{\max\{s_0, \ldots, s_n\}, m - z + \max\{-s_n, \ldots, -s_0\}\} \\ = \; & \max\{\max\{s_0, \ldots, s_n\}, m - z - \min\{s_0, \ldots, s_n\}\}. \end{aligned}$$

Ko se sprehajamo po tabeli in računamo vsote $s_0, \ldots, s_n$, si moramo torej zapomniti največjo in najmanjšo med njimi in na koncu izračunati maksimum po dobljeni formuli.

```
procedure Razdeli;
var Vsota, MinVsota, MaxVsota, MinKje, MaxKje, i: integer;
begin
  MinVsota := 0; MinKje := 0; MaxVsota := 0; MaxKje := 0; Vsota := 0;
  for i := 1 to N do begin
    Vsota := Vsota + Raznjic[i − 1];
    if Vsota < MinVsota then begin MinVsota := Vsota; MinKje := i end;
    if Vsota > MaxVsota then begin MaxVsota := Vsota; MaxKje := i end;
  end; {for i}
  if MaxVsota > Vsota − MinVsota
  then WriteLn('Aci naj dobi levih ', MaxKje, ' kosov,',
               'Buba pa desnih ', N − MaxKje, '.')
  else WriteLn('Aci naj dobi desnih ', N − MinKje, ' kosov,',
               'Buba pa levih ', MinKje, '.');
end; {Razdeli}
```

R2003.X.2 Kocka Sierpińskega

N: 4 Ker delimo pri sestavljanju kocke Sierpińskega vsako kocko vedno na $3 \times 3 \times 3$ manjše kocke, je koristno pri opisovanju položaja znotraj kock uporabljati trojiški zapis števil. Kocko K_n si predstavljajmo v trodimenzionalni mreži, sestavljeni iz $3^n \times 3^n \times 3^n$ manjših kockic; nekatere od njih so v K_n res prisotne, nekatere pa ne. Položaj vsake kockice lahko opišemo s trojico koordinat (x, y, z), pri čemer so $x, y, z \in \{0, 1, \ldots, 3^n - 1\}$, tako da si lahko te koordinate predstavljamo tudi kot n-mestna števila v trojiškem zapisu.

Spomnimo se, da smo do kocke K_n prišli tako, da smo jo razdelili na 27 manjših kock, nato sedem od njih zavgrli, na ostalih dvajsetih pa nadaljevali z enakim postopkom; iz vsake od tistih dvajsetih tako nastane pravzaprav po en izvod kocke K_{n-1}. Za vsako od teh 27 kock velja, da se vse kockice v njej ujemajo v prvi števki svoje x-koordinate, prav tako pa tudi v prvi števki svoje y-koordinate ter v prvi števki svoje z-koordinate. Za posamezno kockico (x, y, z) torej ni težko ugotoviti, v kateri od 27 kock leži: pogledati moramo le prve števke trojiškega zapisa koordinat x, y in z (koordinate pri tem zapišemo vedno z n števkami, četudi mora biti zato prvih nekaj števk enakih 0).

Če bi radi za neko kockico (x, y, z) ugotovili, ali je v naši kocki K_n prisotna, lahko zdaj razmišljamo takole. Poglejmo, kateri od 27 kock, v katere je razdeljena K_n, pripada kockica (x, y, z); če je to ena od tistih sedmih kock, ki jih pri tvorbi kocke K_n v celoti zavržemo (prepoznamo jih po tem, da imata vsaj dve od treh koordinat v trojiškem zapisu prvo števko 1, ne pa 0 ali 2), vemo, da naša kockica pač ni prisotna v K_n; drugače pa preračunajmo njene koordinate relativno glede na tisto kopijo kocke K_{n-1}, v kateri smo se znašli (kar pomeni le, da jim moramo v trojiškem zapisu odbiti prvo števko), in nadaljujmo z enakim razmislekom kot doslej. Rekurzija se konča, ko pridemo do

kocke K_0, ki je pravzaprav že sama po sebi ena sama osnovna kockica in nima lukenj.

```
{ V parametru nn naj bo vrednost 3^n. }
function JePrisotna(x, y, z, nn: integer): boolean;
var i, nn1: integer;
begin
  while nn > 1 do begin
    { Koliko koordinat ima prvo številko 1? }
    i := 0; nn1 := nn div 3;
    if x div nn1 = 1 then i := i + 1;
    if y div nn1 = 1 then i := i + 1;
    if z div nn1 = 1 then i := i + 1;
    if i >= 2 then break;
    { Porežimo prvo številko vseh koordinat. }
    x := x mod nn1; y := y mod nn1; z := z mod nn1; nn := nn1;
  end; {while}
  { Če se ustavimo prej kot pri nn = 1, pomeni, da se je
    zanka končala s stavkom break, ker smo naleteli na luknjo. }
  JePrisotna := (nn = 1);
end; {JePrisotna}
```

Z besedami lahko rečemo tudi: kockica (x, y, z) je prisotna, če se nikoli ne zgodi, da bi imeli vsaj dve izmed števil x, y in z v trojiškem zapisu na istem mestu številko 1.

Zdaj ni težko narisati posamezne rezine kocke K_n:

```
procedure NarisiRezino(var T: text; z, nn: integer);
var x, y: integer;
begin
  for y := 0 to nn - 1 do begin
    for x := 0 to nn - 1 do
      if JePrisotna(x, y, z, nn)
      then Write(T, '*') else Write(T, '.');
    WriteLn(T);
  end; {for y}
end; {NarisiRezino}
```

Paziti moramo le na to, da besedilo naloge šteje rezine od 1 do 3^n, naš podprogram pa šteje z-koordinate od 0 do $3^n - 1$.

Kakšna je časovna zahtevnost tega postopka? Dovolj bo, če preštejemo število ponovitev zanke v podprogramu JePrisotna. Naj bo J_n povprečno število iteracij po vseh klicih JePrisotna za kocko K_n. Ko smo gradili kocko K_n, smo začeli s kocko s stranico 3^n in iz nje izrezali sedem kock s stranico 3^{n-1}; če pade naša trojica (x, y, z) v eno od njih, se ustavi tista zanka že po prvi iteraciji. V nasprotnem primeru pa naredimo po tisti prvi še nekaj iteracij, v povprečju

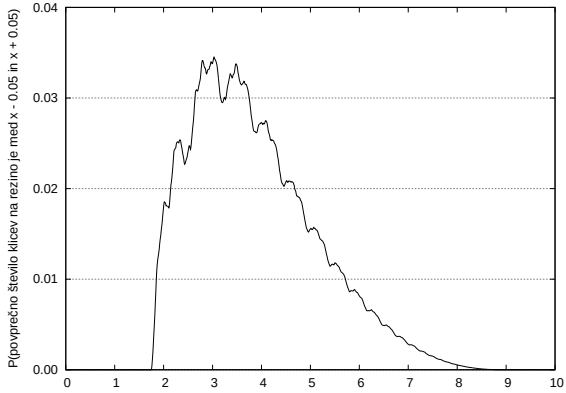
ravno J_{n-1}; tako dobimo $J_n = 1 + (20/27)J_{n-1}$. Iz te zveze lahko izpeljemo eksplicitno formulo $J_n = 27/7 \cdot (1-(20/27)^n)$, ki nam pove, da je zaporedje J_n strogo naraščajoče in konvergira k vrednosti $27/7$. Povprečno število izvajanj naše zanke (če gledamo povprečje po vseh 27^n kockicah znotraj kocke K_n) je torej vedno $\leq 27/7$, ne glede na to, kako velik n vzamemo. To je posledica dejstva, da so naše kocke K_n vse bolj redke, kolikor večje n-je gledamo. V kocki K_n je prisotnih le 20^n kockic od vseh 27^n možnih, torej le $(20/27)^n$ vseh kockic, ta vrednost pa pri naraščajočem n-ju pada proti 0. Zato se vse redkeje dogaja, da bi bilo treba izvesti veliko iteracij zanke **while**.

Podprogram NarisiRezino bo torej v povprečju porabil za vsak klic funkcije JePrisotna le konstantno mnogo časa, zato je njegova časovna zahtevnost v povprečju $O(9^n)$ (ker mora preveriti prisotnost za $3^n \cdot 3^n$ kockic). Res pa je, da se konstantni faktorji od rezine do rezine razlikujejo; če ima rezina veliko lukenj, bo potrebnih manj iteracij. Za rezino $z = 0$ (ki je pravzaprav kar navaden kvadrat Sierpińskega; na tej rezini je od 9^n možnih kockic prisotnih kar 8^n kockic) se na primer izkaže, da je povprečno število izvajanj notranje zanke pri kockicah s te rezine enako $9 \cdot (1-(8/9)^n)$. Po drugi strani je za rezino $z = (3^n - 1)/2$ (na sredi kocke, kjer je največ lukenj; prisotnih je le 4^n od 9^n možnih kockic) povprečno število izvajanj notranje zanke le $9/5 \cdot (1 - (4/9)^n)$ iteracij na kockico. Torej imamo pri rezini $z = 0$ skoraj petkrat toliko dela kot pri sredinski rezini $z = (3^n - 1)/2$ (razmerje je tem bližje 5, čim večji je n). Ostale rezine so nekje vmes; če označimo povprečno število iteracij po kockicah iz rezine z kocke K_n z $J_n(z)$, je

$$J_n(z) = \begin{cases} 1 + 4J_{n-1}(z \bmod 3^{n-1}) & : \text{če je } 3^{n-1} \leq z < 2 \cdot 3^{n-1} \\ 1 + 8J_{n-1}(z \bmod 3^{n-1}) & : \text{sicer.} \end{cases}$$

Graf na str. 21 kaže, kako so porazdeljene vrednosti $J_n(z)$ pri $n = 30$, torej pri kocki K_{30}. Vidimo lahko, da so res vse med približno 1,8 in 9, vendar so visoke vrednosti $J_n(z)$ redkejše (le malo rezin zahteva veliko iteracij), zato je povprečje res blizu $22/7 \approx 3{,}86$.

Oglejmo si zdaj še drugi del naloge: kako ugotoviti, koliko rezin je enakih neki dani rezini. Naša kocka K_n je sestavljena iz $3 \times 3 \times 3$ delov, od katerih je sedem praznih, v dvajsetih pa tičijo kopije kocke K_{n-1}. V srednji tretjini kocke K_n (torej za $3^{n-1} \leq z < 2 \cdot 3^{n-1}$) manjka pet delov (vsi razen vogalnih), v zgornji in spodnji tretjini pa le eden (osrednji). Zgornja in spodnja tretjina sta si torej povsem enaki; če je z neka rezina iz zgornje ali spodnje tretjine, je njeno število pojavitev v celi kocki dvakrat tolikšno kot samo v njeni tretjini. Za srednjo tretjino pa velja, da se nobena njena rezina ne pojavlja v kateri od ostalih dveh tretjin, saj se od rezin v teh dveh tretjinah razlikuje po razporedu praznih delov. Ostane le še vprašanje, kolikokrat se neka rezina pojavlja v svoji tretjini kocke K_n. Ker so si glede praznih delov vse rezine v neki tretjini enake, jih lahko razlikujemo le na podlagi nepraznih delov — tistih, kjer je del naše



Ta graf prikazuje (kot funkcijo števila x), pri kolikšnem deležu rezin kocke K_{30} leži povprečno število iteracij v zanki podprograma JePrisotna, če ga pokličemo za vse kockice s tiste rezine, na intervalu $[x - 0{,}05, x + 0{,}05]$. Povprečno število iteracij po rezini se giblje od približno 1,8 za zelo „prazne“ rezine (na sredini kocke) do skoraj 9 za zelo „polne“ (na površju kocke), povprečje po vseh rezinah pa je približno 22/7.

rezine pravzaprav rezina manjše kocke K_{n-1}; natančneje povedano, neprazni deli rezine z kocke K_n so enaki rezini $(z \bmod 3^{n-1})$ kocke K_{n-1}. Rezina z kocke K_n se torej v svoji tretjini te kocke pojavi prav tolikokrat, kolikorkrat se pojavi rezina $(z \bmod 3^{n-1})$ kocke K_{n-1} v celi kocki K_n.

```
function StPojavitevRezine(z, nn: integer): integer;
var s: integer;
begin
  s := 1;
  while nn > 1 do begin
    nn := nn div 3;
    { Če ni iz srednje tretjine, je število pojavitev dvakrat tolikšno. }
    if z div nn <> 1 then s := s * 2;
    z := z mod nn;
  end; {while}
  StPojavitevRezine := s;
end; {StPojavitevRezine}
```

Vidimo lahko, da ta funkcija ne dela drugega, kot da prešteje, kolikokrat se v trojiškem zapisu števila z pojavlja števka 1. Če se pojavlja k-krat, se rezina z v kocki K_n pojavlja 2^{n-k}-krat. Različnih rezin s takšnim številom pojavitev pa je $\binom{n}{k}$, kajti na toliko načinov si lahko izberemo, katere izmed n števk bodo imele vrednost 1. Ker sta zgornja in spodnja tretjina vsake kocke enaki, je oblika rezine odvisna le od tega, kje v trojiškem zapisu števila z so enice, ne pa tudi od tega, kje na ostalih mestih so ničle in kje dvojke.[4]

Zapišimo še glavni blok našega programa:

[4]Sebi podobnim množicam, kot je kocka Sierpińskega iz te naloge, pravimo „fraktali“. Za več o njih gl. npr. MathWorld *s. v.* “Fractal” ali poplavo literature, npr. B. B. Mandelbrot, *The Fractal Geometry of Nature*. Po istem kopitu kot tukaj kocko Sierpińskega bi lahko dobili še kup podobnih fraktalov, npr. če bi zgolj spremenili pravilo o tem, na koliko delov razdelimo kocko na vsakem koraku in katere od njih zavržemo. Namesto s kocko bi lahko delali tudi s tetraedrom, s štiristrano piramido, pa z liki, npr. s kvadratom

```
var i, z, n, nn: integer; T: text;
begin {MengerjevaSpuzva}
  Assign(T, 'kocka.in'); Reset(T); ReadLn(T, n, z); Close(T);
  z := z - 1;
  nn := 1; for i := 1 to n do nn := nn * 3;
  Assign(T, 'kocka.out'); Rewrite(T);
  WriteLn(T, 'Vzorec ', z + 1, '. rezine se pojavi ',
            StPojavitevRezine(z, nn), '-krat.');
  NarisiRezino(T, z, nn);
  Close(T);
end. {MengerjevaSpuzva}
```

Za konec si oglejmo še vprašanje, iz koliko ločenih kosov je sestavljena rezina z kocke K_n; označimo to z $A_n(z)$ (za $0 \le z < 3^n$). Koristno je vpeljati še pomožno količino: koliko izmed teh kosov se dotika vsakega roba rezine; recimo temu $B_n(z)$. Pokazati je mogoče, da velja naslednje (označimo $z' = z \bmod 3^{n-1}$, torej z brez prve števke v trojiškem zapisu):

$$A_0(z) = B_0(z) = 1$$

$$A_n(z) = \begin{cases} 4A_{n-1}(z'), & \text{če } 3^{n-1} \le z < 2 \cdot 3^{n-1} \\ 1, & \text{sicer, če } B_{n-1}(z') = 1 \\ 8(A_{n-1}(z') - B_{n-1}(z')), & \text{sicer} \end{cases}$$

$$B_n(z) = \begin{cases} 2B_{n-1}(z'), & \text{če } 3^{n-1} \le z < 2 \cdot 3^{n-1} \\ 3B_{n-1}(z') - 2, & \text{sicer} \end{cases}$$

Najmanjše možno število kosov je 1, kar dosežemo npr. pri $A_n(0)$ za poljuben n. Največje število kosov pri rezinah kocke K_n pa je $(44/35) \cdot 8^{n-1} + (8/5) \cdot 3^{n-1} + (8/7)$, kar dosežemo npr. pri $A_n(1)$.[5]

R2003.X.3 Ugibanje nizov

N: 5 Z zanko pojdimo po obeh nizih in primerjajmo istoležne znake; tako lahko preštejemo, na koliko mestih se niza `s` in `t` ujemata v istoležnih znakih. Za ostale znake niza `t` pa nas zanima, koliko jih najdemo med ostalimi znaki niza `s`. Torej, če se neka črka a pojavlja v nizu t recimo $\#_t(a)$-krat, v nizu

(„preproga Sierpińskega“), trikotnikom (iz njega bi lahko dobili npr. trikotnik Sierpińskega ali pa Kochovo snežinko), lahko tudi z navadno daljico (dobimo „Cantorjev prah“, ki ga vidimo tudi na robovih naše kocke Sierpińskega). Če uporabimo podobne prijeme nad krivuljami, lahko dobimo npr. razne prostor zapolnjujoče krivulje, kot so Peanova, Hilbertova, zmajeva itd.

[5] Do te formule ni težko priti na podlagi prej navedenih rekurzivnih formul. Dokaz, da res nima nobena rezina več kot toliko kosov, pa je malo bolj zamuden in ga tu ne bomo navajali. Pomagamo si lahko tako, da najprej dokažemo, da doseže pri $z = 1$ svoj maksimum vrednost $B_n(z)$, pa tudi vrednost $A_n(z) - 3B_n(z)$; iz tega pa ni težko dokazati, da doseže pri $z = 1$ svoj maksimum tudi $A_n(z)$.

s pa $\#_s(a)$-krat, in je $\#_t(a) \le \#_s(a)$, vemo, da bomo lahko v s-ju našli vse pojavitve črke a iz niza t; če pa je $\#_t(a) > \#_s(a)$, bomo lahko pokrili vse s-jeve pojavitve črke a, nekaj pojavitev a-ja v t-ju pa bo še ostalo. V vsakem primeru se torej število ujemanj poveča za $\min\{\#_s(a), \#_t(a)\}$. Vrednosti $\#_s$ in $\#_t$ hrani spodnji podprogram v tabelah HistS in HistT (imeni sta dobili po tem, da vsebujeta ravno to, kar potrebujemo, če hočemo s histogramom predstaviti pogostost posameznih črk v nizih s in t).

```
procedure Primerjaj(s, t: string);
var i, StUjemanj: integer; c: char;
    HistS, HistT: array ['a'..'z'] of integer;
begin
  if Length(s) <> Length(t) then
    begin WriteLn('Niza sta različno dolga!'); exit end;
  for c := 'a' to 'z' do
    begin HistS[c] := 0; HistT[c] := 0 end;
  StUjemanj := 0;
  for i := 1 to Length(s) do
    if s[i] = t[i] then StUjemanj := StUjemanj + 1
    else begin
      HistS[s[i]] := HistS[s[i]] + 1;
      HistT[t[i]] := HistT[t[i]] + 1;
    end; {if}
  WriteLn('Niza se ujemata na ', StUjemanj, ' mestih.');
  StUjemanj := 0;
  for c := 'a' to 'z' do
    if HistS[c] < HistT[c]
    then StUjemanj := StUjemanj + HistS[c]
    else StUjemanj := StUjemanj + HistT[c];
  WriteLn('Od ostalih črk niza t se jih ', StUjemanj,
          ' pojavlja na ostalih mestih niza s.');
end; {Primerjaj}
```

R2003.X.4 Palindromi

Recimo, da je naš vhodni niz s dolg n znakov. Označimo s $s[i..j]$ podniz, ki obsega znake od i-tega do j-tega. Naj bo $D(i)$ množica dolžin palindromnih podnizov niza s, ki se končajo pri znaku $s[i]$, torej i-tem znaku niza s. Opazimo, da sta si prvi in zadnji znak palindroma vedno enaka, vse vmes pa je tudi samo zase palindrom (na primer: v palindromu *radar* tiči palindrom *ada*). Če se torej pri znaku $s[i]$ končuje nek palindrom dolžine d, pomeni, da sta si znaka $s[i]$ in $s[i-d+1]$ enaka, znaki vmes pa tvorijo nek palindrom, ki ima torej dolžino $d-2$ in se končuje pri znaku $s[i-1]$. Očitno pa je, da velja tudi obratno: če sta si $s[i]$ in $s[i-d+1]$ enaka in znaki med njima tvorijo

N: 5

palindrom, je tudi celoten podniz $s[i-d+1..i]$ palindrom. Tako torej vidimo:

$$d \in D(i) \Longleftrightarrow (d-2) \in D(i-1) \wedge s[i-d+1] = s[i].$$

To pravilo lahko uporabimo za $d > 1$, za $d = 1$ pa tako ali tako vedno vemo, da je črka $s[i]$ sama zase palindrom in je torej $1 \in D(i)$. V množico $D(i)$ je koristno vključiti tudi število 0, ker bomo potem lahko z zgornjim pravilom opazili tudi palindrome dolžine 2 (če sta v nizu s dve zaporedni enaki črki).

Ko imamo enkrat množice $D(i)$, ni težko poiskati najboljšega razporeda palindromov. Naj bo $f(i)$ največje število znakov, ki jih lahko pokrijemo s palindromnimi podnizi niza $s[1..i]$, ne da bi se ti podnizi med seboj prekrivali. Možni kandidati za najboljši razpored so naslednji: (1) lahko ne uporabimo nobenega palindroma, ki se konča pri $s[i]$, in v tem primeru bo $f(i)$ kar enak $f(i-1)$, torej najboljšemu razporedu za niz $s[1..i-1]$; (2) lahko pa uporabimo nek palindrom dolžine d s koncem pri $s[i]$, in ker se palindromi med seboj ne smejo prekrivati, predstavljajo ostali palindromi v našem razporedu najboljše pokritje niza $s[1..i-d]$, torej $f(i-d)$. Tako vidimo:

$$f(i) = \max\{f(i-1), \max\{f(i-d)+d : d \in D(i), d > 1\}\}.$$

Vrednost funkcije $f(i)$ lahko torej računamo kar sproti, medtem ko določamo tudi množico $D(i)$.

Časovna zahtevnost našega postopka je sorazmerna s skupnim številom elementov v vseh množicah $D(i)$, torej s skupnim številom vseh palindromnih podnizov niza s; v najslabšem primeru je to $O(n^2)$. Prostorska zahtevnost pa je le $O(n)$ za tabele, v katerih hranimo funkcijo f, dve množici $D(i)$ in še eno tabelo, ki nam pove, kako smo prišli do vrednosti $f(i)$, da bomo lahko rekonstruirali najboljšo rešitev.

```
program Palindromi;
const MaxDolz = ...;
var s: string;
    { D[id] vsebuje seznam elementov množice D(i), D[1 − id] pa množice D(i − 1). }
    D: array [0..1, 1..MaxDolz + 1] of integer;
    nD: array [0..1] of integer; { nD[.] = velikost množice D[.] }
    f, fKako: array [0..MaxDolz] of integer;
    i, j, id, dd: integer;
begin
    ReadLn(s);
    id := 0; nD[id] := 0; f[0] := 0;
    for i := 1 to Length(s) do begin
      id := 1 − id; nD[id] := 2; D[id, 1] := 0; D[id, 2] := 1;
      { V tabeli D[1 − id] je množica D(i − 1). V tabeli D[id] pa
        bomo pripravili množico D(i); zaenkrat smo dodali vanjo števili 0 in 1. }
      f[i] := f[i − 1]; fKako[i] := 1; { če ne uporabimo palindroma s koncem pri s[i] }
```

```
      for j := 1 to nD[1 − id] do begin
        dd := D[1 − id, j]; { trenutni element množice D(i − 1) }
        if dd >= i − 1 then continue;
        if s[i − dd − 1] <> s[i] then continue;
        { Dodajmo dd + 2 v množico D(i). }
        nD[id] := nD[id] + 1; D[id, nD[id]] := dd + 2;
        if f[i − dd − 2] + dd + 2 > f[i] then { nova najboljša rešitev }
          begin f[i] := f[i − dd − 2] + dd + 2; fKako[i] := dd + 2 end;
      end; {for j}
    end; {for i}
    { Rekonstruirajmo rešitev. V tabelo D[0] bomo pisali indekse, na katerih
      se končajo uporabljeni polinomi. }
    nD[0] := 0; i := Length(s);
    while i > 0 do begin
      nD[0] := nD[0] + 1; D[0, nD[0]] := i;
      i := i − fKako[i];
    end; {while}
    for j := nD[0] downto 1 do begin
      i := D[0, j]; dd := fKako[i]; if dd = 1 then continue;
      { V najboljši rešitvi je uporabljen palindrom dolžine dd s koncem pri i. }
      while dd > 0 do begin dd := dd − 1; Write(S[i − dd]) end;
      Write(' ');
    end; {for j}
    WriteLn;
end. {Palindromi}
```

R2003.X.5 Računanje z ulomki

N: 6

Za začetek preverimo, če je katero od števil b in d enako 0; če je, vrednost $a/b + c/d$ pač ni definirana. Drugače pa vemo, da lahko ulomka a/b in c/d spravimo na skupni imenovalec bd in njuno vsoto zapišemo kot $(ad+bc)/(bd)$. Zdaj jo moramo le še okrajšati. Označimo števec z $u = ad + bc$, imenovalec z $v = bd$; poiskati moramo torej največji skupni delitelj števil u in v (recimo mu $\gcd(u, v)$) in ju oba deliti z njim. Eleganten način za iskanje največjega skupnega delitelja je Evklidov algoritem (glej npr. rešitev naloge 2001.1.3). Ta temelji na opažanju, da je $\gcd(u, v) = \gcd(v, u \bmod v)$, pri čemer je $u \bmod v$ ostanek pri deljenju u z v. Če to formulo uporabljamo v zanki, postajata števili u in v vse manjši, dokler ne deljenje enkrat ne izide ($u \bmod v = 0$, kar je znak, da je največji skupni delitelj kar v).

```
program Ulomki;

  function Gcd(u, v: integer): integer;
  var t: integer;
  begin
    while v > 0 do begin t := v; v := u mod v; u := t end;
```

```
    Gcd := u;
  end; {Gcd}

  procedure Krajsaj(var u, v: integer);
  var t: integer;
  begin
    t := Gcd(Abs(u), Abs(v)); u := u div t; v := v div t;
  end; {Krajsaj}

  procedure Izpisi(u, v: integer);
  begin
    if v < 0 then begin u := -u; v := -v end; { prenesimo predznak v števec }
    if v > 1 then Write(u, '/', v) else Write(u);
  end; {Izpisi}

var a, b, c, d, u, v: integer;
begin {Ulomki}
  ReadLn(a, b, c, d);
  if (b = 0) or (d = 0) then
    begin WriteLn('Deljenje z nič!'); exit end;
  Write('Vsota '); Izpisi(a, b); Write(' in '); Izpisi(c, d);
  u := a * d + b * c; v := b * d;                                    { (†) }
  Krajsaj(u, v);
  Write(' je '); Izpisi(u, v); WriteLn('.');
end. {Ulomki}
```

Slabost tega programa je, da za skupni imenovalec vedno uporabi produkt $b \cdot d$. To vsekakor je skupni imenovalec, ni pa nujno najmanjši skupni imenovalec; če sta b in d velika, bi bil lahko njun produkt prevelik za spremenljivko tipa integer, njun najmanjši skupni imenovalec pa mogoče ne. Če označimo najmanjši skupni večkratnik števil b in d z $\mathrm{lcm}(b, d)$ in se spomnimo, da je $\mathrm{lcm}(b, d) \cdot \gcd(b, d) = b \cdot d$, vidimo, da lahko skupni imenovalec $\mathrm{lcm}(b, d)$ računamo po formuli $(b/\gcd(b, d)) \cdot d$. Poleg tega lahko ulomka a/b in c/d pred računanjem še pokrajšamo, da bomo delali ves čas s čim manjšimi števili. Vrstico (†) gornjega programa bi torej lahko zamenjali z:

```
  Krajsaj(a, b); Krajsaj(c, d);
  v := (b div Gcd(b, d)) * d;
  u := a * (v div b) + c * (v div d);
```

R2003.X.6 Urejanje logičnih vrednosti

N: 6 Ena možnost je, da preprosto preštejemo, kolikokrat se pojavlja v tabeli false, in potem v prvih toliko celic vpišemo vrednost false, v ostale pa true.

```
procedure Urejanje;
var i, StFalse: integer;
```

```
begin
  StFalse := 0;
  for i := 1 to N do if not Tabela[i] then StFalse := StFalse + 1;
  for i := 1 to StFalse do Tabela[i] := false;
  for i := StFalse + 1 to n do Tabela[i] := true;
end; {Urejanje}
```

Slabost te rešitve je, da smo morali narediti dva prehoda skozi tabelo: v prvem smo šteli vrednosti false, v drugem pa smo popravljali vsebino tabele. Tabelo lahko uredimo tudi z enim samim prehodom: z enim števcem se premikamo z leve, z drugim z desne in če pride levi števec do vrednosti true, desni pa do false, ju zamenjamo. Levi števec pušča na svoji levi same vrednosti false, desni števec pa na svoji desni same vrednosti true. Ko se števca srečata, vemo, da je celotna tabela urejena. Ta postopek se zgleduje po postopku za razdeljevanje tabele pri urejanju z algoritmom quicksort.

```
procedure Urejanje2;
var i, j: integer;
begin
  i := 1; j := n;
  while i < j do begin
    { Invarianta: Tabela[1..i − 1] = false, Tabela[j + 1..n] = true. }
    while (i < j) and not Tabela[i] do i := i + 1;
    while (i < j) and Tabela[j] do j := j − 1;
    if i < j then begin
      Tabela[i] := false; Tabela[j] := true;
      i := i + 1; j := j − 1
    end; {if}
  end; {while}
end; {Urejanje2}
```

Še en način, kako urediti celotno tabelo v enem samem prehodu, pa je ta, da pred seboj „odrivamo“ vse vrednosti true, ki smo jih dotlej že našli, ko pa pridemo mimo kakšne vrednosti false, jo premaknemo nazaj za vse doslej najdene vrednosti true.[6]

```
procedure Urejanje3;
var i, j: integer;
begin
  i := 1; while i <= n do if Tabela[i] then break else i := i + 1;
  j := i; i := i + 1;
  while i <= n do begin
    { Invarianta: Tabela[1..j − 1] = false, Tabela[j..i − 1] = true. }
    if not Tabela[i] then
```

[6] Gl. npr. Wikipedia *s. v.* Several Unique Sort.

```
      begin Tabela[j] := false; Tabela[i] := true; j := j + 1 end;
    i := i + 1;
  end; {while}
end; {Urejanje3}
```

Za poskus smo izmerili čas, potreben za urejanje tabele 10^8 elementov. Naslednja tabela kaže povprečne čase po desetih urejanjih:

Začetno stanje tabele	Urejanje	Urejanje2	Urejanje3
tabela naključnih vrednosti	2,00 s	1,48 s	1,74 s
padajoče urejena tabela	1,52 s	1,31 s	1,26 s
naraščajoče urejena tabela	1,52 s	0,67 s	0,73 s
same vrednosti true	1,49 s	0,67 s	0,75 s
same vrednosti false	1,55 s	0,67 s	0,73 s

V praksi je mogoče še najzanimivejši primer tabela z naključnimi podatki, na kateri je, kot vidimo, najhitrejša rešitev Urejanje2, ki je približno 25 % hitrejša od Urejanje; Urejanje3 pa je približno na pol poti med njima.

R2003.X.7 Stikala

N: 6 Naš podprogram po vrsti pregleduje stikala in pri tem šteje, koliko je odkljukanih. Če to število preseže največje dovoljeno število hkrati odkljukanih stikal, bomo eno od odkljukanih stikal izključili. Naloga nič ne določa, katero naj izključimo, zato naj bo to kar zadnje odkljukano stikalo, na katero pri pregledovanju stikal naletimo. Pazimo le na to, da ne bomo izključili tistega stikala, ki ga je uporabnik ravnokar vključil, ampak raje kakšno drugo (razen če je MaxHkratiOdkljukanih enako 0).

```
procedure KlicanObSpremembi(StStikala: integer);
var i, StOdkljukanih, NekoOdkljukano: integer;
begin
  { Če je uporabnik stikalo izključil, zdaj gotovo ni vključenih preveč stikal. }
  if not JeOdkljukano(StStikala) then exit;
  { Preštejmo odkljukana stikala. }
  StOdkljukanih := 0; NekoOdkljukano := StStikala;
  for i := 1 to StStikal do if JeOdkljukano(i) then begin
    if i <> StStikala then NekoOdkljukano := i;
    StOdkljukanih := StOdkljukanih + 1;
    if StOdkljukanih > MaxHkratiOdkljukanih then break;
  end; {for i, if}
  { Izključimo kakšno stikalo, če je to potrebno. }
  if StOdkljukanih > MaxHkratiOdkljukanih then
    PostaviStikalo(NekoOdkljukano, false);
end; {KlicanObSpremembi}
```

R2003.X.8 **Nabori znakov**

Če je vsak znak predstavljen z nekim celim številom, je niz, ki bi ga radi izpisali, pravzaprav zaporedje takšnih celih števil, recimo $s = s_1 s_2 \ldots s_n$. Pisave pa lahko predstavimo z množicami števil, ki povedo, katere znake vsebuje posamezna pisava. Recimo, da je pisav m in označimo jih s $P_1, P_2, \ldots, P_m$. N: 7

Naloga sprašuje, kako razdeliti s na čim manj podnizov, pri čemer mora za vsak podniz obstajati kakšna pisava, ki vsebuje vse njegove znake. Pri vsakem znaku s-ja se lahko vprašamo, ali je pametno pri njem začeti nov podniz ali pa nadaljevati dosedanjega; načeloma je to odvisno od tega, s katero pisavo ta dosedanji podniz pišemo, saj mogoče trenutnega znaka sploh ni mogoče pisati z njo. Zato si zastavimo podprobleme oblike: kakšno je najmanjše število podnizov, na katere je treba razdeliti niz $s_1 s_2 \ldots s_i$, tako da se bo dalo vsak podniz v celoti izpisati z eno pisavo in da bo mogoče zadnji podniz izpisati s pisavo P_j? Recimo temu najmanjšemu številu podnizov kar $f(i,j)$. Takoj opazimo, da če $s_i \notin P_j$, je podproblem sploh nerešljiv in si lahko mislimo, da je $f(i,j) = \infty$. Tudi pri $i = 1$, torej izpisovanju prvega znaka, je rešitev na dlani: enega znaka sploh ne moremo razdeliti na več podnizov, zato je $f(1,j) = 1$ (če je seveda $s_1 \in P_j$). Pri kasnejših znakih, recimo pri s_i, pa moramo pregledati obe možnosti: (1) če začnemo tu nov podniz, bo skupno število podnizov enako $1 + f(i-1,k)$, če je k pisava, s katero smo končali dosedanji podniz (tisti, ki se končuje z znakom s_{i-1}). Ker smo začeli pri s_i nov podniz, nam je čisto vseeno, s katero pisavo smo izpisovali prejšnji niz, zato bomo vzeli tisti k, pri katerem je $f(i-1,k)$ najmanjša, saj hočemo čim manj podnizov. (2) Če pa tu ne začnemo novega podniza, ostane število podnizov enako kot pri prejšnjem znaku, torej $f(i-1,j)$. Od teh dveh možnosti moramo izbrati najmanjšo. Tako vidimo:

$$f(1,j) = \begin{cases} 1, & \text{če } s_1 \in P_j \\ \infty & \text{sicer.} \end{cases}$$

$$f(i,j) = \begin{cases} \min\{f(i-1,j), 1 + \min\{f(i-1,k) : 1 \le k \le m\}\}, & \text{če } s_i \in P_j \\ \infty & \text{sicer.} \end{cases}$$

Opazimo lahko lepo lastnost, da $\min f(i-1,k)$ po k ni nič odvisen od j, zato ga bomo lahko izračunali le enkrat in ga potem uporabljali pri računanju vrednosti $f(i,j)$ za vse možne j.

Po gornjih formulah bi lahko računali vrednosti $f(i,j)$ z rekurzivnim podprogramom, vendar bi bilo to neučinkovito, ker bi tako mnoge podprobleme reševali po večkrat. Najmanj, kar bi morali narediti, je to, da že izračunane vrednosti $f(i,j)$ odlagamo v neko tabelo in jih kasneje, če jih spet potrebujemo, preberemo od tam in jih ne računamo še enkrat (temu pristopu pravimo „pomnjenje“ ali „memoizacija“). Lahko pa tudi opazimo, da za izračun vrednosti $f(i,\cdot)$ potrebujemo le vrednosti $f(i-1,\cdot)$, zato lahko funkcijo f računamo

čisto sistematično z dvema gnezdenima zankama, zunanjo po naraščajočih i in notranjo po pisavah j (dinamično programiranje).

Če nas na koncu zanima tudi, kje točno so meje med podnizi, si moramo pri računanju vrednosti $f(i,j)$ zapomniti, kako smo prišli do vsakokratnega minimuma.

Razmislimo še o podatkovnih strukturah, ki bi jih uporabljal naš program. Niz s lahko predstavimo s tabelo; tudi vrednosti $f(i,j)$ lahko hranimo v tabeli. Za predstavitev pisav je več možnosti; predvsem si želimo, da bi se dalo čim hitreje pregledati vse pisave, s katerimi je mogoče napisati nek dani znak. Če imamo za vsako pisavo le seznam vseh znakov v njej, bo treba pregledati celoten seznam, kar bo šlo prepočasi; če bi imeli naraščajoče urejen seznam, shranjen v tabeli, bi ga lahko preiskali z bisekcijo, kar bi bilo že precej bolje. Uporabili bi lahko tudi razpršeno tabelo in tako še hitreje preverjali, ali je nek znak v pisavi prisoten ali ne. Lahko pa bi ustvarili tudi „obrnjen indeks“ (*inverted index*), v katerem bi imeli za vsak znak pripravljen seznam vseh pisav, ki ga vsebujejo. Slednje je za naše potrebe še najbolj učinkovito, saj se bomo tako lahko pri vsakem znaku (vsakem i) ukvarjali le s tistimi pisavami (tistimi j), ki ga res vsebujejo.

Naloga omenja tudi alternativni problem: poiskati najmanjše število različnih pisav, potrebnih za izpis vseh znakov niza s. To je pravzaprav le malo drugače zapisan problem pokrivanja množic (*set covering*), ki je znan NP-težak problem in zanj ne poznamo učinkovitih algoritmov, ki bi dajali optimalne rešitve (v našem primeru: poiskali najmanjše potrebno število pisav). Lahko bi napisali rešitev na podlagi rekurzije in načela „razveji in omeji“ (*branch and bound*), ki pa bi utegnila pri velikem številu pisav porabiti nesprejemljivo mnogo časa. Lahko pa z učinkovitejšimi algoritmi pridemo do rešitev, ki sicer ne bodo nujno optimalne, bo pa vsaj znano, za največ koliko so slabše od optimalnih (Johnsonov aproksimacijski algoritem za pokrivanje množic zagotavlja, da ne bi porabili več kot $(1 + \lg n)$-krat toliko pisav kot optimalna rešitev). Z raznimi hevristikami, lokalno optimizacijo, simuliranim ohlajanjem ipd. lahko poskusimo potem takšne rešitve še izboljšati.

R2003.X.9 Hiperkocka in mreža

N: 7 Do elegantne rešitve lahko pridemo s pomočjo Grayevega koda. To je način, kako našteti vsa n-bitna števila v takšnem vrstnem redu, da se po dve sosednji števili razlikujeta v natanko enem bitu.[7] Za prvih nekaj vrednosti n je Grayev

[7] Imenuje se po Franku Grayu, ki je takšno kodiranje uporabil v nekem patentu leta 1953 (podobne stvari so bile sicer znane že tudi prej). Grayevih kodov je več (pravzaprav zelo veliko, gl. npr. zaporedji A091299 in A006069 v *The On-Line Encyclopedia of Integer Sequences*); tu opisana različica je tista, na katero v praksi najpogosteje naletimo. Posplošimo jih lahko tudi na druge številske sestave (naštevanje števil od 0 do $b^n - 1$ tako, da se dve zaporedni števili razlikujeta v natanko eni števki, če ju zapišemo v b-iškem zapisu); gl. tudi rešitev naloge 1990.2.1.

kod takšen:

$$n = 1 : 0, 1$$
$$n = 2 : 00, 01, 11, 10$$
$$n = 3 : 000, 001, 011, 010, 110, 111, 101, 100$$

Grayev kod za n-bitna števila dobimo tako, da vzamemo kod za $(n-1)$-bitna števila, nato pa še isti kod v obratnem vrstnem redu; številom v prvi polovici tako dobljenega zaporedja pripišemo na levi še eno ničlo, številom v drugi polovici pa enico. Ni se težko prepričati, da se zdaj res vsako n-bitno število pojavlja v tem zaporedju natanko enkrat in da se po dve sosednji števili razlikujeta v natanko enem bitu. Naj bo $g_n(i)$ položaj (med 0 in 2^n-1) števila i v Grayevem kodu za n-bitna števila.

Preimenujmo v naši hiperkocki H_n vsako točko u v $(u \operatorname{div} 2^{n-m}, u \bmod 2^{n-m})$. Za x-koordinato torej uporabimo zgornjih m bitov števila u, za y-koordinato pa ostalih $n-m$ bitov. Zdaj obstaja povezava med (x, y) in (x', y') natanko v primeru, če se točki v eni od koordinat ujemata, v drugi pa se razlikujeta natanko v enem bitu.

Preimenujmo točke še enkrat tako, da bivša (x, y) po novem postane $(g_m(x), g_{n-m}(y))$. Recimo, da se po tem preimenovanju neki točki ujemata v eni od koordinat, v drugi pa se razlikujeta za 1, npr. (x, y) in $(x \pm 1, y)$. Zaradi lastnosti Grayevega koda sta morali biti x-koordinati teh dveh točk pred preimenovanjem dve taki števili, ki se razlikujeta le v enem bitu, y-koordinati pa sta bili pri obeh točkah enaki, torej obstaja v grafu med tema dvema točkama povezava. Podobno bi razmišljali, če bi se točki ujemali v x-koordinati in se v y-koordinati razlikovali za 1. Tako torej vidimo, da ima graf po dosedanjih preimenovanjih že točke s prav takimi oznakami kot mreža $M_{2^m,2^{n-m}}$, obenem pa vsebuje že tudi vse njene povezave. Zdaj ni treba drugega, kot da pregledamo vse povezave našega grafa in pobrišemo tiste, ki jih na mreži ni (to bodo povezave, ki povezujejo po dve točki, ki se v eni koordinati sicer ujemata, v drugi pa se razlikujeta za več kot 1).

Slabost opisane naloge je, da je precej lahka, če reševalec že pozna Grayev kod, drugače pa je težje priti do elegantne rešitve.

Za konec si oglejmo še, kako lahko računamo $g_n(t)$. Če se n-bitno število t začne v dvojiškem zapisu na ničlo, je v prvi polovici zaporedja, ki ga določa Grayev kod, sicer pa v drugi polovici; vsaka od teh dveh polovic zase je pravzaprav le malo dopolnjen (in v primeru druge polovice še obrnjen) Grayev kod za $(n-1)$-bitna števila, torej lahko natančen položaj števila t ugotovimo s pomočjo njegovega položaja v Grayevem kodu za $(n-1)$-bitna števila:

$$g_0(0) = 0$$
$$g_n(t) = \begin{cases} g_{n-1}(t), & \text{če } t < 2^{n-1} \\ 2^n - 1 - g_{n-1}(t \bmod 2^{n-1}), & \text{sicer.} \end{cases}$$

Pri n-bitnem številu t je $t \bmod 2^{n-1})$ preprosto t brez svojega zgornjega bita

(bita $n-1$). Vrednost $2^n-1-g_{n-1}(t \bmod 2^{n-1})$ pa ni nič drugega kot n-bitno število, ki ima zgornji bit prižgan, na spodnjih $n-1$ bitih pa ima vrednost $g_{n-1}(t \bmod 2^{n-1})$, v kateri so vsi biti ravno obrnjeni. Ker se $g_{n-1}(t \bmod 2^{n-1})$ seveda lahko računa po enakem pravilu, lahko rezultat računamo tudi bit za bitom in si pri tem le zapomnimo, koliko obračanj bita so zahtevali višji nivoji rekurzije. Če bi imeli števili t in $g_n(t)$ eksplicitno predstavljeni kot tabeli bitov, bi lahko postopek zapisali takole:

```
StObracanj := 0;
for i := n − 1 downto 0 do begin
  if Odd(StObracanj) then g[i] := t[i] xor 1 else g[i] := t[i];
  StObracanj := StObracanj + t[i];
end; {for i}
```

Vidimo lahko, da je dovolj, če si zapomnimo le, ali je število obračanj sodo ali liho. Namesto da bi StObracanj povečevali za t[i], je dovolj, če jo z njim **xor**amo, tako da bo StObracanj hranila pravzaprav le spodnji bit pravega števila obračanj; zato lahko obračanje zdaj namesto s stavkom **if** zapišemo kar kot g[i] := t[i] **xor** StObracanj. Zdaj pa vidimo, da med g[i] in StObracanj ni nobene razlike več in naš postopek se poenostavi v:

```
g[n − 1] := t[n − 1];
for i := n − 2 downto 0 do
  g[i] := t[i] xor g[i + 1];
```

Iz te zanke lahko tudi takoj opazimo, da je g[i] pravzaprav preprosto **xor** vseh bitov t[i..n − 1]. To pa lahko računamo še bolj elegantno z zamikanjem:

```
g := t xor (t shr 1);
g := g xor (g shr 2);
g := g xor (g shr 4);
g := g xor (g shr 8);
...
```

Po prvi vrstici imamo v vsakem bitu g-ja **xor** dveh zaporednih bitov t-ja; po drugi vrstici imamo zato v bitu g[i] že **xor** štirih bitov t[i..i + 3], po tretji v vsakem že **xor** osmih bitov t-ja in tako naprej. Štiri vrstice zadostujejo za n-je do 16; v splošnem potrebujemo $\lceil \lg n \rceil$ vrstic, če dekodiramo n-bitni Grayev kod.

R2004.X.1 Graf signala

V zanki se bomo istočasno premikali po x-koordinatah zaslona (od 0 do ZaslonX $-$ 1) in po vrednostih iz tabele Vrednosti (indeksi gredo od 0 do StVrednosti $-$ 1. Če je vrednosti več kot x-koordinat, se (kot zahteva naša naloga) premaknemo v vsakem koraku vsaj za en piksel v desno: x-koordinato povečamo za 1 in izračunamo, katero vrednost bi bilo najprimerneje prikazati na novi x-koordinati. Če pa je vrednosti manj kot x-koordinat, se premaknemo v vsakem koraku za eno vrednost naprej po tabeli Vrednosti in izračunamo, na kateri x-koordinati bi bilo treba prikazati to vrednost. N: 7

Formula za preračun med x-koordinatami in indeksi v tabeli Vrednosti temelji na ideji, naj skrajna leva x-koordinata ($x = 0$) ustreza prvi vrednosti (indeks 0 v tabeli Vrednosti), skrajna desna x-koordinata (torej ZaslonX $-$ 1) pa zadnji vrednosti (torej StVrednosti $-$ 1); vmes pa naj x-koordinate in indeksi vrednosti naraščajo premosorazmerno (če smo prehodili že polovico zaslona, hočemo videti vrednost na polovici tabele; ipd.). Tako na primer za vrednost z indeksom i vidimo, da je na $i/$(StVrednosti $-$ 1) poti od začetka do konca tabele (npr. na pol poti, če je i ravno indeks srednjega elementa tabele), zato mu ustreza koordinata na $i/$(StVrednosti $-$ 1) poti od levega do desnega roba zaslona; to pa je $x = (\text{ZaslonX}-1)\cdot i/(\text{StVrednosti}-1)$. Na enak način lahko pridemo tudi do formule za preračun x-koordinat v indekse. Ker tidve formuli ne dasta nujno celoštevilskih rezultatov, jih moramo pred uporabo še zaokrožiti do najbližjega celega števila (saj zahteva podprogram NarisiDaljico celoštevilske koordinate, indeksi v tabelo Vrednosti pa morajo biti tudi celoštevilski).

```
procedure NarisiSignal;
var i, x, y, x2, y2: integer;
begin
  x := 0; i := 0; y := Vrednosti[0];
  while (i < StVrednosti − 1) and (x < ZaslonX − 1) do begin
    if StVrednosti > ZaslonX then begin
      x2 := x + 1;
      i := Round(x2 / (ZaslonX − 1) * (StVrednosti − 1));
    end else begin
      i := i + 1;
      x2 := Round(i / (StVrednosti − 1) * (ZaslonX − 1));  { (†) }
    end; {if}
    y2 := Vrednosti[i];
    NarisiDaljico(x, y, x2, y2);
    x := x2; y := y2;
  end; {while}
end; {NarisiSignal}
```

Ena od slabosti takšnega prikazovanja signala je (kot pravi že besedilo naloge), da se lahko marsikaj zanimivega izgubi, če je število vrednosti veliko večje od

vodoravne ločljivosti zaslona. Pri vsaki x-koordinati prikažemo le eno vrednost, med x in $x+1$ pa mogoče mnogo vrednosti izpustimo; če se tam vmes zgodi v signalu kaj zanimivega, uporabnik o tem ne bo izvedel ničesar. Možna izboljšava našega podprograma bi lahko pri vsaki x-koordinati pregledala vse vrednosti signala, ki se po formuli iz vrstice (†) preslikajo v to x-koordinato; na grafu bi potem lahko prikazali razne statistične lastnosti te skupine vrednosti, na primer njihovo povprečje, srednjo vrednost (mediano), minimum, maksimum, itd.

R2004.X.2 Ribe

N: 8 Ker ribe vedno lahko požirajo le svoje sosede, tvorijo požrte ribe v vsakem trenutku neko strnjeno podzaporedje prvotnega zaporedja, trenutna riba pa je bodisi tista neposredno levo ali pa tista neposredno desno od požrtega podzaporedja. Vsota, ki bi jo radi maksimizirali, je vsota velikosti trenutne ribe in vseh požrtih.

Do največje vsote lahko pridemo s preprostim požrešnim algoritmom. Če trenutna riba nima nobene sosede, ki bi jo lahko požrla, se naš postopek ustavi; če ima eno, jo ta soseda pač požre; če pa ima dve sosedi in bi jo lahko požrla katerakoli od njiju, naj jo požre manjša izmed teh dveh sosed (oz. poljubna, če sta obe enako veliki). Večja izmed teh dveh sosed lahko kasneje še vedno požre tisto manjšo, obratno pa ne bi šlo.

Spodnji podprogram izračuna največjo vsoto, ki jo lahko dobimo, če začnemo pri ribi z. Trenutna riba in vse že požrte tvorijo neko strnjeno podzaporedje rib od L do R, vsota njihovih velikosti pa je v spremenljivki Vsota. Velikost trenutne ribe hranimo v spremenljivki T.

```
function NajboljsaVsota(Z: integer): integer;
var L, R, T, Vsota, aL1, aR1: integer;
begin
  L := Z; R := Z; T := A[Z]; { Trenutna riba je Z, požrta ni še nobena. }
  Vsota := T;
  while (L > 0) or (R < N) do begin
    { Trenutna riba je bodisi L ali pa R, vse ostale iz podzaporedja L..R pa so
      že požrte. Trenutno ribo lahko torej požre bodisi riba L - 1 ali pa riba R + 1.
      Poglejmo, kako veliki sta tidve ribi; če smo že na robovih zaporedja,
      si mislimo, da sta tam neki majhni ribi, ki trenutne ne moreta požreti. }
    if L > 1 then aL1 := A[L - 1] else aL1 := T - 1;
    if R < N then aR1 := A[R + 1] else aR1 := T - 1;
    { Mogoče nas ne more požreti nobena od sosed. }
    if (aL1 < T) and (aR1 < T) then break;
    { Če nas lahko požre le ena, naj nas pač tista;
      če obe, pa naj nas požre manjša od njiju. }
    if (aR1 < T) or ((T <= aL1) and (aL1 <= aR1))
```

```
      then begin Vsota := Vsota + aL1; L := L − 1; T := aL1 end
      else begin Vsota := Vsota + aR1; R := R + 1; T := aR1 end;
    end; {while}
    NajboljsaVsota := Vsota;
end; {NajboljsaVsota}
```

Do najboljše vsote sploh bi prišli, če bi poklicali NajboljsaVsota(Z) za vse z od 1 do n in si zapomnili največjo izmed dobljenih vsot. Podprogram NajboljsaVsota mora v najslabšem primeru pregledati vse ribe, zato ima časovno zahtevnost $O(n)$. Ker ga moramo klicati po enkrat za vsak z od 1 do n, je skupna časovna zahtevnost našega postopka $O(n^2)$. Prepričajmo se, da je ta rešitev res pravilna (dokaz je rahlo puščoben in se ga lahko brez velike škode preskoči); nato pa si bomo ogledali še eno učinkovitejšo rešitev.

Dokaz pravilnosti. Recimo, da je bilo nekaj rib že požrtih. To, kaj se bo lahko dogajalo v nadaljevanju, je odvisno le od tega, katere ribe so bile doslej požrte in katera je trenutna, neodvisno pa je od tega, pri kateri ribi smo začeli in v kakšnem vrstnem redu so se ribe žrle med sabo. Označimo torej trenutno stanje s trojico (l, r, t), $1 \le l \le r \le n$, ki nam pove, da so požrte vse ribe od l do r razen ene od rib l in r, ki pa je trenutna in ima velikost t (torej je t ali enako a_l ali pa enako a_r). (Spomnimo se, da že požrte ribe vedno tvorijo neko strnjeno podzaporedje vseh rib — v našem primeru je to ali $l+1, \ldots, r$ ali pa $l, \ldots, r-1$, odvisno pač od tega, ali je trenutna riba l ali riba r.)

Iz stanja (l, r, t) se lahko premaknemo v $(l-1, r, a_{l-1})$, če je $l > 1$ in $a_{l-1} \ge t$, ali pa v $(l, r+1, a_{r+1})$, če je $r < n$ in $a_{r+1} \ge t$. Vsota, ki jo skušamo maksimizirati, se v prvem primeru poveča za a_{l-1}, v drugem pa za a_{r+1}. Naloga pravzaprav sprašuje, kako priti do stanja s čim večjo vsoto, če začnemo v stanju (z, z, a_z) za nek z, $1 \le z \le n$.

Če v nekem stanju ni mogoč noben premik naprej, se moramo pač ustaviti; če je mogoč en sam premik, gremo pač po njem (ker velikosti rib ne morejo biti negativne, se nam ne splača ustaviti predčasno). Kaj pa, če sta mogoča oba premika — z drugimi besedami, če lahko trenutno ribo z velikostjo a požreta obe njeni sosedi, tako a_{l-1} kot a_{r+1}? Zgoraj smo zatrdili, da je bolje, če trentuno ribo požre manjša od obeh sosed. O tem se lahko prepričamo takole.

Recimo, da je $a_{l-1} \le a_{r+1}$ (obe pa sta seveda $\ge t$). Ali bo kaj narobe, če v naslednjem koraku trenutno ribo požre riba $l-1$, ne pa riba $r+1$? Naj bo $(l^\star, r^\star, t^\star)$ neko najboljše stanje, ki ga je iz trenutnega (l, r, t) sploh še mogoče doseči. To, da v naslednjem koraku žre riba $l-1$, je narobe le v primeru, če iz stanja $(l-1, r, a_{l-1})$ (v katerega bi po tem žretju prišli) ne moremo več priti v stanje $(l^\star, r^\star, t^\star)$ (niti v nobeno drugo enako dobro stanje). Prepričajmo se, da se to ne more zgoditi.

Ker se požrto podzaporedje ob nadaljnjih premikih lahko le širi, mora biti $l^\star \le l$ in $r^\star \ge r$. Recimo, da je $l = l^\star$; pot od trenutnega stanja (l, r, a) do $(l^\star, r^\star, t^\star)$ so torej sestavljala sama žretja z desne: $r+1$ je požrla trenutno

ribo (z velikostjo t), nato je riba $r+2$ požrla ribo $r+1$ in tako naprej. Toda ker je $a_{l-1} \geq t$ in hkrati $a_{l-1} \leq a_{r+1}$, bi lahko to pot popravili tako, da bi najprej $l-1$ požrla trenutno ribo, nato bi $r+1$ požrla ribo $l-1$, nadaljevalo pa bi se tako kot prej. S tem bi prišli namesto do stanja $(l, r^\star, t^\star)$ do stanja $(l-1, r^\star, t^\star)$, ki je vsaj tako dobro (pravzaprav še boljše; njegova vsota je večja za a_{l-1}). Torej s tem, ko smo začeli z žretjem z leve, nismo bili na koncu nič na slabšem, kvečjemu na boljšem.

Druga možnost je, da je $l^\star < l$. Mislimo si pot od (l, r, t) do $(l^\star, r^\star, t^\star)$. Če se ta pot začne z žretjem z leve, se iz (l, r, t) premaknemo v $(l-1, r, a_{l-1})$ in je torej $(l^\star, r^\star, t^\star)$ iz tega stanja še vedno dosegljivo; prav to smo hoteli dokazati. Če pa se pot od (l, r, t) do $(l^\star, r^\star, t^\star)$ začne z nekaj (recimo k) žretji z desne (kar nas pripelje do stanja $(l, r+k, a_{r+k})$), mora v njej vendarle prej ali slej nastopiti prvo žretje z leve (vsaj eno žretje z leve namreč potrebujemo, da bomo prišli od l do $l^\star$, ki je $< l$): takrat bo riba $l-1$ požrla ribo $r+k$ in prišli bomo v stanje $(l-1, r+k, a_{l-1})$. V prejšnjih korakih je riba $r+k$ požrla ribo $r+k-1$, ta še prej ribo $r+k-2$, ..., riba $r+2$ je požrla ribo $r+1$, ta pa prej trenutno ribo (z velikostjo t). Vse to pomeni, da je $a_{l-1} \geq a_{r+k} \geq \ldots \geq a_{r+1}$; po drugi strani pa smo na začetku rekli, da je $a_{l-1} \leq a_{r+1}$. Torej so vse te ribe v resnici enako velike: $a_{l-1} = a_{r+1} = a_{r+2} = \ldots = a_{r+k}$. Potem pa ni nič narobe, če našo pot preuredimo tako, da najprej izvedemo žretje z leve, nato pa k žretij z desne; ker so vse vpletene ribe enako velike, je to izvedljivo in nas popelje v stanje $(l-1, r+k, a_{r+k})$. Ker je $a_{r+k} = a_{l-1}$, je to stanje isto kot stanje $(l-1, r+k, a_{l-1})$, do katerega smo prišli prej; našo pot do optimalnega stanja se bo torej dalo nadaljevati na enak način kot prej. Tako torej vidimo, da je stanje $(l^\star, r^\star, t^\star)$ vsekakor še naprej dosegljivo, če v trenutnem stanju izvedemo najprej žretje z leve.

Doslej smo razmišljali o možnosti, da je $a_{l-1} \leq a_{r+1}$; če velja $\geq$ namesto $\leq$, bi lahko z analognim razmislekom ugotovili, da je pametno najprej izvesti žretje z desne. V vsakem primeru torej vidimo, da je v primeru, ko lahko trenutno ribo požreta obe sosedi, najbolje, če jo najprej požre manjša od njiju.

Učinkovitejša rešitev. Naša gornja rešitev je imela časovno zahtevnost $O(n^2)$; oglejmo si zdaj primer rešitve z zahtevnostjo $O(n)$. Recimo, da bi začeli pri neki ribi z in potem po vrsti izvedli nekaj žretij z ribami $i_1, i_2, \ldots, i_k$. Med temi ribami jih je mogoče nekaj, ki ležijo v prvotnem zaporedju levo od z; ker ribo vedno požre njena soseda, se mora v tem zaporedju rib, ki so žrle, najprej pojaviti $z-1$, nekje kasneje mogoče $z-2$, še kasneje $z-3$ in tako naprej. Ker manjša riba ne more požreti večje, vidimo, da se v našem zaporedju rib, ki so žrle, lahko pojavi največ toliko rib levo od z-ja, dokler njihove velikosti še ne začnejo padati. Podoben razmislek velja za ribe, ki ležijo v prvotnem zaporedju desno od z.

Recimo, da je riba i „levi maksimum“, če je strogo večja od svoje leve sosede (ali pa leve sosede sploh nima, torej $i = 1$), in „desni maksimum“, če

je strogo večja od svoje desne sosede (ali pa je sploh nima, torej $i = n$). Naj bo l_z najbolj desna riba, ki je levi maksimum in ni desno od z; naj bo r_z najbolj leva riba, ki je desni maksimum in ni levo od z. V prejšnjem odstavku smo videli, da če začnemo pri ribi z, bodo v vsoto, ki jo iščemo, vključene v najboljšem primeru ribe od l_z do r_z. Vse te ribe pa tudi res lahko vključimo v našo vsoto. Zaporedje velikosti $a_{z-1}, a_{z-2}, \ldots, a_{l_z}$ je nepadajoče, enako tudi zaporedje $a_{z+1}, a_{z+2}, \ldots, a_{r_z}$; zdaj lahko v mislih ti dve zaporedji zlivamo: če je trenutni člen prvega manjši od trenutnega člena drugega, se premaknemo naprej po prvem zaporedju in izvedemo žretje z leve, sicer pa se premaknemo naprej po drugem in izvedemo žretje z desne. Očitno je, da pri tem delamo same dovoljene korake: riba vedno žre svojo sosedo in to tako, ki ni večja od nje.

Če torej za neko z poznamo l_z in r_z, že tudi vemo, kaj bi nam vrnila funkcija NajboljsaVsota(z): vrnila bi vsoto $a_{l_z} + a_{l_z+1} + \ldots + a_{r_z-1} + a_{r_z}$. Lepo pri tem je, da ko enkrat poznamo l_z in r_z, ni težko priti do l_{z+1} in r_{z+1}. Definicija pravi, da je l_{z+1} najbolj desni levi maksimum, ki še ni desno od $z+1$. Mi pa vemo, da je en levi maksimum pri l_z in da nato vse do vključno z ni nobenega (sicer bi bil l_z šele tam); potem pa ostane le še $z+1$, ostale ribe so že desno od $z+1$. Torej je l_{z+1} bodisi enak l_z bodisi enak $z+1$, odvisno pač od tega, ali je $z+1$ levi maksimum ali ni. Podobno lahko razmišljamo pri r_{z+1}. En desni maksimum je pri r_z, med njim in z ni nobenega, tisti levo od z-ja pa so tudi levo od $z+1$ in za nas ne pridejo v poštev. Torej je $r_{z+1} = r_z$, če le ne leži levo od $z+1$; toda to je možno le v primeru, da je $r_z = z$: takrat moramo pač poiskati prvi naslednji desni maksimum.

Torej, ko se z povečuje, se tudi l_z in r_z povečujeta. Pri vsakem z-ju moramo izračunati vsoto velikosti rib $a_{l_z} + a_{l_z+1} + \ldots + a_{r_z-1} + a_{r_z}$; če se l_z poveča za 1, vsota na levi strani izgublja seštevance, če se r_z poveča za 1, pa jih na desni strani pridobiva; v vsakem primeru je torej ni težko popravljati. Zato imamo, ko pregledamo vse z-je od 1 do n, z računanjem novih vrednosti l_z in r_z ter popravljanjem vsot vsega skupaj le $O(n)$ dela.

```
function NajboljsaVsota2: integer;
var Z, LZ, RZ, Vsota, NajVsota, S: integer;
begin
  LZ := 1; RZ := 0; Vsota := 0; NajVsota := 0;
  for Z := 1 to N do begin
    { Trenutno je v LZ vrednost l_{z-1}, v RZ pa r_{z-1}. Izračunajmo novi LZ = l_z. }
    if Z = 1 then S := A[Z] - 1 else S := A[Z - 1];
    if S < A[Z] then { Z je levi maksimum; povečajmo LZ do Z. }
      while LZ < Z do begin Vsota := Vsota - A[LZ]; LZ := LZ + 1 end;
    { Izračunajmo novi RZ = r_z. }
    if RZ < Z then { Z - 1 je bil desni maksimum, }
      repeat          { zdaj pa moramo najti naslednjega. }
        RZ := RZ + 1; Vsota := Vsota + A[RZ];
```

```
        if RZ = N then S := A[RZ] − 1 else S := A[RZ + 1];
      until S < A[RZ];
    { Zdaj imamo v LZ pravo l_z, v RZ pa r_z. V spremenljivki Vsota je
      vsota velikosti vseh rib od LZ do RZ; mogoče je to najboljša vsota doslej. }
    if Vsota > NajVsota then NajVsota := Vsota;
  end; {for Z}
  NajboljsaVsota2 := NajVsota;
end; {NajboljsaVsota2}
```

R2004.X.3 Cezarjev kod

N: 8 Naj bo P naš znani podniz nekodiranega sporočila (parameter Znano), K pa naše kodirano sporočilo (parameter Kodirano). Najpreprostejša rešitev je najbrž ta, da poskusimo kodirati P z vsemi možnimi ključi (kličemo podprogram Kodiraj) in za vsakega od tako dobljenih kodiranih nizov preverimo, če se pojavlja kot podniz v kodiranem sporočilu K.

```
{ Če ne bi v nalogi pisalo, da sta podprograma StevilkaCrke in
  CrkaIzStevilke že dana, bi ju lahko napisali takole. }
function StevilkaCrke(Crka: char): integer;
  begin StevilkaCrke := Ord(Crka) − Ord('A') + 1 end;
function CrkaIzStevilke(Stevilka: integer): char;
  begin CrkaIzStevilke := Chr(Stevilka + Ord('A') − 1) end;

function Kodiraj(Niz: string; Kljuc: integer): string;
var Kodiran: string; i, Stev: integer;
begin
  Kodiran := '';
  for i := 1 to Length(Niz) do begin
    Stev := StevilkaCrke(Niz[i]) + Kljuc;
    if Stev > n then Stev := Stev − n;
    Kodiran := Kodiran + CrkaIzStevilke(Stev);
  end; {for i}
  Kodiraj := Kodiran;
end; {Kodiraj}

function Dekodiraj(Kodirano, Znano: string): integer;
var k: integer; KodZnano: string;
begin
  for k := 1 to n − 1 do begin
    KodZnano := Kodiraj(Znano, k);
    if JePodniz(Kodirano, KodZnano) > 0 then
      begin Dekodiraj := k; exit end;
  end; {for k}
  Dekodiraj := 0; { Primernega ključa sploh ni. }
end; {Dekodiraj}
```

Preverjanja, ali je KodZnano res podniz niza Kodirano, se lahko lotimo na različne načine. Najpreprosteje bo, če po vrsti pregledujemo vse možne položaje podniza v nizu in preverjamo, če se naš podniz res pojavlja na tistem mestu v nizu. Vsak znak podniza mora biti enak istoležnemu znaku niza.

```
function JePodniz(Niz, Podniz: string): boolean;
var i, j: integer;
begin
  for i := 1 to Length(Niz) − Length(Podniz) + 1 do begin
    { Poglejmo, če je Podniz enak Niz[i..i + Length(Podniz) − 1]. }
    j := 0;
    while j < Length(Podniz) do
      if Niz[i + j] = Podniz[j + 1] then j := j + 1 else break;
    { Podniz in Niz[i..i + Length(Podniz) − 1] se ujemata v prvih j znakih. }
    if j = Length(Podniz) then begin JePodniz := true; exit end;
  end; {while}
  JePodniz := false;
end; {JePodniz}
```

Kakšna je časovna zahtevnost te naše rešitve? Naj bo $|P|$ dolžina niza P, $|K|$ pa dolžina niza K. Za kodiranje P-ja z vsemi možnimi ključi smo porabili $O(n|P|)$ časa. Naj bo P_k niz, ki ga dobimo po kodiranju P-ja s ključem k. Ko podprogram JePodniz primerja P_k z nizom $K[i..i + |P| - 1]$, izvede recimo j_{ki} iteracij svoje notranje zanke **while**. Skupno število iteracij te zanke (po vseh klicih podprograma JePodniz) je torej $J := \sum_{k=1}^{n-1} \sum_{i=1}^{|K|-|P|+1} j_{ki}$. Opazimo lahko naslednje: recimo, da ob primerjanju niza P_k z nizom $K[i..i + |P| - 1]$ ugotovimo ujemanje prvih znakov, torej da je $P_k[1] = K[i]$. Če bi P kodirali s kakšnim drugim ključem, recimo s k' namesto s k, bi bil prvi znak dobljenega niza $P_{k'}$ vsekakor drugačen kot pri P_k, torej se prvi znak niza $P_{k'}$ gotovo ne bi ujemal s $K[i]$. Tako torej vidimo, da je od vrednosti $j_{1i}, j_{2i}, \ldots, j_{ki}$ lahko le ena večja od 1 (gotovo pa ni večja od $|P|$ — tako dolg je pač podniz, s katerim se ukvarjamo), ostale pa so enake 1 (razen če je $|P| = 0$; tedaj so vsi $j_{ki} = 0$). Torej je $\sum_{k=1}^{n-1} j_{ki} \leq (n-2) + |P|$. Tako dobimo $J = \sum_{i=1}^{|K|-|P|+1} \sum_{k=1}^{n-1} j_{ki} \leq |K|(n+|P|)$, kar pomeni, da ima naša rešitev časovno zahtevnost $O(|K|(n+|P|))$.

Obstajajo tudi učinkovitejše rešitve. Podprogram JePodniz bi lahko za iskanje podniza v nizu na primer uporabil Knuth-Morris-Prattov algoritem.[8] Ta ima najprej $O(|P|)$ dela za pripravo neke pomožne tabele, nato pa poišče podniz v nizu (ali pa ugotovi, da ga ni) v času $O(|K|)$. Skupna časovna zahtevnost vseh kodiranj in iskanj je tako le $O(n(|P| + |K|))$, kar je pravzaprav preprosto $O(n|K|)$, saj P gotovo ni daljši od K (oz. če je, lahko takoj rečemo, da primernega ključa ni).

[8] Gl. npr. Cormen *et al.*, *Introduction to Algorithms*, 34.4 v prvi izdaji, 32.4 v drugi.

Lahko pa bi si pri iskanju podnizov v nizu K pomagali z drevesom končnic (*suffix tree*).[9] To je podatkovna struktura, ki jo lahko za niz K zgradimo v $O(|K|)$ časa, nato pa v času $O(|P|)$ preverimo, če je nek P_k podniz K-ja. Tako bi porabili za celoten postopek le še $O(|K| + n|P|)$ časa. To utegne biti bolje od $O(n|K|)$ iz prejšnjega odstavka, če je P dovolj kratek v primerjavi s K.

R2004.X.4 Števila zveri

N: 10 Števila si predstavljajmo kot nize števk; v praksi bi jih shranjevali v tabelah. Oznaka xy naj pomeni stik nizov x in y, oznaka $x^{\underline{k}}$ pa stik k izvodov niza x. (Oznako x^k bomo uporabljali v običajnem pomenu, torej za k-to potenco števila x.) Števili a in b lahko razbijemo na posamezne števke in ju zapišemo kot $a = a_n a_{n-1} \ldots a_1 a_0$, $b = b_m b_{m-1} \ldots b_1 b_0$. Pri tem so a_0 enice, a_1 desetice in tako naprej. Ker je (kot pravi naloga) $a \geq b$, je $n \geq m$.

Naj bo c_i najmanjše naravno število, ki je $\geq a$ in vsebuje b kot podniz na mestih od i-tega do $(i+m)$-tega (mesta štejmo od desne proti levi; 0 so enice, 1 desetice, 2 stotice in tako naprej). (Na primer: za $a = 19345678$ in $b = 654$ imamo $c_0 = 19346\underline{654}$, $c_1 = 1934\underline{6540}$, $c_2 = 193\underline{65400}$, $c_3 = 19\underline{654000}$, $c_4 = 2\underline{6540000}$, $c_5 = \underline{654}00000$, $c_6 = \underline{654}000000$, itd.) Število, po katerem sprašuje naša naloga, je ravno najmanjši c_i po vseh možnih i.

Kako priti do c_i? Ta naj bi imel na mestih od i do $i+m$ vrednost b; oglejmo si, kakšne števke so na teh mestih v a-ju: recimo jim $d_i = a_{i+m} a_{i+m-1} \ldots a_i$. Če je $d_i = b$, se torej b pojavlja na teh mestih že v a-ju, zato je $c_i = a$. Če pa je $d_i \neq b$, lahko v mislih povečujemo a za 1 in gledamo, kaj se dogaja na mestih $i+m, \ldots, i$; ko tam dobimo vrednost b, se ustavimo in imamo c_i. No, ko povečujemo a za 1, se tudi d_i vsake toliko časa poveča za 1 (ko pride pri povečevanju a za 1 do prenosa z mesta $i-1$ na i; tik pred takšnim povečanjem je desnih i števk a-ja enakih $9^{\underline{i}}$, po njem pa $0^{\underline{i}}$), razen če je bil d_i pred trenutnim povečanjem enak $9^{\underline{m+1}} = 999\ldots9$ ($m+1$ devetic): v tem primeru pride do prenosa z mesta $i+m$ na $i+m+1$ in d_i pade na $0^{\underline{m+1}}$ ($m+1$ ničel).

Če je bila prvotna vrednost d_i manjša od b, bo pri povečevanju d_i dosegel b, še preden bo prišlo do tega padca na $0^{\underline{m+1}}$; c_i bo zato enak kar $a_n \ldots a_{i+m+1} b 0^{\underline{i}}$. Oglejmo si primer: $a = 12\underline{345}67, b = 987, i = 2$. Torej je d_i (podčrtani del a-ja) enak 345. Ko povečujemo a za 1, dobivamo $12\underline{345}68$, $12\underline{345}69$, …, $12\underline{345}99$, $12\underline{346}00$, $12\underline{346}01$, …, $12\underline{346}99$, $12\underline{347}00$, …, $12\underline{986}98$, $12\underline{986}99$, $12\underline{987}00$. Vidimo torej, da se d postopoma povečuje in to v vsakem takem trenutku, ko se števke desno od njega spremenijo iz samih devetic v same ničle. Ker je bil

[9] Ta drevesa je prvi predlagal Peter Weiner leta 1973 (*Linear pattern matching algorithms*, Proc. 14th Symp. on Switching and Automata Theory, pp. 1–11). Dandanes jih običajno gradimo z Ukkonenovim algoritmom (E. Ukkonen: *On-line construction of suffix trees*, Algorithmica, 14(3):249–260, September 1995). Gl. tudi R. Giegerich, S. Kurtz: *From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction*, Algorithmica, 19(3):331–353, November 1997.

d_i sprva manjši od b, se do trenutka, ko doseže d_i vrednost b, levo od njega v številu a ne zgodi še nič; desno pa so v tem prvem trenutku (takoj po zadnjem povečanju d_i) pač same ničle (v našem primeru dve ničli, ker je $i = 2$).

Če pa je bil d_i na začetku večji od b, se bo moral najprej povečati do $9^{\underline{m+1}}$, nato pasti na $0^{\underline{m+1}}$ in se nato povečati do b. Pri padcu d_i z $9^{\underline{m+1}}$ na $0^{\underline{m+1}}$ pa pride do prenosa z mesta a_{i+m} na a_{i+m+1}, torej se tisti del a-ja poveča za 1. Zato je v tem primeru $c_i = (a_n \ldots a_{i+m+1} + 1)b0^{\underline{i}}$. Oglejmo si isti primer kot prej: $a = 1234568$, $i = 2$, le za b vzemimo kakšno manjšo vrednost, recimo 210. Ko povečujemo a za 1, dobivamo 12$\underline{345}$68, 12$\underline{345}$69, …, 12$\underline{345}$99, 12$\underline{346}$00, 12$\underline{346}$01, …, 129$\underline{999}$8, 129$\underline{999}$9, 13$\underline{000}$00, …, 13$\underline{000}$99, 13$\underline{001}$00, …, 13$\underline{209}$98, 13$\underline{209}$99, 13$\underline{210}$00. Vidimo torej, da se mora d_i najprej povečati do samih devetic, nato pasti na same ničle (ob tem se del a-ja, ki leži levo od d, poveča za 1; v našem primeru se je povečal z 12 na 13), nato pa zrasti do b.

Zdaj znamo torej pri vsakem i določiti najmanjši tak c_i, ki je $\geq a$ in vsebuje b kot podniz od i-tega mesta naprej. Med c_i-ji, ki jih dobimo za različne vrednosti i, moramo vrniti najmanjšega (recimo mu $c^\star$).

```
function Naslednik(a_n a_{n-1} ... a_0, b_m b_{m-1} ... b_0);
begin
  c* := ∞;
  for i := 0 to n − m + 1 do begin
    d_i := a_{i+m} a_{i+m-1} ... a_i;
    if d_i = b then begin c* := a; break end;
    if d_i > b then c_i := (a_n a_{n-1} ... a_{i+m+1} + 1)b0^i
                else c_i := (a_n a_{n-1} ... a_{i+m+1})b0^i;
    if c_i < c* then c* := c_i;
  end; {for}
  Naslednik := c*;
end; {Naslednik}
```

Vrednost i je šla do $n - m + 1$ namesto do $n - m$, ker se lahko zgodi, da je iskani rezultat daljši od a; pri $a = 456$ in $b = 123$ je iskani rezultat na primer 1230. Pri $i = n - m + 1$ potrebujemo števko a_{n+1}; mislimo si, da je enaka 0. Večjih i nam ni treba pregledovati; ko se i povečuje od $n - m$ naprej, se c_i vsakič podaljša za eno ničlo na koncu, torej je desetkrat večji kot prej. Takih nima smisla pregledovati, saj nas zanima le najmanjši c_i.

Zgornji postopek predpostavlja, da znamo primerjati velika števila in povečevati število za 1. Oboje je enostavno; moramo se le zgledovati po tem, kako bi takšne reči počeli ročno. Ob primerjanju dveh števil bi gledali istoležne števke od bolj pomembnih proti manj pomembnim (torej od leve proti desni), dokler ne naletimo na mesto, kjer se števili v trenutni števki razlikujeta:

```
procedure Primerjaj(x_u x_{u-1} ... x_0, y_v y_{v-1} ... y_0);
begin
```

```
  for p := max{u, v} downto 0 do
    if x_p > y_p then begin WriteLn('x > y'); exit end
    else if x_p < y_p then begin WriteLn('x < y'); exit end;
  WriteLn('x = y');
end; {Primerjaj}
```

Če sta števili različno dolgi, ju v mislih podaljšajmo z ničlami: $x_p = 0$ za $p > u$ in $y_p = 0$ za $p > v$. Na primer: ko primerjamo $x = 123456$ in $y = 129876$, bi najprej primerjali 1 in 1 (pri $p = 5$), nato 2 in 2 (pri $p = 4$), nato pa 3 in 9 (pri $p = 3$) in iz dejstva, da je $3 < 9$, zaključili, da mora biti $x < y$.

Povečevanja števila za 1 pa se lotimo od desne proti levi (od manj pomembnih števk k bolj pomembnim). Spremenljivka c hrani prenos s prejšnjega mesta; temu prištejemo trenutno števko našega števila x; če je rezultat večji ali enak 10, se prenos nadaljuje še na naslednje mesto.

```
function PovecajZa1(x_u x_{u-1} ... x_0);
begin
  c := 1; i := 0;
  while i ≤ u or c > 0 do begin
    if i ≤ u then c := c + x_u;
    y_i := c mod 10; c := c div 10; i := i + 1;
  end; {while}
  PovecajZa1 := y_{i-1} ... y_0;
end; {PovecajZa1};
```

Kakšna je časovna zahtevnost naše rešitve? Pri vsakem i imamo $O(m)$ dela, da pripravimo število d_i in ga primerjamo z b-jem, nato pa še $O(n)$ dela, da pripravimo c_i in ga primerjamo s $c^\star$. Ker gre i do $n - m$, je skupna časovna zahtevnost v najslabšem primeru $O(n(n + m))$.

To rešitev lahko še izboljšamo. Naš b je dolg $m + 1$ števk; torej, če povečujemo a po 1, se v 10^{m+1} povečanjih izmenjajo na njegovih spodnjih $m+1$ števkah že vse možne vrednosti teh $m+1$ števk, med drugim tudi tista, pri kateri imajo te števke vrednost b. Kandidat c_i, ki ga naša funkcija Naslednik izračuna pri $i = 0$, je torej $< a + 10^{m+1}$, zato tudi za končni rezultat $c^\star$, ki ga ta funkcija vrne, velja $c^\star < a + 10^{m+1}$.

Osredotočimo se zdaj pri večjih i na tiste primere, ko je $d_i \neq b$. (Primerov, ko je $d_i = b$, se lahko rešimo takole: naš postopek naj na začetku pogleda, če se b pojavlja kot podniz v številu a; če se, je rezultat kar a in lahko takoj nehamo. S primernim algoritmom, kot je na primer Knuth-Morris-Prattov, bi se dalo to narediti v $O(n + m)$ časa.) Takrat je c_i vsekakor večji od a; za koliko večji? Recimo, da je $b \neq (d_i + 1) \bmod 10^{m+1}$. Če bi a postopoma povečevali po 1, bi sčasoma prišlo do prenosa z mesta $i - 1$ na i in ob tem bi se vrednost na mestih $i, \ldots, i + m$ povečala za 1 (mod 10^{m+1}). Ker smo rekli, da $b \neq (d_i + 1) \bmod 10^{m+1}$, vrednost na mestih $i, \ldots, i + m$ zdaj še

ni enaka b in bi morali s povečevanjem a-ja po 1 nadaljevati tako dolgo, da bi se vrednost na mestih $i, \ldots, i+m$ spremenila vsaj še enkrat; toda takoj po neki spremembi i-te števke so na spodnjih i številkah ničle in do naslednje spremembe pri i-ti številki bo prišlo šele, ko se bo a povečal še za 10^i. Tako torej vidimo, da je v tem primeru $c_i > a + 10^i$. Ker smo v prejšnjem odstavku ugotovili, da je $c^\star < a + 10^{m+1}$, to pomeni, da pri $i > m+1$ vrednost c_i prav gotovo ne pride v poštev za rezultat $c^\star$. Primer: $a = 12345678$, $b = 987$; pri $i = 4$ imamo $d_i = 234$ in $c_i = 19870000$. Če bi a povečevali po 1, bi prišli do 12350000, malo kasneje do 12360000 in šele precej kasneje do c_i. Že tisto povečanje od 12350000 do 12360000 pa nam zagotavlja, da je c_i vsaj za 10^i (v našem primeru: $10^4 = 10000$) večji od a. Takšen c_i je neobetaven, saj se že po samo 1000 povečanjih a-ja za 1 izmenjajo na spodnjih treh mestih vse kombinacije treh števk, med drugim tudi naša iskana 987. (Tako je c_0 tukaj enak 123456987, kar je precej boljša rešitev od 19870000.)

Tako torej vidimo, da glavne zanke našega postopka ni treba speljati do $i = n - m + 1$, ampak le do $i = m + 1$. Višje vrednosti i so zanimive le, če se tam v a-ju pojavi vrednost $d_i = (b-1) \bmod 10^{m+1}$ (na primer, če je $b = 123$, mora biti $d_i = 122$; če pa je $b = 000$, mora biti $d_i = 999$). Če bi bil a recimo enak 1234999956 in $b = 35$, torej $m = 1$, bi pri $i = 6$ dobili $d_i = 34$ in $c_i = 12350000000$, kar je v tem primeru tudi najmanjši od vseh c_i-jev. Čeprav je i velik (večji od $m+1$), je razlika $c_i - a$ majhna (v tem primeru le 44). V splošnem vidimo, da bo v takem primeru $c_i - a$ enako $10^i - a_{i-1}a_{i-2}\ldots a_0$, ker se mora a povečati le toliko, da prvič pride do prenosa z mesta $i-1$ na i (takrat se d_i poveča za 1 in tako postane enak b). Oglejmo si zdaj niz $s_i := a_{i-1}a_{i-2}\ldots a_0$. Če so vse te števke enake 9, je $c_i - a = 10^i - s_i = 1$; torej se bo b pojavil kot podniz že v $a+1$: boljše rešitve torej sploh ne bi mogli dobiti, razen če se ne pojavlja b kot podniz že v samem a-ju (kar pa tako ali tako preverimo posebej že na začetku, kot smo rekli v prejšnjem odstavku). Drugače pa naj bo u_i indeks najbolj leve take števke v s_i, ki je različna od 9 (torej je $s_i = 9^{\underline{i-u_i-1}}a_{u_i}a_{u_i-1}\ldots a_0$). Pri povečevanju a-ja po 1 bo sčasoma prišlo do prenosa z mesta $u_i - 1$ na u_i, tako da se bo u_i-ta števka povečala; ker pa je bila prej manjša od 9, zdaj ne bo prišlo do prenosa naprej. Zato se na mestih od i naprej ne bo zgodilo še nič; tam se lahko kaj zgodi šele ob naslednji spremembi na mestu u_i. Med dvema spremembama na mestu u_i pa se a poveča za 10^{u_i}, torej je $c_i - a > 10^{u_i}$. Spomnimo se, da je $c^\star \le c_0 < a + 10^{m+1}$; če je torej $u_i > m+1$, že vemo, da je c_i prevelik in da to ne bo tista najboljša rešitev, ki jo iščemo. Torej se s tistimi indeksi i, za katere je $u_i > m+1$, sploh ni treba ukvarjati. Pri ostalih indeksih i pa vidimo, da je $c_i - a = 10^i - s_i = 10^i - 9^{\underline{i-u_i-1}}a_{u_i}a_{u_i-1}\ldots a_0 = 10^{u_i+1} - a_{u_i}a_{u_i-1}\ldots a_0$. (Na primer: $100000 - 99932$ je enako $100 - 32$.) Zato za izračun razlike $c_i - a$ porabimo tu le $O(u_i) = O(m)$ časa, ne glede na to, kako velik je i. Te razlike lahko primerjamo med seboj po vseh i in vemo, da je najmanjši c_i (to pa je

naša iskana rešitev) tisti, ki ima najmanjšo vrednost $c_i - a$.

Recimo, da smo pri nekem $i > m+1$ ugotovili, da je $d_i = (b-1) \bmod 10^{m+1}$, obenem pa je $u_i \le m+1$ in smo zato morali računati $c_i - a$ v skladu z razmislekom iz prejšnjega odstavka; in recimo, da se nam je enako zgodilo tudi pri nekem kasnejšem indeksu j ($j > i$). Iz definicije u_i sledi, da so števke $a_{u_i+1}, a_{u_i+2}, \ldots, a_{i-1}$ vse enake 9; to med drugim pomeni (ker je $u_i \le m+1 < i$), da so števke $a_{u_i+1}, \ldots, a_{m+1}$ same devetice. Podobno, ker je $u_j \le m+1 < j$, so števke $a_{u_j+1}, \ldots, a_{j-1}$ same devetice — med njimi so tudi vse števke $a_{m+2}, \ldots, a_{j-1}$. Torej so števke od a_{u_i+1} do a_{j-1} same devetice, a_{u_i} pa ne; zato je $u_j = u_i$. V prejšnjem odstavku smo videli, da je $c_i - a = 10^{u_i+1} - a_{u_i} a_{u_i-1} \ldots a_0$; torej, ker je $u_j = u_i$, je $c_j - a = c_i - a$. Torej je $c_j = c_i$, kar pomeni, da s pregledovanjem indeksov, večjih od i, ne bomo pridobili nobene boljše rešitve od tiste pri i. Zato se lahko ustavimo, čim obdelamo prvi i, večji od $m+1$.

Zapišimo zdaj celoten postopek v enem kosu. Za razliko od prejšnjega postopka, ki je računal vrednosti c_i in si zapomnil najmanjšo (recimo $c^\star$), bomo tukaj računali razlike $r_i := c_i - a$ in si zapomnili najmanjšo. Če je $r^\star$ najmanjša razlika, je $c^\star = r^\star + a$ rezultat, ki ga iščemo. Lepo pri tem je, da so vrednosti c_i dolge $O(n)$ števk, vrednosti r_i pa (za tiste indekse i, ki jih bo pregledal naš novi algoritem) le $O(m)$ števk, zato je delo z njimi hitrejše.

algoritem Naslednik2(a, b);
vhod: $a = a_n a_{n-1} \ldots a_0$, $b = b_m b_{m-1} \ldots b_0$;
vrne najmanjše tako naravno število, ki je $\ge a$ in vsebuje b kot podniz;

1 **if** $m > n$ **or** ($m = n$ **and** $a \le b$) **then** vrni b;
2 **if** se b pojavlja kot podniz v a **then** vrni a;
3 $r^\star := \infty$;
4 **for** $i := 0$ **to** $\min\{m+1, n-m+1\}$ **do begin**
5 $\quad d_i := a_{i+m} a_{i+m-1} \ldots a_i$;
6 $\quad$ **if** $d_i > b$ **then** $r_i := 1b0^{\underline{i}}$ **else** $r_i := b0^{\underline{i}}$;
7 $\quad r_i := r_i - d_i a_{i-1} a_{i-2} \ldots a_0$;
8 $\quad$ **if** $r_i < r^\star$ **then** $r^\star := r_i$;
9 **end**; {*for*}
10 **if** $b = 0^{\underline{m+1}}$ **then** $b' := 9^{\underline{m+1}}$ **else** $b' := b - 1$;
11 naj bo i najmanjši indeks, ki je $> m+1$ in velja $a_{i+m} a_{i+m-1} \ldots a_i = b'$; (če takega i ni, skoči na korak 17);
12 naj bo u_i najmanjši tak indeks, ki je $< i$ in so $a_{u_i+1}, \ldots, a_{i-1}$ same devetke; če so $a_0, \ldots, a_{i-1}$ same devetke, si mislimo $u_i = -1$;
13 **if** $u_i \le m+1$ **then begin**
14 $\quad r_i := 10^{u_i+1} - a_{u_i} a_{u_i-1} \ldots a_0$; (pri $u_i = -1$ dobimo $r_i = 1$)
15 $\quad$ **if** $r_i < r^\star$ **then** $r^\star := r_i$;
16 **end**; {*if*}
17 vrni $a + r^\star$;

Kakšna je časovna zahtevnost tega novega postopka? $O(n+m)$ za vrstico 1; prav toliko za vrstico 2, če za iskanje podniza b v nizu a uporabimo npr. Knuth-Morris-Prattov postopek; d_i v vrstici 5 je dolg $m+1$ znakov, i pa je $\leq m+1$, zato je r_i v vrstici 6 dolg $O(m)$; vrstice 5–8 porabijo zato v vsaki iteraciji zanke po $O(m)$ časa, teh iteracij pa je največ $m+1$ (vrstica 4), tako da porabijo skupaj $O(m^2)$ časa. Vrstica 10 porabi $O(m)$ časa, vrstica 11 $O(n+m)$ (spet s Knuth-Morris-Prattovim algoritmom), vrstica 12 le $O(n)$ časa (zmanjšujmo u_i v zanki od $i-1$ navzdol, dokler še opažamo v a_{u_i} same devetke). Vrstici 14 in 15 porabita $O(m)$ časa, ker je $u_i \leq m+1$. Seštevanje v vrstici 17 porabi $O(n+m)$ časa. Tako vidimo, da je časovna zahtevnost našega novega postopka vsega skupaj $O(n+m^2)$, kar vsekakor ni slabše od $O(n(n+m))$ iz prejšnjega postopka (saj je $m \leq n$, oz. če ni, vemo, da je rezultat, ki ga iščemo, kar b, tako da lahko $m \leq n$ obdelamo kot poseben primer v $O(1)$ časa), lahko pa je celo precej bolje (če je m manjši od n — torej če je b krajši od a).

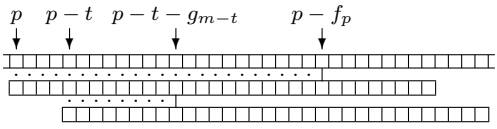
Tudi to rešitev lahko še precej izboljšamo. Za začetek izračunajmo nekaj pomožnih vrednosti, ki nam bodo prišle prav kasneje. Naj bo

$$g_p = \max\{d : 0 \leq d \leq p+1,\ b_p b_{p-1} \ldots b_{p-d+1} = b_m b_{m-1} \ldots b_{m-d+1}\} \text{ in}$$
$$f_p = \max\{d : 0 \leq d \leq p+1,\ d \leq m+1,\ a_p a_{p-1} \ldots a_{p-d+1} = b_m b_{m-1} \ldots b_{m-d+1}\}.$$

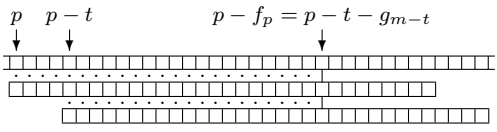
(Pri $d = 0$ si mislimo, da sta $b_p b_{p-1} \ldots b_{p-d+1}$ in $b_m b_{m-1} \ldots b_{m-d+1}$ prazna niza.) Z besedami: če gledamo niz b od števke p naprej („naprej“ je tu mišljeno proti desni), se ujema s celotnim nizom b v levih g_p znakih (v več pa ne). Podobno, če gledamo niz a od števke p naprej, se ujema s celotnim nizom b v levih f_p znakih (v več pa ne). Iz te definicije lahko takoj vidimo, da bi lahko vse g_p izračunali z dvema gnezdenima zankama (po p in po d), podobno pa tudi vse f_p, vendar bi nam vse to vzelo $O(m^2+nm)$ časa; na srečo gre tudi hitreje. Do vrednosti g_p (za $p = 0, \ldots, m$) lahko pridemo tako, da zgradimo drevo končnic (*suffix tree*) niza b; ko imamo enkrat to, ni težko določiti vseh g_p. Postopek gradnje takšnega drevesa je razmeroma zapleten in ga tu ne bomo podrobneje predstavljali (gl. op. na str. 40). Gradnja drevesa in računanje vseh g_p nam vzame $O(m)$ časa. Mimogrede, pri gradnji takega drevesa je koristno, če je na koncu niza nek znak, ki se drugod v nizu nikjer ne pojavlja; zato si mislimo, da obstaja tudi znak b_{-1}, ki je različen od vseh števk (0, 1, ..., 9); zanj tudi definirajmo $g_{-1} = 0$. Kasneje bomo videli, da je koristno uvesti tudi znak a_{-1}, ki je različen od vseh števk in tudi od b_{-1}.

Zdaj ko poznamo vse g_p, lahko vrednosti f_p izračunamo v $O(n+m)$ časa tako, da malo prikrojimo Knuth-Morris-Prattov postopek za iskanje podniza b v nizu a. Niz a bomo pregledovali od leve proti desni; najprej izračunajmo f_n kar po definiciji, z zanko po d. Recimo zdaj, da že poznamo f_p za nek konkreten p. To pomeni, da je $a_p a_{p-1} \ldots a_{p-f_p+1} = b_m b_{m-1} \ldots b_{m-f_p+1}$ in $a_{p-f_p} \neq b_{m-f_p}$. Oglejmo si zdaj, kako lahko poceni pridemo do vrednosti f za

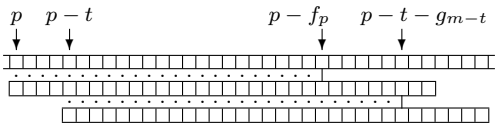
(1) Če je $g_{m-t} < f_p - t$:



(2) Če je $g_{m-t} = f_p - t$:



(3) Če je $g_{m-t} > f_p - t$:



Tri situacije, v katerih se lahko znajdemo pri računanju f_{p-t}, če že poznamo f_p. Pika med dvema znakoma pomeni, da sta enaka, črta pa, da sta različna.

Vidimo, da v teh situacijah velja naslednje:
(1) $f_{p-t} = g_{m-t}$;
(2) $f_{p-t} \geq g_{m-t}$;
(3) $f_{p-t} = f_p - t$.

---

nekaj naslednjih položajev, npr. do f_{p-t} za nek majhen pozitiven t. Vidimo, da se $a_{p-t} \ldots a_0$ in $b_{m-t} \ldots b_0$ ujemata vsaj v levih $f_p - t$ znakih, obenem pa vemo, da se $b_{m-t} \ldots b_0$ in $b_m \ldots b_0$ ujemata v levih g_{m-t} znakih, v naslednjem pa ne več. Primerjajmo zdaj g_{m-t} in $f_p - t$; ločimo tri možnosti (gl. gornjo sliko). (1) Če je $g_{m-t} < f_p - t$, lahko zaključimo, da se tudi $a_{p-t} \ldots a_0$ in $b_m \ldots b_0$ ujemata v levih g_{m-t} znakih, v naslednjem pa ne več; takrat je torej $f_{p-t} = g_{m-t}$. (2) Če je $g_{m-t} = f_p - t$, lahko zaključimo, da se $a_{p-t} \ldots a_0$ in $b_m \ldots b_0$ ujemata v levih $f_p - t = g_{m-t}$ znakih, za naslednjega pa ne vemo, če se v njem ujemata ali ne: vemo le, da je $a_{p-t-g_{m-t}} = a_{p-f_p} \neq b_{m-f_p} = b_{m-t-g_{m-t}} \neq b_{m-g_{m-t}}$, kar pa nam v splošnem ne pove nič zanesljivega o tem, ali sta si $a_{p-t-g_{m-t}}$ in $b_{m-g_{m-t}}$ enaka ali različna. Zato bomo morali nadaljevati s primerjanjem a-ja in b-ja od teh dveh položajev naprej. Lepo pri tem je, da se nam je, ko smo računali f_p, primerjanje a-ja z b-jem ustavilo pri a_{p-f_p}, kar je (zaradi $g_{m-t} = f_p - t$) ravno isti položaj kot $a_{p-t-g_{m-t}}$, na katerem bomo zdaj s primerjanjem spet začeli. Zato se z nobenim delom a-ja ne ukvarjamo po večkrat; časovna zahtevnost našega postopka bo le $O(n+m)$. (3) Če je $g_{m-t} > f_p - t$, lahko razmišljamo podobno kot pri (2); $a_{p-t} \ldots a_0$ in $b_m \ldots b_0$ se ujemata v levih $f_p - t$ znakih, na naslednjem mestu pa imamo $a_{p-t-(f_p-t)}$ (kar je isto kot a_{p-f_p}) in $b_{m-(f_p-t)}$; ker je $f_p - t < g_{m-t}$, je $b_{m-(f_p-t)} = b_{(m-t)-(f_p-t)}$, slednje je isto kot b_{m-f_p}, za tega pa po definiciji f_p vemo, da je različen od a_{p-f_p}. Torej lahko ravnamo enako kot v primeru (2), le da takoj opazimo neujemanje.

Postopek za računanje vrednosti f_p je torej lahko tak:

$p := n$; $f_p := 0$; **while** $a_{p-f_p} = b_{m-f_p}$ **do** $f_p := f_p + 1$;
while $p \geq 0$ **do begin**
{ f_p *že poznamo. Pregledujmo zdaj naslednje položaje,* $p - t$ *za* $t = 1, 2, \ldots$, *dokler pri njih velja primer (1) in jim lahko takoj določimo* f_{p-t}. }
$t := 1$;
while $g_{m-t} < f_p - t$ **do**
begin $f_{p-t} := g_{m-t}$; $t := t + 1$ **end**;
{ *Premaknimo se na položaj* $p - t$, *kjer velja primer (2) ali (3); v obeh primerih vemo, da je* $f_{p-t} \geq f_p - t$. *Poseben primer je* $f_p = 0$, *ko se zgolj premaknemo na položaj* $p - 1$ *in postavimo* f_{p-1} *na 0.* }
if $f_p = 0$ **then** $f_{p-t} := 0$ **else** $f_{p-t} := f_p - t$;
$p := p - t$;
{ *Določimo pravo vrednost* f_p *na novem položaju* p (*bivšem* $p - t$). }
while $a_{p-f_p} = b_{m-f_p}$ **do** $f_p := f_p + 1$;
end; {*while*}

Kot smo omenili že zgoraj, si na koncu nizov mislimo znaka a_{-1} in b_{-1}, ki sta različna od vseh ostalih znakov in še drug od drugega; to nam zagotavlja, da se zanke v gornjem postopku pravočasno in brez posebnih kompliciranj ustavijo. Gornji postopek nam izračuna tudi $f_{-1} = 0$; za g_{-1} pa smo že prej rekli, da je tudi enak 0.

Po teh predpripravah se posvetimo zanki v vrsticah 4–9 našega starega postopka Naslednik2. Ravno ta zanka namreč prinese časovno zahtevnost $O(m^2)$, saj ima $O(m)$ iteracij, v vsaki iteraciji pa nekaj operacij z zahtevnostjo $O(m)$. Oglejmo si zdaj, kako lahko s pomočjo vrednosti g_p in f_p izvedemo vsako iteracijo te zanke v času $O(1)$.

Najprej moramo primerjati d_i z b, da vidimo, ali velja $d_i > b$ ali $d_i < b$. Spomnimo se, da je d_i le podniz a-ja: $d_i = a_{i+m}a_{i+m-1}\ldots a_i$. Zato se d_i in b ujemata v levih f_{i+m} številkah, na naslednjem mestu pa se razlikujeta; tako moramo le primerjati ti dve številki, torej $a_{i+m-f_{i+m}}$ in $b_{m-f_{i+m}}$: če je večja prva, je $d_i > b$, če je večja druga, pa je $d_i < b$.

Naša zanka bo svoje delo opravila, če v vsaki iteraciji določi c_i in ugotovi, kateri od vseh c_i-jev je najmanjši. Potrebujemo torej postopek, ki bo znal v konstantnem času primerjati c_i in c_j. Seveda si tudi ne moremo privoščiti, da bi c_i in c_j eksplicitno zapisali v kakšno tabelo, saj nam potem vsaka iteracija že ne bi vzela več le konstantno mnogo časa. Pokazali bomo, da imata števili c_i in c_j tako preprosto in predvidljivo zgradbo, da ju lahko primerjamo že v konstantno mnogo časa (in to ne da bi ju zapisali eksplicitno); pri tem pa si bomo pomagali s prej izračunanimi vrednostmi g_p in f_p.

Recimo, da pri nekem konkretnem i velja $d_i > b$. Torej je število c_i oblike $(a_n a_{n-1}\ldots a_{i+m+1} + 1)b0^i$. V nizu $a_n a_{n-1}\ldots a_{i+m+1}$ je zadnjih nekaj števk mogoče enakih 9, prej ali slej pa je ena različna od 9 (če drugega ne, si mislimo, da obstaja še $a_{n+1} = 0$). Recimo, da je a_{t_i}, $t_i \leq i + m + 1$,

najbolj desna ne-devetka v tem delu a-ja. Torej je $a_n a_{n-1} \ldots a_{i+m+1} + 1 = a_n a_{n-1} \ldots a_{t_i+1} a_{t_i} 9^{\underline{t_i-1-m-1}} + 1 = a_n a_{n-1} \ldots a_{t_i+1}(a_{t_i}+1) 0^{\underline{t_i-i-m-1}}$. Zato je c_i sestavljen iz petih kosov (recimo jim „A_i“, „T_i“ in tako naprej):

$$c_i \quad = \quad \overbrace{a_n a_{n-1} \ldots a_{t_i+1}}^{A_i} \quad \overbrace{(a_{t_i}+1)}^{T_i} \quad \overbrace{0^{\underline{t_i-1-m-1}}}^{N_i} \quad \overbrace{b}^{B_i} \quad \overbrace{0^{\underline{i}}}^{S_i} .$$

(Vrednosti t_i za vse i od 0 do n lahko izračunamo na začetku našega postopka v času $O(n)$.) Pri tistih i, za katere je $d_i < b$, je stvar še preprostejša; tam je c_i sestavljen le iz treh delov (A_i, B_i in S_i):

$$c_i \quad = \quad \overbrace{a_n a_{n-1} \ldots a_{i+m+1}}^{A_i} \quad \overbrace{b}^{B_i} \quad \overbrace{0^{\underline{i}}}^{S_i} .$$

Spomnimo se, kako bi prej omenjeni podprogram Primerjaj primerjal števili c_i in c_j: šel bi s številom p od njune dolžine (to je načeloma n ali pa največ $n+1$) navzdol proti 0 in na vsakem koraku primerjal p-ti števki obeh števil, dokler ne bi na nekem mestu opazil, da se trenutni števki razlikujeta. Ker se p v vsaki ponovitvi zanke zmanjša za 1, bi takšno primerjanje trajalo v najslabšem primeru $O(n)$ časa.

Zgoraj smo videli, da sta tako c_i kot c_j sestavljena iz petih (ali celo samo treh) kosov. Trenutna vrednost p pade pri vsakem od njiju v enega od teh petih kosov. Izkaže se, da lahko v vsakem primeru, ne glede na to, katera dva kosa sta to, v konstantno mnogo časa bodisi odkrijemo položaj prvega neujemanja (in tako ugotovimo, ali je manjši c_i ali c_j) bodisi zmanjšamo p za toliko, da vsaj pri enem od c_i in c_j zdaj pade na začetek naslednjega kosa. Ker je kosov v vsakem od primerjanih števil največ 5, je jasno, da bomo morali izvesti le omejeno mnogo iteracij naše zanke, ne glede na dolžino a-ja in b-ja.

Oglejmo si konkreten primer: recimo, da je trenutni položaj p tak, da pri številu c_i pade v del A_i, pri številu c_j pa že v del N_j. Če imamo v neki tabeli za vsak položaj v a-ju pripravljen podatek o tem, koliko strnjenih ničel je desno od tega položaja (torej $h_p = \max\{d : a_p a_{p-1} \ldots a_{p-d+1} = 0^{\underline{b}}\}$ — vrednosti h_p za vse p lahko izračunamo na začetku našega postopka v času $O(n)$), lahko razmišljamo takole: pri c_j so od položaja p pa vse do konca dela N_j, torej vse do vključno položaja $j+m+1$, same ničle; to je skupaj $p-j-m$ ničel. Po drugi strani lahko v številu a od mesta p naprej vidimo h_p ničel; v številu c_j torej $\min\{h_p, p-t_i\}$ ničel, kajti po največ $p-t_i$ številkah bo dela A_i že konec in bomo prišli v T_i (tista številka tam pa je gotovo neničelna). Torej imata na naslednjih $\min\{p-j-m, h_p, p-t_i\}$ mestih tako c_i kot c_j ničlo; p lahko takoj zmanjšamo za toliko. S tem skokom smo mogoče prišli v c_i do konca dela A_i, mogoče v c_j do konca dela N_j, mogoče celo oboje, mogoče pa sicer nič od tega, kar pa pomeni, da smo odkrili mesto, kjer ima c_i že neničelno številko, v c_j pa še traja zaporedje ničel, torej imamo neujemanje in lahko s primerjavo zaključimo.

V prejšnjem odstavku smo razmišljali o primeru, ko smo v c_i znotraj A_i in v c_j znotraj N_j. Možnih je seveda še veliko drugih kombinacij, načeloma 25 (pet možnosti glede tega, v katerem delu c_i-ja smo, in pet glede tega, v katerem delu c_j-ja smo; izkaže se sicer, da so nekatere kombinacije nemogoče); s podobnim razmislekom se lahko za vsako od njih prepričamo, da lahko res v $O(1)$ časa odkrijemo neujemanje ali pa zmanjšamo p za toliko, da se v vsaj enem od števil c_i in c_j znajdemo v naslednjem kosu. Včasih si moramo pomagati z vrednostmi f_p in g_p, podobno kot smo si v prejšnjem odstavku pomagali z vrednostmi h_p.

Tako torej vidimo, da smo porabili na začetku $O(n+m)$ časa za računanje vrednosti f_p, g_p, h_p in t_i, nato pa pri vsaki izmed $O(m)$ iteracij naše glavne zanke še $O(1)$ časa za primerjavo d_i in b ter za primerjavo trenutnega c_i z največjim doslej znanim. Na koncu zanke vemo, pri katerem indeksu i; smo dobili najmanjši c_i; zdaj lahko izračunamo r_i in jo zapišemo v spremenljivko $r^\star$, nato pa izvedemo vrstice 10–17 postopka Naslednik2. Te vrstice porabijo, kot smo videli že zgoraj, le $O(n+m)$ časa, tako da zdaj tudi celoten postopek iskanja naslednika porabi le $O(n+m)$ časa. Takšna zahtevnost je v nekem smislu tudi najboljša možna, saj bi porabil $O(n+m)$ časa že tudi vsak postopek, ki bi niza a in b vsaj enkrat v celoti prebral.

R2004.X.5 Zamenjave

Odprta gesla (taka, pri katerih smo že naleteli na `$(`, na zaklepaj pa še ne) odlagajmo na sklad. Ko naletimo na zaklepaj, se zadnje odprto geslo (tisto z vrha sklada) zapre in ga zamenjamo z novo vrednostjo iz slovarja (če ga seveda najdemo v slovarju). Če je ob koncu vrstice še kaj gesel odprtih, pomeni, da manjka nekaj zaklepajev. N: 10

```
program Zamenjave;
const MaxSlovar = 1000;
      MaxGnezdenje = 50;
var Slovar: array [1..MaxSlovar, 1..2] of string;
    StGesel: integer;

  function Zamenjaj(Geslo: string): string;
  var i: integer;
  begin
    for i := 1 to StGesel do
      if Slovar[i, 1] = Geslo then
        begin Zamenjaj := Slovar[i, 2]; exit end;
    Zamenjaj := '$(' + Geslo + ')';
  end; {Zamenjaj}

var Sklad: array [0..MaxGnezdenje] of string; SP: integer;
    S: string; i, L: integer;
```

```
    T, TT: text;
begin
  { Preberimo gesla in njihove zamenjave. }
  Assign(T, 'zamenjave.in'); Reset(T); StGesel := 0;
  while true do begin
    ReadLn(T, S); L := Length(S);
    if L = 0 then break;
    i := 1; while i < L do if S[i] = ' ' then break else i := i + 1;
    StGesel := StGesel + 1;
    Slovar[StGesel, 1] := Copy(S, 1, i - 1);
    Slovar[StGesel, 2] := Copy(S, i + 1, L - i);
  end; {while}

  { Berimo besedilo, izpisujmo predelano besedilo. }
  Assign(TT, 'zamenjave.out'); Rewrite(TT);
  while not Eof(T) do begin
    ReadLn(T, S); SP := 0; Sklad[SP] := '';

    { Obdelajmo trenutno vrstico. }
    i := 1; L := Length(S);
    while i <= L do begin
      if S[i] = '$' then begin
        if i = L then { Dolar na koncu vrstice — pustimo ga kar pri miru. }
          Sklad[SP] := Sklad[SP] + S[i]
        else if S[i + 1] = '(' then begin { Začetek gesla. }
          SP := SP + 1; Sklad[SP] := ''; i := i + 1;
        end else if (S[i + 1] = '$') or (S[i + 1] = ')') then begin
          { Naleteli smo na par '$$' ali '$)'. }
          i := i + 1; Sklad[SP] := Sklad[SP] + S[i];
        end else { Napaka — pustimo dolar pri miru. }
          Sklad[SP] := Sklad[SP] + S[i];
      end else if S[i] = ')' then begin
        if SP = 0 then { Zaklepaj brez pripadajočega '$('. }
          Sklad[SP] := Sklad[SP] + S[i]
        else begin { Zamenjajmo geslo na vrhu sklada. }
          Sklad[SP - 1] := Sklad[SP - 1] + Zamenjaj(Sklad[SP]);
          SP := SP - 1;
        end; {if}
      end else { Navaden znak. }
        Sklad[SP] := Sklad[SP] + S[i];
      i := i + 1;
    end; {while}

    { Izpišimo prežvečeno vrstico. }
    Write(TT, Sklad[0]);
    { Še morebitna nedokončana gesla (če se pojavi '$(' brez pripadajočega ')'). }
    for i := 1 to SP do Write(TT, '$(', Sklad[i]);
    WriteLn(TT);
```

```
  end; {while}
  Close(T); Close(TT);
end. {Zamenjave}
```

Podprogram Zamenjaj išče zamenjavo za dano geslo tako, da po vrsti pregleduje zapise v slovarju, dokler ne naleti na pravega. Če je slovar velik, zna biti to časovno precej potratno; v tem primeru bi bilo dobro slovar hraniti v razpršeni tabeli.

R2004.X.6 Vžigalice

N: 11

Stanje igre lahko opišemo s v-terico števil $x = (x_1, \ldots, x_v)$, ki povedo, koliko vžigalic je še ostalo v posamezni vrstici. Na začetku igre je $x = a = (a_1, \ldots, a_v)$. Naj bo $N(x)$ množica vseh stanj, v katera lahko pridemo iz stanja x v eni potezi. Recimo, da je x trenutno stanje igre; naj bo $Z(x)$ logična vrednost (0 ali 1), ki nam pove, če obstaja za igralca, ki je trenutno na potezi, zanesljiva zmagovalna strategija (temu bomo rekli tudi, da je stanje zmagovalno), $P(x)$ pa logična vrednost, ki nam pove, če obstaja zanesljiva zmagovalna strategija za drugega igralca (tistega, ki trenutno ni na potezi; temu bomo rekli tudi, da je stanje pogubno, ker zagotavlja poraz tistemu, ki je na potezi, ko je igra v tem stanju).

Očitno je, da $Z(x)$ in $P(x)$ ne moreta biti oba hkrati resnična. Naloga pravi, da je vedno resničen natanko eden od njiju; prepričajmo se, da je res tako. Za končno stanje, $x = (0, 0, \ldots, 0)$, imamo $Z(x) = 1$ in $P(x) = 0$, kajti če smo mi na potezi in na mizi ni nobene vžigalice, pomeni, da je drugi igralec tik pred tem pobral zadnjo in tako izgubil igro. Drugače pa lahko razmišljamo takole: tisti, ki je na potezi, ima zagotovljeno zmago, če lahko v svoji naslednji potezi popelje igro v stanje, ki zagotavlja poraz drugega igralca; tisti, ki ni na potezi, pa ima zagotovljeno zmago šele, če mu je le-ta zagotovljena pri vseh možnih naslednikih trenutnega stanja (saj šele v tem primeru njegov nasprotnik, ki je trenutno na potezi, ne bo mogel storiti ničesar, da bi se izognil porazu). S simboli lahko to zapišemo takole: $Z(x) = \bigvee_{y \in N(x)} P(y)$ in $P(x) = \bigwedge_{y \in N(x)} Z(y)$.

Recimo zdaj, da za nekatera stanja x ne bi veljalo niti $Z(x)$ niti $P(x)$. Naj bo x med temi stanji tako z najmanjšim skupnim številom vžigalic (če je takšnih več, je vseeno, katero vzamemo). To pomeni, da za vse $y \in N(x)$ (ki imajo seveda manj vžigalic kot x) velja $Z(x) \Leftrightarrow \neg P(x)$. Ker x gotovo ni končno stanje (tisto brez vžigalic; zanj smo že videli, da velja $Z(x) \wedge \neg P(x)$), ima vsaj enega naslednika. Iz $\neg P(x)$ in dejstva, da ima x vsaj enega naslednika, sledi, da obstaja nek $y \in N(x)$, za katerega je $\neg Z(y)$. Ker ima y manj vžigalic kot x, sledi iz $\neg Z(y)$ tudi $P(y)$. To pa po definiciji Z pomeni tudi $Z(x)$, kar je v protislovju z našo predpostavko, da za x ne velja niti $Z(x)$ niti $P(x)$. Torej je zagotovilo iz besedila naloge res resnično: v vsakem stanju x je enemu od

igralcev zagotovljena zmagovalna strategija. Če velja $Z(x)$, je to tisti, ki je trenutno na potezi; če velja $P(x)$, pa tisti drugi.

Rešitev s pregledovanjem prostora stanj. Zdaj že tudi vidimo, kako bi lahko rešili nalogo: formuli za $Z(x)$ in $P(x)$ iz predprejšnjega odstavka bi lahko zapisali kot dva podprograma, ki bi v zanki pregledovala naslednike stanja x, torej vse $y \in N(x)$, in klicala drug drugega, da bi prišla do vrednosti $P(y)$ oz. $Z(y)$. Šlo bi pravzaprav tudi z enim samim podprogramom: stanje je zmagovalno natanko tedaj, ko lahko iz njega v eni potezi ustvarimo neko pogubno stanje.

```
function Z(x): boolean;
begin
  if x = (0,...,0) then return true;
  for each y ∈ N(x) do
    if not Z(y) then return true;
  return false;
end;
```

Naštevanje naslednikov trenutnega stanja bi lahko izvedli z dvema zankama — v zunanji si izberemo vrstico, iz katere bo trenutna poteza vzela vžigalice, v notranji pa si izberemo število vžigalic, ki jih bomo vzeli iz trenutne vrstice.

Slabost te rešitve je njena časovna potratnost, saj lahko pregleda veliko stanj med začetnim stanjem $(a_1, \ldots, a_v)$ in končnim stanjem $(0, \ldots, 0)$ (čeprav ne nujno vseh). Nekatera lahko pregleda celo po večkrat; temu bi se sicer lahko izognili, če bi vrednosti funkcije Z za že izračunana stanja hranili v kakšni tabeli (ali pa v razpršeni tabeli ali pa v drevesu), vendar bi s tem močno narasla poraba pomnilnika. Spomnimo se, da je iz stanja $a = (a_1, \ldots, a_v)$ načeloma mogoče doseči vsako stanje $(x_1, \ldots, x_v)$, ki ima $0 \le x_i \le a_i$ za vse $i = 1, \ldots, v$; to je z a-jem vred skupaj kar $(a_1 + 1)(a_2 + 1) \cdots (a_v + 1)$ stanj. Nekaj časa in pomnilnika bi lahko prihranili tako, da bi upoštevali, da lahko vrstice premešamo ter dodajamo ali brišemo prazne vrstice, ne da bi to na igro zares kaj vplivalo. Stanji $(3, 4, 2, 3, 2)$ in $(4, 3, 3, 2, 2)$ sta povsem enakovredni; tudi ko naštevamo naslednike tega stanja, je dovolj, če pri odvzemanju vžigalic gledamo le eno od vrstic s po tremi vžigalicami, druge pa ne, ipd.

Učinkovitejša rešitev. Nalogo lahko rešimo tudi precej hitreje, čeprav do te rešitve ni ravno preprosto priti. Označimo $x = (x_1, \ldots, x_v)$, $y = (y_1, \ldots, y_v)$, $2x = (2x_1, \ldots, 2x_v)$. Izkaže se, da je $Z(2x) = Z(x)$, razen če je $x_1 = \ldots = x_v = 1$, tedaj pa je $Z(2x) = \neg Z(x)$. Če je y sestavljen iz samih ničel in enic in je enic sodo mnogo, je $Z(2x + y) = Z(2x)$. Če pa je enic liho mnogo, je $Z(2x + y) = 1$. S temi ugotovitvami pridemo do naslednjega postopka (dokaz, da je ta rešitev pravilna, bomo videli na str. 55):

```
function Z(x): boolean;
```

begin
1 če ni v x nobena komponenta večja od 1:
2 če ima x liho mnogo enic, vrni false, sicer vrni true;
3 ponavljaj:
4 če je vsota komponent x liha, vrni true;
5 vsako komponento x deli z 2 in zaokroži navzdol;
6 če ni v x nobena komponenta večja od 1:
7 če ima x liho mnogo enic, vrni true, sicer vrni false;
end;

V vsaki iteraciji glavne zanke se število vžigalic v vsaki neprazni vrstici zmanjša vsaj za polovico, zato se izvede največ $O(\lg m)$ iteracij, če je imelo začetno stanje v vsaki vrstici največ m vžigalic.

Z nekaj pazljivosti lahko to rešitev implementiramo še bolj učinkovito. Oglejmo si, kakšne so videti operacije, ki jih izvaja naš gornji algoritem, če števila $x_1, \ldots, x_v$ predstavimo v dvojiškem zapisu.

Vsota komponent vektorja x (ki jo računamo v vrstici 4) je liha natanko tedaj, če je izmed števil $x_1, \ldots, x_v$ liho mnogo lihih; posamezno od teh števil pa je liho natanko tedaj, če je prižgan njegov najnižji bit. Za preverjanje, ali je takih števil liho ali sodo mnogo, si lahko pomagamo z operacijo xor: če xoramo vsa števila $x_1, \ldots, x_v$, je najnižji bit rezultata prižgan natanko tedaj, če je bil ta bit prižgan v liho mnogo izmed števili $x_1, \ldots, x_v$.

Vrstica 5 mora deliti vsak x_i z 2 in rezultat zaokrožiti navzdol; ta operacija preprosto pobriše najnižji bit števila x_i. Če nas bo v bodoče nekoč zanimal najnižji bit števila x_i (na primer ob izvajanju vrstice 4 v naslednji iteraciji glavne zanke), je to prav ista vrednost, ki je bila pred deljenjem z 2 na bitu 1. Torej ni treba res deliti vseh x_i z 2, pač pa je dovolj že, če si zapomnimo, na katerem bitu so zdaj tiste vrednosti, ki bi bile v najnižjem bitu, če bi deljenja res izvajali. To število se ob vsakem deljenju (torej: v vsaki iteraciji glavne zanke) poveča za 1.

V vrsticah 1 in 6 moramo preveriti, če je ostala še kakšna komponenta, večja od 1. Na začetku algoritma bi lahko pogledali, kateri je najvišji bit, ki je še prižgan v vsaj enem izmed x_i; recimo, da je to bit B. Ker ob vsakem deljenju z 2 (v vrstici 5) izgubi vsak x_i svoj najnižji bit, bo treba B deljenj, preden bo tudi največja izmed vrednosti x_i padla na 1. Tako ni težko preveriti, ali je še kateri od x_i večji od 1 ali ne: treba je le pogledati, če smo že izvedli B deljenj.

V vrsticah 2 in 7 moramo preveriti — po tistem, ko vemo, da so vse komponente vektorja x zdaj enake 0 ali 1 — ali je komponent z vrednostjo 1 liho mnogo. To je pravzaprav enako vprašanje kot v vrstici 4 in odgovor lahko spet dobimo s pomočjo operatorja xor.

Naj bo y vrednost, ki jo dobimo, če xoramo vsa števila $x_1, \ldots, x_v$. V zadnjih nekaj odstavkih smo se prepričali, da glavna zanka našega algoritma

pravzaprav ne počne drugega, kot da gleda, če je na bitih $0, \ldots, B-1$ v številu y kakšna enica (vrstica 4) — če jo najde, vrne true — drugače pa pogleda bit B števila y (vrstici 2 in 6) in vrne true ali pa false v odvisnosti od tega, ali je ta bit prižgan, in od tega, ali je B večji od 0 ali ne. Stanje x je zmagovalno, če je $B = 0$ in najvišji bit y-a ugasnjen ali pa je $B > 1$ in najvišji bit y-a prižgan. Zdaj lahko zapišemo spodnjo elegantno in učinkovito rešitev:

```
const v = ...;
type StanjeT = array [1..v] of integer;

function Zmagovalno(var x: StanjeT): boolean;
var i, y, m: integer;
begin
  m := x[1]; y := x[1];
  for i := 2 to v do begin
    if x[i] > m then m := x[i];
    y := y xor x[i];
  end; {for i}
  Zmagovalno := (m <= 1) = (y = 0);
end; {Zmagovalno}
```

V gornjem podprogramu je m največja vrednost v vektorju x. Pri $m \leq 1$ je torej $B = 0$ in y je dolg en sam bit; enak je 0, če je v stanju x sodo mnogo enic (in le takrat je x zmagovalno stanje). Pri $m > 1$ pa je $B > 0$ in stanje x je zmagovalno, če je prižgan vsaj eden od bitov v y (torej: če y ni enak 0).

Zmagovalna strategija. Če je neko stanje x zmagovalno (torej zanj gornja funkcija Zmagovalno vrne true) in smo pri tem stanju na potezi mi, vemo, da lahko igramo tako, da bomo nasprotnika prisilili v poraz. Kakšno potezo moramo najprej povleči? Pri $B = 0$ (oz. $m \leq 1$) sploh nimamo nobene izbire — v stanju x so le ničle in enice in ne moremo storiti drugega, kot da iz poljubne neprazne vrstice poberemo njeno edino vžigalico. Pri $B > 0$ (oz. $m > 1$) pa, ker je Zmagovalno vrnila true, vemo, da je $y \neq 0$. Iz trenutnega stanja moramo s svojo potezo narediti neko novo stanje, ki ne bo zmagovalno. Ker je $m > 1$, je v x neka komponenta $x_i > 1$; če je to edina komponenta, večja od 1, jo lahko v svoji potezi zmanjšamo bodisi na 1 bodisi na 0; ena od teh dveh možnosti nam bo zanesljivo dala $y = m = 1$ (katera, je odvisno od ostalih komponent stanja x: če je xor teh ostalih komponent enak 1, moramo x_i postaviti na 0, sicer pa na 1), kar pomeni, da novo stanje ne bo zmagovalno. Druga možnost pa je, da je v x več komponent, večjih od 1. Oglejmo si najvišji prižgani bit v številu y (recimo, da je to bit b); ker je y nastal z xoranjem komponent stanja x, pomeni, da ima tudi vsaj ena od komponent tega stanja tisti bit prižgan; recimo, da je to komponenta x_i. Če zdaj x_i spremenimo v $x_i \operatorname{xor} y$, se bo y postavil na 0, poleg tega pa se bo x_i zmanjšal,[10] tako da bo to veljavna poteza

[10] Zakaj je $x_i \operatorname{xor} y < x_i$? Najvišji prižgani bit v y je bit b, ki je prižgan tudi v x_i; ob

v skladu s pravili igre. Dobimo torej stanje, ki ima $y = 0$, vrednost m pa je še vedno > 1 (ker smo rekli, da x_i ni edina komponenta, večja od 1), zato novo stanje ni zmagovalno.

Dokaz pravilnosti. Strategijo iz prejšnjega odstavka lahko uporabimo tudi za eleganten dokaz, da je podprogram Zmagovalno pravilna rešitev naše naloge (spomnimo se, da tega doslej nismo dokazali — podprogram Zmagovalno temelji na trditvah, ki smo jih brez dokaza navedli na str. 52). Dokazovali bomo z indukcijo po številu vžigalic. Pri stanju z 0 vžigalicami dobimo $m = 0$ in $y = 0$ in Zmagovalno vrne true, kar je tudi prav. Pri eni vžigalici dobimo $m = 1$, $y = 1$ in funkcija pravilno vrne false. Recimo zdaj, da smo dokazali pravilnost že za vsa stanja z manj kot n vžigalicami; naj bo x neko stanje z n vžigalicami. (1) Če ima trenutno stanje $m = 1$, je vsaka vžigalica v svoji vrstici in očitno je, da je tako stanje zmagovalno natanko tedaj, ko je vrstic sodo mnogo; to pa je ravno enako pogoju $y = 0$, torej funkcija v tem primeru vrne pravilno vrednost. (2) Če pa je $m > 1$, ločimo dva primera. (2a) Pri $y \neq 0$ nam strategija iz prejšnjega odstavka kaže, kako lahko iz stanja x v eni potezi naredimo neko tako stanje, pri katerem funkcija Zmagovalno vrne false, ker pa ima to novo stanje manj vžigalic kot trenutno, sledi po induktivni predpostavki, da to stanje tudi v resnici ni zmagovalno; ker torej obstaja način, kako iz trenutnega stanja v eni potezi dobiti nezmagovalno stanje, mora biti trenutno stanje zmagovalno; in ker je $m > 1$ in $y \neq 0$, bi naša funkcija vrnila true, kar je torej popolnoma pravilno. (2b) Pri $y = 0$ pa naša funkcija vrne false; prepričajmo se, da stanje x v tem primeru res ni zmagovalno. Če bi bilo, bi se dalo iz njega v eni potezi narediti neko nezmagovalno stanje x'; ker bi imelo x' manj vžigalic od x, bi po induktivni predpostavki naša funkcija Zmagovalno za x' vrnila false. Torej bi za x' veljalo $m' = y' = 1$ ali pa $m' > 1$ in $y' = 0$. Poteza iz x v x' bi morala zmanjšati neko komponento x_i v x'_i, ostale komponente pa pustiti pri miru; torej je $y' = y \operatorname{xor} x_i \operatorname{xor} x'_i$, kar pomeni, da možnost $y' = 0$ odpade, saj je že y enak 0 in bi to pomenilo, da je $x_i = x'_i$. Torej mora biti $m' = y' = 1$; ker je bil $m > 1$, pomeni, da je bila v x večja od 1 le komponenta x_i in da smo jo zdaj zmanjšali na $x'_i \leq 1$; toda če je bila v x samo komponenta x_i večja od 1, bi moral biti najvišji prižgani bit števila x_i prižgan tudi v številu y, torej y ne bi bil enak 0. Prišli smo v protislovje, torej stanje x res ne more biti zmagovalno.

Ta dokaz nam sicer dokazuje, da je naša rešitev pravilna, ne pove pa nam kaj dosti o tem, kako bi človek do takšne rešitve sploh prišel. Dalo bi se sestaviti tudi bolj ilustrativen dokaz, ki bi lepše odražal to, kako lahko pridemo do te rešitve, če je še ne poznamo; ker pa bi bil ta dokaz malo bolj dolgovezen, ga

---

prehodu iz x_i v $x_i \operatorname{xor} y$ se ta bit ugasne, nekateri od nižjih bitov pa se lahko mogoče prižgejo; toda ugašanje bita b zmanjša vrednost števila za 2^b, prižiganje vseh nižjih bitov pa lahko vrednost poveča za največ $1 + 2 + 4 + \ldots + 2^{b-1} = 2^b - 1$, torej bo končna vrednost vendarle manjša od začetne.

tu ne bomo navajali.

Štetje stanj. Za konec si oglejmo še, koliko je vseh razporedov n vžigalic in koliko med njimi je neugodnih za prvega igralca (torej tistega, ki je prvi na potezi). Za začetek opazimo, da če lahko en razpored dobimo iz drugega le s spreminjanjem vrstnega reda vrstic ali pa z brisanjem ali dodajanjem praznih vrstic, sta z vidika naše igre oba razporeda pravzaprav čisto enakovredna: prazne vrstice na igro ne vplivajo, saj ni mogoče v njih izvesti nobene poteze, spreminjanje vrstnega reda vrstic pa na igro tudi ne vpliva, saj lahko še vedno izvajamo enake poteze kot prej (le da je vrstica, iz katere pobiramo, pač drugje kot prej). Zato se dogovorimo, da takšnih stanj ne bomo razlikovali med sabo; ali, z drugimi besedami, od vsake skupine stanj, ki se jih da dobiti drugo iz drugega le z premetavanjem vrstic in dodajanjem ali brisanjem praznih vrstic, bomo gledali le eno stanje — recimo kar tisto, v katerem ni praznih vrstic in v katerem so dolžine vrstic urejene nenaraščajoče. Primer takega stanja je na primer $(5, 3, 3, 2, 2, 2)$, medtem ko npr. stanja $(2, 5, 0, 3, 2, 2, 3)$, ki mu je sicer enakovredno, ne bomo obravnavali.

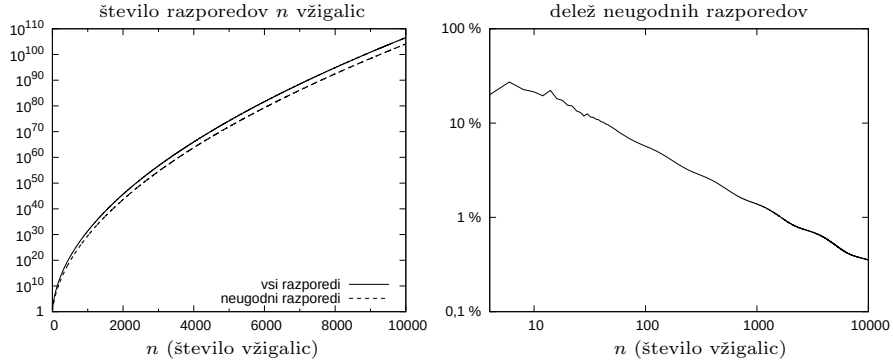
Koliko je vseh stanj z n vžigalicami? Kot pogosto pri takšnih problemih je koristno tudi zdaj ta problem malo posplošiti: vprašajmo se, koliko je vseh takih stanj z n vžigalicami, ki imajo v prvi vrstici kvečjemu k vžigalic; recimo temu $f(n, k)$. Ločimo dve možnosti: če je v prvi vrstici dejansko k vžigalic (kar je sicer možno le pri $n \geq k$), lahko ostale vrstice zgradimo na $f(n-k, k)$ načinov; če pa v prvi vrstici ni k vžigalic, pač pa manj, lahko celotno stanje zgradimo na $f(n, k-1)$ načinov. Tako smo prišli do rekurzivne zveze $f(n, k) = f(n-k, k) + f(n, k-1)$. Robni primeri so $f(0, 0) = 1$, $f(0, k) = 0$ za $k > 0$, $f(n, 0) = 0$ za $n > 0$, $f(n, k) = 0$ za $n < 0$. Funkcijo f lahko računamo z rekurzijo, lahko pa s pomočjo tabele (po naraščajočih k in pri vsakem k po vseh n, kar nam bo vzelo $O(n^2)$ časa in $O(n)$ pomnilnika). Število vseh razporedov z n vžigalicami je $f(n, n)$.[11]

Koliko pa je med stanji z n vžigalicami takih, ki so neugodna za prvega igralca (torej takih, pri katerih ima zmagovalno strategijo tisti, ki ni prvi na potezi)? Spomnimo se, kaj smo ugotovili za takšna stanja: imeti morajo $m > 1$ in $y = 0$ ali pa $m = 1$ in $y = 1$. Pri tem je y vrednost, ki jo dobimo pri xoranju vseh dolžin vrstic, m pa je dolžina najdaljše vrstice (zdaj je to kar dolžina prve vrstice, saj smo rekli, da gledamo le stanja, ki imajo vrstice urejene po nenaraščajoči dolžini). Naj bo $g(n, k, y)$ število vseh stanj, ki imajo n vžigalic, v prvi vrstici kvečjemu k vžigalic in pri katerih je xor vseh dolžin vrstic enak y. Enako kot pri f lahko tudi zdaj ločimo dve možnosti: če je v prvi vrstici k

[11] Štetje takšnih razporedov je pravzaprav isti problem kot štetje razbitij (particij) števila n (torej: na koliko načinov lahko izrazimo n kot vsoto nič ali več pozitivnih celih števil), ki smo ga srečali že pri nalogi 2002.3.3. Izkaže se na primer, da pri 1000 vžigalicah obstaja približno $2{,}4 \cdot 10^{31}$ razporedov, pri 10000 vžigalicah pa že $3{,}6 \cdot 10^{106}$ razporedov. Izkaže se, da velja $\ln f(n, n) = O(\sqrt{n})$.

vžigalic, lahko preostanek stanja zgradimo na $g(n-k, k, y \operatorname{xor} k)$ načinov; če pa je v prvi vrstici manj kot k vžigalic, lahko celo stanje zgradimo na $g(n, k-1, y)$ načinov. Torej je $g(n,k,y) = g(n-k, k, y \operatorname{xor} k) + g(n, k-1, y)$. Robni primeri so zdaj $g(0,0,0) = 1$, $g(n,0,y) = 0$ za $n \neq 0$ ali $y \neq 0$, $g(n,k,y) = 0$ za $n < 0$. Koristno je kot robni primer upoštevati tudi dejstvo, da je $g(n,k,y) = 0$, če je najvišji prižgani bit v y višji od najvišjega prižganega bita v k (ker z xoranjem samih števil, manjših ali enakih k, ne bo mogoče prižgati tistega najvišjega bita v y). Tudi funkcijo g lahko računamo z rekurzijo ali pa s pomočjo tabele (po naraščajočih k in pri vsakem k po vseh primernih n in y, kar nam bo vzelo $O(n^3)$ časa in $O(n^2)$ pomnilnika).

Vrednost $g(n,n,0)$ nam torej pove, koliko je vseh razporedov n vžigalic, ki imajo $y = 0$. Tu so všteti vsi nezmagovalni razporedi z $m > 1$; če je n pozitiven in sod, je v tem številu zajet tudi razpored, ki ima vsako vžigalico v svoji vrstici ($m = 1$, $y = 0$), ki pa je zmagovalen, zato je pravo število nezmagovalnih razporedov enako $g(n,n,0) - 1$; če pa je n lih, je pravo število nezmagovalnih razporedov $g(n,n,0) + 1$, ker v $g(n,n,0)$ ni vštet razpored, ki ima vsako vžigalico v svoji vrstici ($m = 1$, $y = 1$).



Na vodoravni osi obeh grafov je število vžigalic (n). Levi graf kaže, koliko je vseh razporedov n vžigalic v vrstice (polna črta) in koliko od teh razporedov je neugodnih za igralca, ki je prvi na potezi (črtkana črta). Desni graf kaže delež neugodnih razporedov v primerjavi z vsemi razporedi; če nanj položimo premico, vidimo, da je pri n vžigalicah delež neugodnih razporedov približno $0{,}92/n^{0{,}61}$. Pri obeh grafih so upoštevani le sodi n, saj je pri lihih n neugoden razpored vedno en sam (namreč tisti, ki ima v vsaki vrstici le po eno vžigalico).

Gornja grafa kažeta, kako narašča z n-jem število vseh razporedov, kako narašča število neugodnih razporedov in kako pada delež neugodnih razporedov v primerjavi z vsemi. Vidimo lahko, da je neugodnih razporedov v primerjavi z vsemi razmeroma malo; to je pravzaprav razumljivo, saj že iz pravil igre sledi, da če je neko stanje x nezmagovalno, so zmagovalna vsa stanja, ki jih lahko dobimo tako, da stanju x v eno od vrstic dodamo poljubno število vžigalic

(kar pa lahko storimo na veliko načinov). Če bi torej začetni razpored vžigalic izbirali naključno in imeli dva igralca, ki znata poiskati zmagovalno strategijo in igrati po njej, bi bila igra zelo nezanimiva, saj bi skoraj vedno zmagal prvi igralec. Pri 50 vžigalicah je od 204 226 možnih stanj le 18 437 stanj neugodnih za prvega igralca (to je približno 9,03 %); pri 100 vžigalicah pade ta delež na 5,69 % (10,8 od 190,6 milijonov stanj); pri 1 000 vžigalicah je neugodnih le še 1,38 % stanj ($3{,}3 \cdot 10^{29}$ od $2{,}4 \cdot 10^{31}$); pri 10 000 vžigalicah pa le še 0,35 % ($1{,}3 \cdot 10^{104}$ od $3{,}6 \cdot 10^{106}$).

R2004.X.7 Trikotniki

N: 11 Trikotnik $\triangle ABC$ je sestavljen iz treh daljic: AB, AC in BC. Takšne trikotnike lahko torej odkrijemo tako, da se postavimo v neko daljico (na primer AB) in pregledamo vse točke, ki so povezane s kakšnim od njenih krajišč (v našem primeru sta krajišči A in B); če je kakšna točka povezana z obema krajiščem (torej če imamo na primer daljici AC in BC), smo odkrili trikotnik (v našem primeru $\triangle ABC$).

Naloga pravi, da ne smemo izpisovati trikotnikov, ki so navznoter razdeljeni na več manjših trikotnikov. Če je trikotnik $\triangle ABC$ tako razdeljen na manjše trikotnike, mora biti vsaka od njegovih stranic tudi stranica enega od manjših trikotnikov; daljica AB mora biti na primer del nekega trikotnika $\triangle ABD$, točka D pa mora ležati v notranjosti trikotnika ABC (saj smo rekli, da gre za razdrobitev trikotnika $\triangle ABC$ na manjše trikotnike).

Ko torej pri daljici AB odkrivamo vse trikotnike, ki vsebujejo stranico AB, moramo pri vsakem preveriti, če vsebuje katerega od prej odkritih trikotnikov ali pa je sam vsebovan v njem — v tem primeru večjega od obeh trikotnikov zavržemo. Daljica AB je lahko vključena v največ dveh takih trikotnikih, ki nista navznoter razdeljena na več manjših (po en trikotnik na vsaki strani daljice AB).

Označimo krajišča daljic s $t_1, \ldots, t_n$. Za vsako od teh točk naj bo $N(t_i)$ množica tistih točk t_j, za katere obstaja daljica od t_i do t_j. Zdaj lahko zapišemo postopek za reševanje naloge:

za vsako daljico (t_i, t_j):
  $u :=$ **nil**; $v :=$ **nil**;
  za vsako točko $t \in N(t_i) \cap N(t_j)$:
    if $u =$ **nil then** $u := t$
    else if $t \in \triangle t_i t_j u$ **then** $u := t$
    else if $u \in \triangle t_i t_j t$ **then continue**
    else if $v =$ **nil then** $v := t$
    else if $t \in \triangle t_i t_j v$ **then** $v := t$
    else if $v \in \triangle t_i t_j t$ **then continue**
    else napaka v podatkih (daljice se križajo);

if $u \neq$ **nil then** izpiši $\triangle t_i t_j u$;
if $v \neq$ **nil then** izpiši $\triangle t_i t_j v$;

Pri izpisu moramo paziti, da ne izpišemo istega trikotnika po večkrat; pomagamo si lahko z oštevilčenjem točk. Rekli smo, da smo točke oštevilčili kot $t_1, \ldots, t_n$; če imamo trikotnik $\triangle t_i t_j t_k$, ga izpišimo le, če je $k < i$ in $k < j$. To nam zagotavlja, da ga bomo izpisali enkrat samkrat, čeprav bomo nanj naleteli trikrat. Naloga tudi pravi, naj bodo oglišča trikotnika izpisana v pozitivnem vrstnem redu; eden od vrstnih redov $t_i t_j t_k$ in $t_i t_k t_j$ je gotovo pozitiven (drugi pa negativen), le ugotoviti moramo, kateri. To lahko naredimo z vektorskim produktom (podprogram Ccw v spodnji rešitvi): točkam v mislih dodajmo še z-koordinato z vrednostjo 0 in izračunajmo vektorski produkt $(t_j - t_i) \times (t_k - t_i)$; če je njegova z-koordinata pozitivna, je tudi orientacija trikotnika $\triangle t_i t_j t_k$ pozitivna, sicer pa je negativne (za več o vektorskem produktu gl. rešitev naloge 2000.3.3).

Z vektorskim produktom lahko tudi preverjamo, ali neka točka leži v danem trikotniku ali ne (podprogram LeziV v spodnjem programu). Če se postavimo v točko A in gledamo v smeri točke B, je z-koordinata vektorskega produkta $(B - A) \times (T - A)$ pozitivna, če je T na naši levi, in negativna, če je T na naši desni (če ležijo A, B in T na isti premici, bo tisti vektorski produkt enak 0). Če imamo pozitivno orientiran trikotnik ABC in se postavimo v eno oglišče in gledamo proti naslednjemu oglišču, imamo cel trikotnik na svoji levi, ne glede na to, v katero oglišče smo se postavili. Če leži T v notranjosti trikotnika in je ta vedno na naši levi, mora biti tudi T vedno na naši levi (oz. desni). Če pa leži T v zunanjosti trikotnika, jo bomo vsaj v enem primeru zagledali na desni strani trenutno opazovane stranice.

V notranji zanki zgoraj opisanega postopka naj bi šel t po vseh točkah iz $N(t_i) \cap N(t_j)$, torej po takih, ki so povezane tako s t_i kot s t_j. V praksi bi to lahko naredili takole: imejmo za vsako točko t_i in t_j seznam sosed te točke (torej takih, ki so z njo neposredno povezane z eno od daljic); pojdimo po enem od teh seznamov in tako preglejmo vse točke t, ki so povezane s t_i; za vsako od njih potem preverimo, če je povezana tudi s t_j. Pametno je iti po tistem od obeh seznamov, ki je krajši; pokazati je mogoče, da nam to zagotavlja, da bomo pri izvajanju celotnega postopka le $O(n)$-krat preverjali, ali je neka soseda tudi v drugem seznamu. To preverjanje lahko izvedemo na razne načine; pametno bi bilo uporabiti razpršeno tabelo, lahko pa na začetku sezname besed uredimo in potem po njih poizvedujemo z bisekcijo. Spodnji program uporablja še malo preprostejšo rešitev: namesto da bi šel le po krajšem od obeh seznamov, gre istočasno po obeh in ju (ker sta oba urejena) „zliva“ ter pri tem ugotavlja, katere točke so prisotne v obeh. Slabost te rešitve je, da je lahko počasna, če ima kakšna točka veliko sosed (potem moramo pri vsaki od teh njenih številnih daljic iti po celem njenem dolgem seznamu sosed — skupna časovna zahtevnost bo $O(n^2)$, pri bisekciji pa bi bila le $O(n \log n)$, pri razpršeni tabeli pa le $O(n)$,

če se daljice v tabeli lepo razpršijo). Primer takšnega neugodnega problema je „obroč“ točk, povezanih z neko osrednjo točko (kot špice pri kolesu). Še ena slabost spodnjega programa, ki bi tudi prišla do izraza pri točkah z veliko sosedami, je ta, da za urejanje seznamov sosed uporablja postopek urejanja z vstavljanjem; ta algoritem je prijetno preprost, ima pa to slabost, da je lahko v najslabšem primeru precej neučinkovit (porabi $O(k^2)$ časa za urejanje seznama dolžine k). V praksi opisane slabosti našega postopka najbrž niso prehude, saj pri delu z mrežami trikotnikov v praktičnih problemih ne pričakujemo, da bodo seznami sosed zelo dolgi (vsaka točka nastopa le v razmeroma majhnem številu daljic).

Precej resnejša slabost spodnjega programa pa je naslednja. Koordinate točk si zapisuje kar v tabeli Xi in Yi (koordinati točke t_i sta Xi[i] in Yi[i]). Pri vsaki novi točki je treba iti po celi tabeli, da preverimo, če smo točko videli že kdaj prej ali pa je nova. Če imamo n daljic in m točk, bomo za to sprehajanje po tabeli porabili v najslabšem primeru skupno $O(n \cdot m)$ časa. Ker je $m \geq n/3$,[12] bi naš postopek porabil $O(n^2)$ časa in to ne glede na to, kakšno mrežo n daljic mu damo.[13]

```
program TrikotnikiIzDaljic;
const MaxDaljic = 18000; MaxTock = 2 * MaxDaljic;
      Eps = 1e−6;
var Xi, Yi: array [1..MaxTock] of real;
    M: integer; { število točk (krajišč daljic) }

  { Koordinati točke i sta (Xi[i], Yi[i]). Spodnji podprogram za dani koordinati
    poišče indeks te točke; če take točke še nimamo v tabelah Xi in Yi, jo doda. }
  function StTocke(X, Y: real): integer;
  var i: integer;
  begin
```

[12] O tem se lahko prepričamo s pomočjo znane Eulerjeve formule: za vsak povezan ravninski graf z m točkami, n povezavami in f ploskvami velja $m - n + f = 2$. (Če graf ni povezan, ampak je sestavljen iz c ločenih kosov, je $m - n + f = c + 1$.) Med ploskve v f je zajeta tudi „zunanjost“, ki ima recimo z stranic (ker naše daljice tvorijo trikotnike oz. like, ki se jih da dobiti s stikanjem trikotnikov, je $z \geq 3$), ostale ploskve pa so pri grafih iz naše naloge sami trikotniki. Ker vsaka daljica pripada dvema ploskvama, sledi $2n = z + 3(f - 1)$. Iz tega (in iz $z \geq 3$) sledi $f \leq 2n/3$; ker je po Eulerjevi formuli $m \geq n - f$, sledi $m \geq n - 2n/3 = n/3$.

[13] Namesto navadne tabele bi lahko koordinate hranili v razpršeni tabeli, vendar bi bilo potem težje upoštevati zahtevo iz naloge, naj imamo dve koordinati za enaki, če se razlikujeta za manj kot $\varepsilon = 10^{-6}$. Ravnino lahko v mislih razdelimo na celice velikosti $\varepsilon \times \varepsilon$ in vidimo, da se točke, ki se po obeh koordinatah od naše (x, y) razlikujejo za manj kot ε, nahajajo bodisi v isti celici kot ona bodisi v eni od osmih okoliških celic. Pri poizvedovanju po razpršeni tabeli bi tako namesto koordinat točke uporabljali koordinate celice. (Testni primeri, ki smo jih imeli pripravljene za to nalogo, sicer niso zahtevali takšnega kompliciranja, saj so imele koordinate vseh točk le šest števk za decimalno vejico, torej bi jih lahko pred računanjem razprševalnih kod preprosto pomnožili z 10^6 in tako dobili cela števila.) Namesto razpršene tabele bi lahko uporabili tudi kakšno drevesasto strukturo, na primer k-d-drevo, štiriško drevo ali pa R-drevo (gl. rešitev naloge 2004.2.3).

```
      for i := 1 to M do
        if (Abs(X − Xi[i]) < Eps) and (Abs(Y − Yi[i]) < Eps)
        then begin StTocke := i; exit end;
      M := M + 1; Xi[M] := X; Yi[M] := Y; StTocke := M;
    end; {StTocke}

    { Spodnja funkcija ugotovi orientacijo trikotnika ABC (pozitivna je, če so
      oglišča navedena v smeri, nasprotni smeri urinega kazalca). }
    function Ccw(A, B, C: integer): integer; { ccw = counterclockwise }
    var ABx, ABy, ACx, ACy, Z: real;
    begin
      ABx := Xi[B] − Xi[A]; ABy := Yi[B] − Yi[A]; { AB = vektor od A do B }
      ACx := Xi[C] − Xi[A]; ACy := Yi[C] − Yi[A]; { AC = vektor od A do C }
      Z := ABx * ACy − ACx * ABy; { vektorski produkt AB in AC je (0, 0, Z) }
      if Z > Eps then Ccw := 1 { pozitivna orientacija }
      else if Z < Eps then Ccw := −1 { negativna orientacija }
      else Ccw := 0; { točke so kolinearne, trikotnik je izrojen }
    end; {Ccw}

    function LeziV(T, A, B, C: integer): boolean; { Ali leži T v trikotniku ABC? }
    var i: integer;
    begin
      if Ccw(A, B, C) < 0 then begin i := B; B := C; C := i end;
      LeziV := (Ccw(A, B, T) >= 0) and (Ccw(B, C, T) >= 0)
                                                  and (Ccw(C, A, T) >= 0);
    end; {LeziV}

{ Tri je tabela vseh najdenih trikotnikov. Vsak trikotnik ima tri stranice, vsaka
  daljica pa je lahko stranica največ dveh trikotnikov, torej je število
  trikotnikov ≤ 2/3 števila daljic in spodnja tabela bo gotovo dovolj velika. }
var nTri: integer; Tri: array [1..MaxDaljic, 1..3] of integer;

    procedure ShraniTrikotnik(A, B, C: integer);
    var i: integer;
    begin
      if C = 0 then exit;
      { Vsak trikotnik bomo našli trikrat: enega od ABC in BAC;
        enega od BCA in BAC; in enega od CAB in ACB. }
      if (C > B) or (C > A) then exit;
      { Pri izpisu zahtevamo pozitivno orientacijo.
        Če je trenutno orientiran negativno, zamenjajmo dve oglišči. }
      if Ccw(A, B, C) < 0 then begin i := B; B := C; C := i end;
      { Zapomnimo si novi trikotnik. }
      nTri := nTri + 1; Tri[nTri, 1] := A; Tri[nTri, 2] := B; Tri[nTri, 3] := C;
    end; {ShraniTrikotnik}

var
    { Točka i ima StSosed[i] sosed, namreč točke Sosede[PrvaSoseda[i] + j]
      za j = 0, 1, ..., StSosed[i] − 1. }
```

```
  StSosed, PrvaSoseda: array [1..MaxTock] of integer;
  Sosede: array [1..2 * MaxDaljic] of integer;
  Krajisca: array [1..MaxDaljic, 1..2] of integer; { krajišči vsake daljice }
  N: integer; { število daljic }
  i, j, u, v, iu, iv, w, ww, Vrh1, Vrh2: integer;
  X1, Y1, X2, Y2: real; T: text;
begin {TrikotnikiIzDaljic}
  { Preberimo podatke o daljicah, izračunajmo stopnje. }
  Assign(T, 'tri.in'); Reset(T); ReadLn(T, N); M := 0;
  for u := 1 to 2 * N do StSosed[u] := 0;
  for i := 1 to N do begin
    ReadLn(T, X1, Y1, X2, Y2);
    u := StTocke(X1, Y1); v := StTocke(X2, Y2);
    Krajisca[i, 1] := u; Krajisca[i, 2] := v;
    StSosed[u] := StSosed[u] + 1;
    StSosed[v] := StSosed[v] + 1;
  end; {for i}
  Close(T);
  { Pripravimo sezname sosed. }
  PrvaSoseda[1] := 1;
  for u := 1 to M - 1 do PrvaSoseda[u + 1] := PrvaSoseda[u] + StSosed[u];
  for u := 1 to M do StSosed[u] := 0;
  for i := 1 to N do begin
    u := Krajisca[i, 1]; v := Krajisca[i, 2];
    Sosede[PrvaSoseda[u] + StSosed[u]] := v; StSosed[u] := StSosed[u] + 1;
    Sosede[PrvaSoseda[v] + StSosed[v]] := u; StSosed[v] := StSosed[v] + 1;
  end; {for i}
  { Uredimo vsak seznam sosed. }
  for u := 1 to M do begin
    for i := 1 to StSosed[u] - 1 do begin
      j := i - 1; v := Sosede[PrvaSoseda[u] + i];
      while j >= 0 do begin
        if Sosede[PrvaSoseda[u] + j] <= v then break;
        Sosede[PrvaSoseda[u] + j + 1] := Sosede[PrvaSoseda[u] + j]; j := j - 1;
      end; {while}
      Sosede[PrvaSoseda[u] + j + 1] := v;
    end; {while}
  end; {for i}
  nTri := 0;
  for i := 1 to N do begin
    u := Krajisca[i, 1]; v := Krajisca[i, 2];
    { Našli smo daljico u—v. Zlijmo seznama sosed u in v (z iu se sprehajamo
      po enem seznamu, z iv pa po drugem); vsaka w, ki je v obeh seznamih,
      tvori skupaj z u in v trikotnik. Izmed teh trikotnikov si moramo zapomniti
      tiste, ki ne vsebujejo nobenega manjšega trikotnika. Taka sta največ dva
```

```
      in njuna w-ja si bomo zapomnili v spremenljivkah Vrh1 in Vrh2. }
    iu := 0; iv := 0; Vrh1 := 0; Vrh2 := 0;
    while (iu < StSosed[u]) and (iv < StSosed[v]) do begin
      w := Sosede[PrvaSoseda[u] + iu]; ww := Sosede[PrvaSoseda[v] + iv];
      if w < ww then iu := iu + 1
      else if w > ww then iv := iv + 1
      else begin { Imamo tri daljice, ki tvorijo trikotnik uvw. }
        if Vrh1 = 0 then Vrh1 := w
        else if LeziV(w, u, v, Vrh1) then Vrh1 := w
        else if LeziV(Vrh1, u, v, w) then begin end
        else if Vrh2 = 0 then Vrh2 := w
        else if LeziV(w, u, v, Vrh2) then Vrh2 := w
        else if LeziV(Vrh2, u, v, w) then begin end
        else WriteLn('Napaka: daljice se križajo!');
        iu := iu + 1; iv := iv + 1;
      end; {if}
    end; {while}
    ShraniTrikotnik(u, v, Vrh1); ShraniTrikotnik(u, v, Vrh2);
  end; {for i}
  Assign(T, 'tri.out'); Rewrite(T); WriteLn(T, nTri);
  for i := 1 to nTri do begin
    WriteLn(T);
    for j := 0 to 3 do
      WriteLn(T, Xi[Tri[i, (j mod 3) + 1]]:0:6, ' ', Yi[Tri[i, (j mod 3) + 1]]:0:6);
  end; {for i}
  Close(T);
end. {TrikotnikiIzDaljic}
```

Viri dodatnih nalog: pretvarjanje znakov Unicode — Gorazd Božič; ribe — Gašper Fele Žorž; mobilni milijonar — Boris Horvat; ražnjič — Jure Leskovec; zamenjave — Ivo List; kocka Sierpińskega — Mojca Miklavec; graf signala, trikotniki — Marjan Šterk; ugibanje nizov — Dorian Šuc; vžigalice — Miha Vuk; stikala — Dare Zupanič; cezarjev kod — Anže Žagar; palindromi, seštevanje ulomkov — Klemen Žagar; urejanje logičnih vrednosti, nabori znakov, hiperkocka in mreža, števila zveri — Janez Brank.