1. tekmovanje IJS v znanju računalništva za srednješolce

6. maja 2006

NASVETI ZA 1. IN 2. SKUPINO

Nekatere naloge so tipa **napiši program** (ali **napiši podprogram**), nekatere pa tipa **opiši postopek**. Pri slednjih ti ni treba pisati programa ali podprograma v kakšnem konkretnem programskem jeziku, ampak lahko postopek opišeš tudi kako drugače: z besedami (v naravnem jeziku), psevdokodo (glej spodaj), diagramom poteka, itd. Glavno je, da je tvoj opis dovolj natančen, jasen in razumljiv, tako da je iz njega razvidno, da si dejansko našel in razumel pot do rešitve naloge.

Psevdokodi pravijo včasih tudi strukturirani naravni jezik. Postopek opišemo v naravnem jeziku, vendar opis strukturiramo na podoben način kot pri programskih jezikih, tako da se jasno vidi strukturo zank, vejitev in drugih programskih elementov.

Primer opisa postopka v psevdokodi: recimo, da imamo zaporedje besed in bi ga radi razbili na več vrstic tako, da ne bo nobena vrstica preširoka.

```
naj bo trenutna vrstica prazen niz;
pregleduj besede po vrsti od prve do zadnje:
    če bi trenutna vrstica z dodano trenutno besedo (in presledkom
    pred njo) postala predolga,
        izpiši trenutno vrstico in jo potem postavi na prazen niz;
    dodaj trenutno besedo na konec trenutne vrstice;
če trenutna vrstica ni prazen niz, jo izpiši;
```

Če pa v okviru neke rešitve pišeš izvorno kodo programa ali podprograma, obvezno poleg te izvorne kode v nekaj stavkih opiši, kako deluje (oz. naj bi delovala) tvoja rešitev in na kakšni ideji temelji.

Pri ocenjevanju so vse naloge vredne enako število točk. Svoje odgovore dobro utemelji. Prizadevaj si predvsem, da bi bile tvoje rešitve pravilne, ob tem pa je zaželeno, da so tudi čim bolj učinkovite (take dobijo več točk kot manj učinkovite). Za manjše sintaktične napake se načeloma ne odbije veliko točk.

Če naloga zahteva branje ali obdelavo kakšnih vhodnih podatkov, lahko tvoja rešitev (če v nalogi ni drugače napisano) predpostavi, da v vhodnih podatkih ni napak (torej da je njihova vsebina in oblika skladna s tem, kar piše v nalogi).

Nekatere naloge zahtevajo branje podatkov s standardnega vhoda in pisanje na standardni izhod. Za pomoč je tu nekaj primerov programov, ki delajo s standardnim vhodom in izhodom:

- Program, ki prebere s standardnega vhoda dve števili in izpiše na standardni izhod njuno vsoto:

```
program BranjeStevil;
var i, j: integer;
begin
  ReadLn(i, j);
  WriteLn(i, ' + ', j, ' = ', i + j);
end. {BranjeStevil}
```

```
#include <stdio.h>
int main() {
  int i, j; scanf("%d %d", &i, &j);
  printf("%d + %d = %d\n", i, j, i + j);
  return 0;
}
```

- Program, ki bere s standardnega vhoda po vrsticah, jih šteje in prepisuje na standardni izhod, na koncu pa izpiše še skupno dolžino:

```
program BranjeVrstic;
var s: string; i, d: integer;
begin
  i := 0; d := 0;
  while not Eof do begin
    ReadLn(s);
    i := i + 1; d := d + Length(s);
    WriteLn(i, '. vrstica: "', s, '"');
  end; {while}
  WriteLn(i, ' vrstic, ', d, ' znakov.');
end. {BranjeVrstic}
```

```
#include <stdio.h>
#include <string.h>
int main() {
  char s[101]; int i = 0, d = 0;
  while (gets(s)) {
    i++; d += strlen(s);
    printf("%d. vrstica: \"%s\"\n", i, s);
  }
  printf("%d vrstic, %d znakov.\n", i, d);
  return 0;
}
```

Opomba: C-jevska različica gornjega programa predpostavlja, da ni nobena vrstica vhodnega besedila daljša od sto znakov. Funkciji gets se je v praksi bolje izogibati, ker pri njej nimamo zaščite pred primeri, ko je vrstica daljša od naše tabele s. Namesto gets bi bilo bolje uporabiti fgets; vendar pa za rešitev naših tekmovalnih nalog v prvi skupini zadošča tudi gets.

- Program, ki bere s standardnega vhoda po znakih, jih prepisuje na standardni izhod, na koncu pa izpiše še število prebranih znakov (ne vštevši znakov za konec vrstice):

```
program BranjeZnakov;
var i: integer; c: char;
begin
  i := 0;
  while not Eof do begin
    while not Eoln do
      begin Read(c); Write(c); i := i + 1 end;
    if not Eof then begin ReadLn; WriteLn end;
  end; {while}
  WriteLn('Skupaj ', i, ' znakov.');
end. {BranjeZnakov}
```

```
#include <stdio.h>

int main() {
  int i = 0, c;
  while ((c = getchar()) != EOF) {
    putchar(c); if (i != '\n') i++;
  }
  printf("Skupaj %d znakov.\n", i);
  return 0;
}
```

1. tekmovanje IJS v znanju računalništva za srednješolce

6. maja 2006

NALOGE ZA PRVO SKUPINO

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

Rešitve bodo objavljene na `http://rtk.ijs.si/`.

1. Odstavki

V nekem besedilu so odstavki ločeni s praznimi vrsticami. **Napiši program**, ki prebira besedilo s standardnega vhoda in ga izpisuje na standardni izhod, pri tem pa, če se kdaj pojavi več zaporednih praznih vrstic, takšno skupino praznih vrstic nadomesti z eno samo prazno vrstico. (Za prazne vrstice štejemo pri tej nalogi le tiste vrstice, ki res ne vsebujejo nobenega znaka, niti presledkov. Predpostaviš lahko, da ni nobena vrstica daljša od sto znakov.)

2. Sneg

Letošnja zima je bila radodarna s snegom. Da bi opazovali časovno spreminjanje količine zapadlega snega, lahko za vsak dan posebej merimo, koliko snega je na novo zapadlo tisti dan in koliko se ga je stalilo (ali pa se je snežna odeja stanjšala zaradi sesedanja). Iz razlike med tema dvema količinama lahko ugotovimo, za koliko se je povečala ali zmanjšala debelina snežne odeje.

Od toplih jesenskih dni naprej vsak dan spremljamo dve meritivi:

- Debelino na novo zapadlega snega na ta dan (v milimetrih).
- Znižanje debeline snežne odeje na ta dan zaradi taljenja in sesedanja (v milimetrih).

Napiši program, ki s standardnega vhoda prebira podatke za 365 zaporednih dni. Za vsak dan dobi vrstico z dvema številoma: prvo je debelina novozapadlega snega, drugo pa znižanje snežne odeje zaradi taljenja in sesedanja. Za vsak dan naj program na standardni izhod izpiše debelino snežne odeje na koncu tega dneva. Predpostavi, da ob začetku merjenja še ni snega in da so vhodni podatki taki, kot bi se res lahko zgodili (da iz njih npr. ne sledi, da je bila debelina snežne odeje kdaj manjša od 0 mm).

Primer: če bi se podatki na vhodu začeli z vrsticami

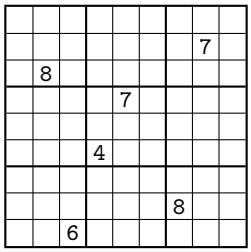
```
0 0
12 3
14 2
2 10
0 13
```

bi se moral izpis programa začeti z vrsticami

```
0
9
21
13
0
```

3. Sudoku

Sudoku je številčna križanka. Igralno polje velikosti 9×9 kvadratkov je dodatno razdeljeno na devet manjših kvadratov velikosti 3×3 (na spodnji sliki so predstavljeni z debelejšimi črtami), vanj pa je vpisanih nekaj števil (od 1 do 9). Primer:



V igralno polje igralec vpisuje števila od 1 do 9 in to tako, da so na koncu izpolnjeni naslednji trije pogoji: (1) v vsakem stolpcu se mora vsako število od 1 do 9 pojavljati natanko enkrat; (2) v vsaki vrstici se mora vsako število od 1 do 9 pojavljati natanko enkrat; (3) v vsakem od devetih malih kvadratov velikosti 3×3 se mora vsako število od 1 do 9 pojavljati natanko enkrat.

Na spodnji sliki levo vidimo primer pravilno izpolnjenega polja. Desno polje pa je izpolnjeno napačno (med drugim zato, ker v zadnjem stolpcu ni števila 6, se pa število 2 v njem pojavlja kar dvakrat).

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

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

Napiši program, ki bo sprejel izpolnjeno polje in izpisal niz `PRAVILNA`, če je rešitev pravilna, in `NAPACNA`, če ni pravilna. Če ti je naloga pretežka, lahko poskusiš napisati program, ki bo preverjal le pogoja (1) in (2), ne pa tudi pogoja (3) (z drugimi besedami: preveri naj, če so prisotna vsa števila od 1 do 9 v vsaki vrstici in v vsakem stolpcu, ni pa se mu treba ukvarjati z malimi kvadrati velikosti 3×3). Za takšno rešitev dobiš pri tej nalogi polovico vseh možnih točk.

Igralno polje je deklarirano kot dvodimenzionalna tabela. Primer deklaracije v programskem jeziku pascal:

```
type SudokuPoljeT = array [1..9, 1..9] of 1..9;
```

Predpostavi, da že obstaja nek podprogram Preberi, ki ga lahko pokličeš, da ti bo prebral izpolnjeno polje iz neke vhodne datoteke:

```
procedure Preberi(var T: SudokuPoljeT);
```

Še deklaraciji v C/C++:

```
typedef int SudokuPoljeT[9][9];
void Preberi(SudokuPoljeT T);
```

4. Naraščajoče besede

V nekaterih besedah so črke že urejene naraščajoče po abecedi: vsaka črka take besede pride v abecedi kasneje kot prejšnja črka te besede. Takšnim besedam pravimo *naraščajoče besede*. Primer naraščajoče besede je `AGILNOST` — `G` je v abecedi kasneje kot `A`, `I` je kasneje kot `G` in tako naprej.

Napiši program, ki prebere zaporedje besed s standardnega vhoda in na koncu izpiše najdaljšo naraščajočo besedo v njem. (Če je najdaljših več enako dolgih naraščajočih besed, je vseeno, katero izmed njih izpiše.) Predpostaviš lahko, da je vsaka beseda v svoji vrstici, v besedah nastopajo samo velike črke angleške abecede in nobena beseda ni daljša od 100 znakov.

5. Podnapisi

Nek predvajalnik filmov bi radi dopolnili tako, da bi znal prikazovati tudi podnapise. Te imamo podane v samostojnih datotekah, ločeno od filma, tako da lahko k istemu filmu pritaknemo podnapise v različnih jezikih. Ob predvajanju je treba, tik preden se prikaže posamezna sličica filma, ugotoviti, kateri podnapis pripada tej sličici (če sploh kakšen). Tvoja naloga je **napisati podprogram** PodnapisZaSlicico, ki ga bo sistem poklical pred prikazom vsake sličice, tvoj podprogram pa bo vrnil podnapis, ki ga je treba prikazati na tej sličici (oz. prazen niz, če ni treba prikazati nobenega podnapisa):

```
function PodnapisZaSlicico(StevilkaSlicice: integer): string;
```

Sličice filma so oštevilčene z zaporednimi celimi števili od 1 naprej. Če bi rad izvedel kakšne operacije še pred prvim klicem podprograma PodnapisZaSlicico, lahko napišeš tudi podprogram Inicializacija, ki naj bo takšne oblike:

```
procedure Inicializacija;
```

Sistem ga bo poklical na začetku predvajanja filma (ko so že znani podnapisi za celoten film, vendar pred prikazom prve sličice). V tem podprogramu lahko na primer inicializiraš svoje globalne spremenljivke, če jih boš uporabljal (v tem primeru seveda tudi ne pozabi navesti deklaracij teh spremenljivk).

Predpostavi, da so ti na voljo naslednji podprogrami, s katerimi dobiš od sistema podatke o podnapisih:

- **function** SteviloPodnapisov: integer;
  Vrne število podnapisov v celem filmu.
- **function** PrvaSlicica(StPodnapisa: integer): integer;
  function ZadnjaSlicica(StPodnapisa: integer): integer;
  function Vsebina(StPodnapisa: integer): string;
  Vrnejo podatke o podnapisu s številko StPodnapisa: številko prve in zadnje sličice filma, na katerih naj se vidi ta podnapis, in besedilo podnapisa. Podnapisi so oštevilčeni od 1 do SteviloPodnapisov v takšnem vrstnem redu, v kakršnem naj bi se prikazovali v filmu. Predpostaviš lahko, da se intervali PrvaSlicica..ZadnjaSlicica različnih podnapisov med sabo ne prekrivajo (torej nobeni sličici ne pripada več kot en podnapis).

Še deklaracije v C/C++:

```
int SteviloPodnapisov();
int PrvaSlicica(int StPodnapisa);
int ZadnjaSlicica(int StPodnapisa);
const char* Vsebina(int StPodnapisa);

void Inicializacija();
const char* PodnapisZaSlicico(int StevilkaSlicice);
```

(Opomba: „**const**“ v gornjih deklaracijah pomeni, da funkcija vrača kazalec na niz znakov, ki ga klicatelj ne sme spreminjati. Če te ta **const** kaj mede, se delaj, kot da ga ni.)

Zaželeno je, da je tvoja rešitev čim hitrejša, predvsem pa ne sme biti preveč potratna s pomnilnikom. Predpostavi, da predvajalnik nima dovolj pomnilnika, da bi si lahko npr. privoščil tabelo, v kateri bi bil po en celoštevilski element za vsako sličico filma (lahko pa bi imeli nekaj takšnih elementov za vsak podnapis v filmu — podnapisov je vendarle večdesetkrat manj kot sličic). (Rešitve, ki porabijo več pomnilnika, bodo dobile malo manj točk.)

1. tekmovanje IJS v znanju računalništva za srednješolce

6. maja 2006

NALOGE ZA DRUGO SKUPINO

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

Rešitve bodo objavljene na `http://rtk.ijs.si/`.

1. 1337ovščina

Med mladimi uporabniki komunikacijskih storitev (IRC, igrice, SMS) sproti nastaja žargon in tudi poseben način pisanja, ki služi več namenom: prikrivanju vsebine pred neposvečenimi tujci, poudarjanju drugačnosti kroga posvečenih uporabnikov (elitizem), hitremu tipkanju ipd.

Eden od načinov pisanja je zamenjava nekaterih črk (naključno, ne vseh in ne vedno) z enim od nadomestnih nizov, ki so videti podobno. Nadomestek je lahko en znak, lahko pa je sestavljen tudi iz več znakov. Tule je nekaj primerov možnih zamenjav:

```
a  →  4  /\  @  ^
b  →  8  6  13  )3
c  →  (  <
d  →  )  []
e  →  3  &  [-
f  →  ph
g  →  6  9  &  gee
h  →  #  )-(
i  →  1  !  |
.  .  .  .  .  .  .  .  .
```

Napiši program, ki bo prepisal neko besedilo s standardnega vhoda na standardni izhod tako, da bo v njem naključno zamenjal nekatere (ne nujno vseh) znake z eno od možnih zamenjav za ta znak. Predpostaviš lahko, da že obstaja funkcija Nakljucje, ki jo lahko uporabiš kot vir naključnih števil:

```
function Nakljucje(n: integer): integer;
```

Parameter `n` mora biti večji od 0, funkcija pa ob vsakem klicu vrne neko naključno izbrano celo število, večje ali enako 1 in manjše ali enako `n`.

Tabela možnih preslikav je podana podobno, kot je prikazano zgoraj. Da ti prihranimo branje in organizacijo preslikovalne tabele, si lahko pomagaš s funkcijo MozneZamenjave, za katero predpostavi, da že obstaja:

```
function MozneZamenjave(c: char): string;
```

Ta funkcija za podani znak `c` vrne niz, v katerem so vsebovana vsa možna nadomestila tega znaka; tako na primer za znak `'b'` dobimo niz `'8 6 13 )3'`. Pri tem so alternative med seboj ločene z natanko enim presledkom. Če za nek znak ni določenega nobenega nadomestila, nam funkcija MozneZamenjave pri tem znaku vrne prazen niz, kar le pomeni, da se takega znaka ne da zamenjati.

Primer rezultata (ob uporabi dopolnjene tabele iz primera):

```
Ta13e7a m0z|\|i# presl1|</\v je podana po)0bno, k0+ je pr1k@z4n0 29024j.
D@ ti |"ri#24n!mo b2anje i|\| o/2ga|\|!z@(1jo pr3sl!Xov/\lne 746313, $1
la#ko |"omag@$ s f00n|<<!j0 Mo2|\|3Z@AAenj/\v&,
```

Še deklaraciji v C/C++:

```
int Nakljucje(int n);
const char* MozneZamenjave();
```

2. Predpone

Telefonski promet poteka prek telefonskih central in vse telefonske številke, za katere skrbi neka centrala, imajo skupnih prvih nekaj števk — to je torej *predpona* (prefiks), ki pripada tisti centrali.

Recimo, da imaš klicni center, na katerem se vsak dohodni klic zabeleži s polno številko klicatelja. Rad bi opravil pregled klicev po telefonskih centralah klicateljev, zato si od telefonskih operaterjev prejel podatke o številčnih predponah in pripadajoči centrali. Primer (podatki so izmišljeni!):

01212	Trzin
012125	Domžale
01213	Mengeš
0121342	Kamnik
. . . . .	

Kot je razvidno iz zgornjega primera, so lahko daljše predpone določene svojim lastnim centralam, čeprav je krajši del že dodeljen neki drugi. Za vsako telefonsko številko, ki jo je zabeležil tvoj klicni center, moraš torej najti najdaljšo ujemajočo predpono, da lahko določiš telefonsko centralo klicatelja. (Ni pa se ti treba ukvarjati z določanjem imena centrale.)

Predpostavi, da obstajata naslednji funkciji, prek katerih lahko prideš do vseh predpon:

```
function StPredpon: integer; { Vrne število predpon. }
function PovejPredpono(ZapSt: integer): string;
  { Vrne predpono z zaporedno številko ZapSt, ki mora biti od 1 do StPredpon. }
```

Opiši postopek, ki bo za dano telefonsko številko poiskal in vrnil najdaljšo predpono, ki se ujema z začetkom te telefonske številke. Računaj na to, da bo moral tvoj postopek hitro najti najdaljšo predpono za veliko različnih telefonskih številk (recimo: na tisoče predpon, na milijone telefonskih številk). Zato si lahko poskusiš pred prvim izvajanjem tvojega postopka podatke preurediti in organizirati tako, da bo potem tvoj postopek za iskanje najdaljše predpone deloval čim hitreje. Če se odločiš za to možnost, opiši tudi svoje globalne spremenljivke, ki bi jih v ta namen uporabil, in opiši postopek za inicializacijo teh spremenljivk (ta postopek za inicializacijo bi poklical sistem še pred prvim klicem postopka za iskanje najdaljše predpone).

Če ti je to kaj v pomoč, si lahko postopek za iskanje najdaljše predpone predstavljaš kot podprogram takšne oblike:

```
function NajdaljsaPredpona(TelSt: string): string;
```

Postopek za inicializacijo pa kot podprogram takšne oblike:

```
procedure Inicializacija;
```

Vendar pa ni nujno, da svoja dva postopka zapišeš kot podprograma; lahko ju opišeš tudi kako drugače (v naravnem jeziku, s psevdokodo, ipd.), samo da bo opis dovolj natančen in čim bolj jasen in razumljiv.

Še deklaracije v C/C++:

```
int StPredpon();
const char* PovejPredpono(int ZapSt);
const char* NajdaljsaPredpona(const char* TelSt);
void Inicializacija();
```

3. Cestne lučke

Cestno podjetje ima lučke, ki jih postavlja ob delih na cesti, da zavaruje mesto del, da ne bi prišlo do nesreč. Lučke lahko utripajo ali pa potujejo levo ali desno, v odvisnosti od postavitve na cesti.

Cestni delavec ima na kontrolni napravi na voljo štiri gumbe, na katerih piše: „Ugasni“, „Utripaj“, „V levo“ in „V desno“.

Napiši štiri **podprograme** (Ugasni, Utripaj, VLevo in VDesno), ki skrbijo za to, da se lučke obnašajo tako, kot zahteva posamezni izmed teh štirih gumbov. Sistem bo ob pritisku na posamezni gumb poklical ustreznega od tvojih štirih podprogramov. Predpostaviš lahko, da sta na voljo naslednja sistemska podprograma:

- **procedure** Nastavi(BitnaSlika: integer) — nastavi stanje vseh lučk. Lučke so oštevilčene od 0 naprej (0 je najbolj desna lučka v vrsti, 1 je druga z desne strani itd.); lučka i se prižge, če je bit i v številu BitnaSlika enak 1, sicer pa se ugasne. (Predpostaviš lahko, da je lučk manj, kot je bitov v številu tipa integer.)
- **procedure** Pocakaj — počaka pol sekunde. Približno toliko časa naj mine med pomiki lučk pri gumbih „V levo“ in „V desno“ in med vklopom in izklopom lučk pri gumbu „Utripaj“.

Predpostavi, da se ob pritisku na gumbe na kontrolni napravi pokliče neka zunanja procedura, ki v trenutku prekine izvajanje trenutnega od tvojih štirih podprogramov (če se kateri od njih trenutno izvaja) in potem pokliče tistega, ki ustreza pravkar pritisnjenemu gumbu.

Predpostavi tudi, da so definirane tri celoštevilske konstante:

- SteviloLuck pove število lučk v sistemu; oštevilčene so od 0 (najbolj desna) do SteviloLuck − 1 (najbolj leva);
- POTUJE je vzorec bitov (celo število), ki pove, v kakšnem vzorcu naj bodo prižgane in ugasnjene lučke pri potovanju v levo/desno (ta vzorec lučk se v vsakem koraku ciklično zamakne za eno mesto v levo/desno; glej tudi spodnji primer);
- UTRIPA je vzorec bitov (celo število), ki pove, katere lučke naj utripajo pri utripajočih lučkah (ostale naj bodo ugasnjene).

Pomen bitov v konstantah POTUJE in UTRIPA je enak kot zgoraj pri parametru BitnaSlika podprograma Nastavi; vsi biti, ki ne sodijo med najnižjih SteviloLuck bitov, pa so ugasnjeni.

Na primer: recimo, da imamo pet lučk in je POTUJE $= 22_{10} = 10110_2$ in UTRIPA $= 9_{10} = 01001_2$. Spodnja slika prikazuje, kakšno naj bi bilo stanje lučk v prvih osmih korakih (štirih sekundah) po pritisku na posameznega od gumbov:

Čas po pritisku	Ugasni	Utripaj	V levo	V desno
0 s	⛀ ⛀ ⛀ ⛀ ⛀	⛀ ☼ ⛀ ⛀ ☼	☼ ⛀ ☼ ☼ ⛀	☼ ⛀ ☼ ☼ ⛀
0,5 s	⛀ ⛀ ⛀ ⛀ ⛀	⛀ ⛀ ⛀ ⛀ ⛀	⛀ ☼ ☼ ⛀ ☼	⛀ ☼ ⛀ ☼ ☼
1 s	⛀ ⛀ ⛀ ⛀ ⛀	⛀ ☼ ⛀ ⛀ ☼	☼ ☼ ⛀ ☼ ⛀	☼ ⛀ ☼ ⛀ ☼
1,5 s	⛀ ⛀ ⛀ ⛀ ⛀	⛀ ⛀ ⛀ ⛀ ⛀	☼ ⛀ ☼ ⛀ ☼	☼ ☼ ⛀ ☼ ⛀
2 s	⛀ ⛀ ⛀ ⛀ ⛀	⛀ ☼ ⛀ ⛀ ☼	⛀ ☼ ⛀ ☼ ☼	⛀ ☼ ☼ ⛀ ☼
2,5 s	⛀ ⛀ ⛀ ⛀ ⛀	⛀ ⛀ ⛀ ⛀ ⛀	☼ ⛀ ☼ ☼ ⛀	☼ ⛀ ☼ ☼ ⛀
3 s	⛀ ⛀ ⛀ ⛀ ⛀	⛀ ☼ ⛀ ⛀ ☼	⛀ ☼ ☼ ⛀ ☼	⛀ ☼ ⛀ ☼ ☼
3,5 s	⛀ ⛀ ⛀ ⛀ ⛀	⛀ ⛀ ⛀ ⛀ ⛀	☼ ☼ ⛀ ☼ ⛀	☼ ⛀ ☼ ⛀ ☼

Še deklaracije v C/C++:

```
const int SteviloLuck = ..., POTUJE = ..., UTRIPA = ...;
void Nastavi(int BitnaSlika);
void Pocakaj();
```

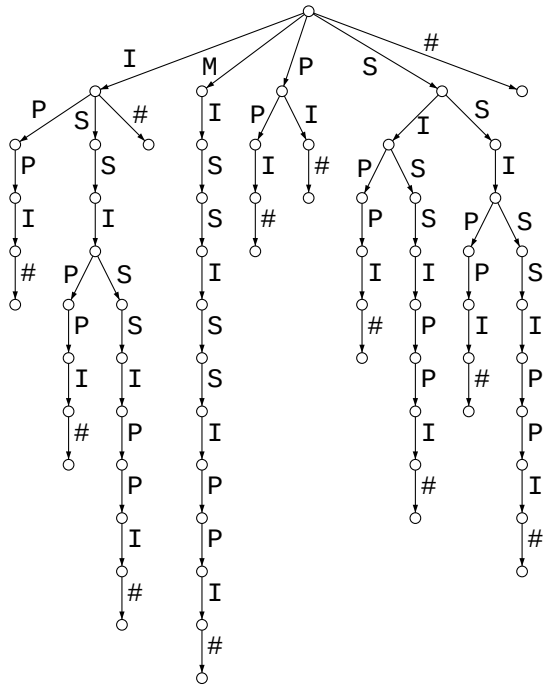
4. Drevo končnic

Človeški genom je sestavljen iz dolgega zaporedja nukleinskih kislin: adenina (`A`), gvanina (`G`), timina (`T`) in citozina (`C`). V genetiki nas pogosto zanimajo zaporedja, ki se v genomu ponavljajo na več mestih, ali pa taka, ki se pojavljajo pri več različnih organizmih hkrati.

Zaporedje nukleinskih kislin lahko predstavimo z nizom črk `A`, `C`, `T` in `G`; včasih pa je koristno takšen niz predstaviti tudi s pomočjo *drevesa končnic* (suffix tree).

Končnice (sufiksi) niza $s = s_1 s_2 \ldots s_n$ so nizi $s_i s_{i+1} \ldots s_n$ za $i = 1, 2, \ldots, n$. Drevo končnic niza s zgradimo takole. Najprej na konec niza s dodamo nek poseben znak za konec niza, ki se drugače ne pojavlja nikjer v nizu (pri tej nalogi bomo uporabljali znak „`#`“). Potem za vsako končnico tako podaljšanega niza, torej za vsak niz $s_i s_{i+1} \ldots s_n$`#` in še za niz `#` ustvarimo od korena drevesa navzdol zaporedje povezav z oznakami s_i, s_{i+1}, s_{i+2} in tako naprej, vse do s_n in nazadnje `#`. Če ima več končnic prvih nekaj črk skupnih, uporabljajo od začetka tudi iste povezave in vozlišča. Tako gre iz vsakega vozlišča naprej največ po ena povezava za vsako črko abecede (v abecedo štejemo zdaj tudi posebni znak `#`). Povezave do poddreves imamo vedno urejene po abecedi (mislimo si, da pride `#` na konec abecede).

Takšno drevo lahko zgradimo za poljuben niz s, ne le za nize iz črk `A`, `C`, `T` in `G`. Naslednja slika prikazuje na primer drevo končnic niza `MISSISSIPPI`:



(a) Po zgornjem zgledu **nariši** drevo končnic za besedo `RABARBARA`.

(b) V genomiki se pogosto srečujemo s kratkimi ponavljajočimi se nukleotidnimi zaporedji. V zaporedju `CATCATCATGGCATTCAT` se na primer podzaporedje `CAT` pojavi petkrat; v zaporedju `CGCGCGCGCGCTTTCGCGC` se podzaporedje `CGC` pojavi šestkrat, podzaporedje `CGCG` petkrat in tako naprej. Čeprav takšna zaporedja navadno ne kodirajo proteinov, imajo za organizem druge pomembne funkcije. **Opiši postopek**, ki ti za podano drevo končnic poljubnega niza in za dani števili m in k izpiše vsa (strnjena) podzaporedja dolžine m ali več, ki se v originalnem nizu pojavijo vsaj k-krat. (Primer: za niz `MISSISSIPPI` in $m = k = 2$ bi dobili podnize `IS`, `SI`, `SS`, `ISS`, `SSI` in `ISSI`.)

(*Opomba:* podvprašanje (a) je vredno pri tej nalogi tretjino točk, podvprašanje (b) pa dve tretjini.)

5. Razvajeni zvezdniki

Najel te je producent novega reality showa, ki bi rad na samotni otok za nekaj tednov poslal skupino tretjerazrednih zvezdnikov, nato pa snemal, kako bodo shajali drug z drugim. Pripravil si je seznam potencialnih udeležencev in jih oštevilčil od 1 do n. Težava je v tem, da so zvezdniki zelo muhasti in so mu postavili kup pogojev. Vsak zvezdnik je pripravil dva seznama in pravi, da bo sodeloval le, če bodo v oddaji sodelovali tudi vsi zvezdniki z njegovega prvega seznama in nobeden od zvezdnikov z njegovega drugega seznama. (Možno je tudi, da je kateri od teh seznamov prazen; lahko sta prazna celo oba.) Na samotni otok bomo poslali le prvih nekaj zvezdnikov s producentovega spiska, torej zvezdnike $\{1, 2, \ldots, k\}$ za nek k. **Opiši postopek**, ki ugotovi, kateri je največji k, za katerega lahko pošljemo na otok zvezdnike $\{1, 2, \ldots, k\}$ (ne pošljemo pa nobenega izmed zvezdnikov $\{k+1, k+2, \ldots, n\}$), ne da bi prekršili katero od omejitev, ki so jih ti zvezdniki postavili.

1. tekmovanje IJS v znanju računalništva za srednješolce

6. maja 2006

PRAVILA TEKMOVANJA ZA TRETJO SKUPINO

Vsaka naloga zahteva, da napišeš program, ki prebere neke vhodne podatke, izračuna odgovor oz. rezultat ter ga izpiše v izhodno datoteko. Programi naj berejo vhodne podatke iz datoteke *imenaloge*.`in` in izpisujejo svoje rezultate v *imenaloge*.`out`. Natančni imeni datotek sta podani pri opisu vsake naloge. V vhodni datoteki je vedno po en sam testni primer. Vaše programe bomo pognali po večkrat, vsakič na drugem testnem primeru. Besedilo vsake naloge natančno določa obliko (format) vhodnih in izhodnih datotek. Tvoji programi lahko predpostavijo, da se naši testni primeri ujemajo s pravili za obliko vhodnih datotek, ti pa moraš zagotoviti, da se bo izpis tvojega programa ujemal s pravili za obliko izhodnih datotek.

Delovno okolje

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

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

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

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

```
rtk -name JNovak
```

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

Za oddajo rešitve uporabi enega od naslednjih ukazov:

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

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

Da se zmanjša možnost zapletov pri prevajanju, ti priporočamo, da ne spreminjaš privzetih nastavitev svojega prevajalnika. Tvoji programi naj uporabljajo le standardne knjižnice svojega programskega jezika in naj ne delajo z datotekami na disku, razen s predpisano vhodno in izhodno datoteko. Dovoljena je uporaba literature (papirnate), ne pa računalniško berljivih pripomočkov (razen tega, kar je že na voljo na tekmovalnem računalniku), prenosnih računalnikov, prenosnih telefonov itd.

Ocenjevanje

Vsaka naloga lahko prinese tekmovalcu od 0 do 100 točk. Vsak oddani program se preizkusi na desetih testnih primerih; pri vsakem od njih dobi od 0 do 10 točk (praviloma 10, če je izpisal popolnoma pravilen odgovor, sicer pa 0; izjemi sta 1. in 5. naloga, kjer dobijo boljše rešitve več točk kot slabše), nato pa se te točke po vseh testnih primerih seštejejo v skupno število točk tega programa. Če si oddal N programov za to nalogo in je najboljši med njimi dobil M (od 100) točk, dobiš pri tej nalogi $\max\{0, M-3(N-1)\}$ točk. Z drugimi besedami: za vsako oddajo (razen prve) pri tej nalogi se ti odbijejo tri točke. Pri tem pa ti nobena naloga ne more prinesti negativnega števila točk. Če nisi pri nalogi oddal nobenega programa, ti ne prinese nobenih točk. Če se poslana izvorna koda ne prevede uspešno, to ne šteje kot oddaja.

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

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

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

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

Primer vhodne datoteke:

```
123
```

Ustrezna izhodna datoteka:

```
1230
```

Primer rešitve:

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

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

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

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

1. tekmovanje IJS v znanju računalništva za srednješolce

6. maja 2006

NALOGE ZA TRETJO SKUPINO

Rešitve bodo objavljene na `http://rtk.ijs.si/`.

1. Optična miška (`miska.in`, `miska.out`)

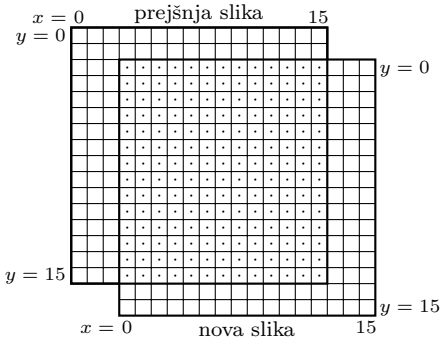
Optične računalniške miške, ki ne potrebujejo posebne podlage, vsebujejo preprosto, a hitro videokamero, ki si stalno ogleduje del podlage pod miško, vgrajeni procesor pa zaporedne slike primerja med seboj. Glede na to, da uporabnik premika miško dokaj počasi in neskokovito glede na kratek čas med zaporednima slikama (pod tisočinko sekunde), sta si zaporedni sliki precej podobni, naslednja slika je kvečjemu nekoliko premaknjena glede na prejšnjo (sukanje lahko zanemarimo, dviga miške s podlage pri tej nalogi ne bomo predvideli).

Vsaka slika je sestavljena iz kvadratne mreže $n \times n$ slikovnih elementov (pikslov). Ker zajema kamera le sivinske slike, je vsak slikovni element predstavljen s svetlostjo, ki je celo število med 0 in 63. Točka s koordinatama $(0, 0)$ je v zgornjem levem vogalu slike. Predpostavimo, da so premiki miške med dvema zaporednima slikama veliki kvečjemu tri piksle v vsaki smeri ($-3 \leq dx \leq 3$, $-3 \leq dy \leq 3$).

S primerjanjem dveh zaporednih slik bi radi ocenili, kakšen je najverjetnejši premik miške v času med trenutkoma, ko sta bili sliki posneti.

Nek možni premik (dx, dy) ocenimo takole: za vsak slikovni element, ki bi bil pri tem premiku viden tako na prejšnji kot na trenutni sliki, izračunamo absolutno vrednost razlike v svetlosti te točke na prejšnji in na trenutni sliki. Nato izračunamo povprečje teh absolutnih vrednosti po vseh slikovnih elementih, ki so vidni tako na prejšnji kot na trenutni sliki. Manjše ko je to povprečje, bolj verjetno je, da je (dx, dy) res pravi premik.

Na primer: če delamo s slikami velikosti 16×16 in se miška premakne za $dx = 3$, $dy = 2$, je prejšnji in trenutni sliki skupnih $13 \cdot 14$ slikovnih elementov in po njih bi računali povprečje iz gornjega odstavka. Na spodnji ilustraciji so ti skupni slikovni elementi označeni s pikami.



Napiši program, ki bo prebral nekaj parov zaporednih slik iz vhodne datoteke in za vsak par izpisal najverjetnejši premik miške v horizontalni in vertikalni smeri.

Vhodna datoteka: v prvi vrstici je celo število m ($0 < m \leq 30$), ki pove, da v tem testnem primeru nastopa m parov slik. Sledijo podatki o posameznih parih slik, pred vsakim od njih pa je prazna vrstica.

Za vsak par slik je v vhodni datoteki najprej vrstica, ki vsebuje celo število n ($4 \leq n \leq 64$), ki pove, da sta sliki v tem paru veliki $n \times n$ slikovnih elementov. Sledi prazna vrstica, nato pa n vrstic, ki opisujejo sliko pred premikom (od zgoraj navzdol: prva

vrstica je za $y = 0$, zadnja za $y = n - 1$); v vsaki od teh vrstic je n celih števil (med vključno 0 in vključno 63), ki povedo svetlost slikovnih elementov te vrstice slike (od leve proti desni: prvo število je za $x = 0$, zadnje za $x = n - 1$). Nato pride še ena prazna vrstica in nato še nadaljnjih n vrstic, ki na enak način podajajo drugo sliko (tisto po premiku miške).

Izhodna datoteka: za vsak par slik poišči zamik (dx, dy), pri katerem je dosežena najmanjša vrednost zgoraj opisane ocene (povprečna razlika absolutnih vrednosti razlik svetlosti tistih slikovnih elementov, ki so skupni stari in novi sliki). Če je ta najmanjša vrednost dosežena pri več zamikih, je vseeno, katerega izpišeš. Veljavni zamiki so le tisti, pri katerih sta dx in dy celi števili in velja $-3 \leq dx \leq 3$, $-3 \leq dy \leq 3$. Najdeni zamik za vsak par slik izpiši v svojo vrstico (najprej dx, nato dy, vmes pa presledek) in to v enakem vrstnem redu, v kakršnem so pari slik podani v vhodni datoteki. Vmes ne izpisuj praznih vrstic ali česa podobnega.

Točkovanje: če izhodna datoteka ni oblikovana tako, kot je opisano v prejšnjem odstavku, dobi tvoj program pri tem testnem primeru 0 točk. Drugače pa, če je od m premikov pravilno prepoznal k premikov, dobi pri tem testnem primeru $(10 \cdot k)$ div m točk.

Primer vhodne datoteke:

```
2

4

56 52 47 40
57 48 43 39
56 50 44 36
56 50 41 35

53 50 45 40
56 54 49 46
52 46 42 34
51 45 38 32

5

53 42 33 26 17
45 36 30 23 13
37 30 23 20 12
27 25 19 15 12
19 16 12 11 9

36 45 43 36 28
35 46 36 31 23
37 47 27 24 18
38 47 19 16 13
39 45 9 9 9
```

Pripadajoča izhodna datoteka:

```
1 -2
-2 1
```

Pri prvem paru slik lahko na primer vidimo, da se slikovni elementi

```
52 47 40
48 43 39
```

na stari sliki

precej lepo ujemajo z elementi

```
52 46 42
51 45 38
```

na novi sliki.

Podobno se pri drugem paru lepo ujemajo

```
45 36 30
37 30 23
27 25 19
19 16 12
```

na stari in

```
43 36 28
36 31 23
27 24 18
19 16 13
```

na novi sliki.

2. Spletne knjigarne (`knjigarne.in`, `knjigarne.out`)

Prek spletnih knjigarn bi radi kupili po en izvod vsake izmed n knjig. Na voljo imamo k spletnih knjigarn; posamezno knjigo lahko kupimo od katerekoli od njih, vendar pa so lahko cene knjig v različnih knjigarnah različne. Plačati moramo tudi dostavo knjig na naš domači naslov. Knjigarna i ($1 \leq i \leq k$) računa a_i denarnih enot za dostavo prve naročene knjige in b_i za dostavo vsake naslednje naročene knjige (če iz neke knjigarne ne naročimo nobene knjige, ji seveda ni treba plačati ničesar). Vedno velja $a_i \geq b_i$. Cena knjige j ($1 \leq j \leq n$) pri knjigarni i je c_{ji} denarnih enot.

Skupna cena, ki jo bomo morali plačati za želenih n knjig, je torej v splošnem lahko odvisna od tega, pri kateri knjigarni bomo naročili katero knjigo. **Napiši program**, ki izračuna najmanjšo skupno ceno knjig (s stroški dostave vred), za katero lahko nakupimo vseh n knjig.

Vhodna datoteka: v prvi vrstici sta celi števili n in k, ločeni s presledkom. Zanju velja $1 \leq n \leq 10000$ in $1 \leq k \leq 10$. Sledi še $n + 2$ vrstic; v vsaki je po k celih števil, ločenih s po enim presledkom. V prvi od teh vrstic so cene $a_1, a_2, \ldots, a_k$; v drugi so cene $b_1, b_2, \ldots, b_k$; v ostalih vrsticah pa so cene posamezne knjige v vseh k knjigarnah: najprej je tu vrstica s cenami $c_{11}, c_{12}, \ldots, c_{1k}$ (cena prve knjige v vseh k knjigarnah), nato vrstica s cenami $c_{21}, c_{22}, \ldots, c_{2k}$ (cena prve knjige v vseh k knjigarnah) in tako naprej. Zadnja vrstica torej vsebuje cene $c_{n1}, c_{n2}, \ldots, c_{nk}$. Vse cene, a_i, b_i, c_{ji} za vse i od 1 do k in vse j od 1 do n, so cela števila, večja ali enaka 1 in manjša ali enaka 10000.

Izhodna datoteka: vanjo izpiši celo število, ki je najnižja skupna cena (s stroški dostave vred), za katero je mogoče dobiti vseh n knjig.

Primer vhodne datoteke:

```
10 3
8 10 10
5 7 10
10 8 11
10 9 12
9 9 12
8 7 11
9 9 12
9 9 14
9 9 10
9 10 11
8 8 11
10 8 3
```

Pripadajoča izhodna datoteka:

```
142
```

Za ta denar lahko pridemo do knjig, če kupimo vse razen zadnje knjige v prvi knjigarni, zadnjo pa v tretji.

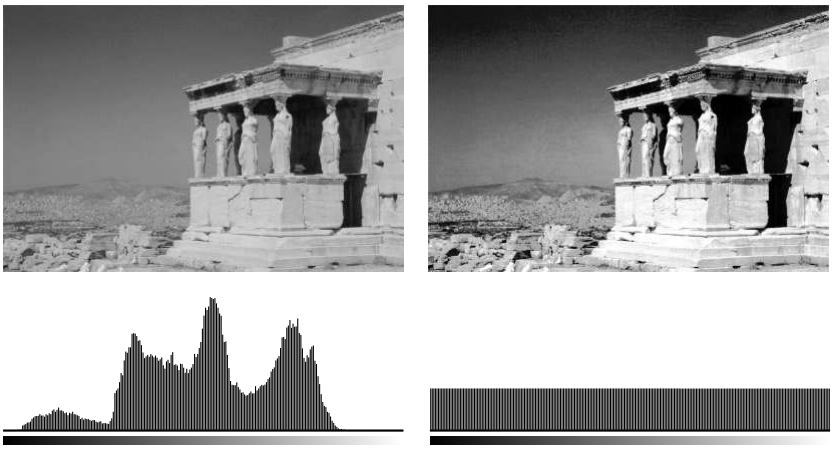
3. Izravnavanje histogramov (`histogrami.in`, `histogrami.out`)

Imamo sivinsko sliko, široko w slikovnih elementov (pikslov) in visoko h slikovnih elementov. Barva posameznega slikovnega elementa je opisana z enim od celih števil od 0 do 255 (večja ko je, svetlejši je ta slikovni element — 0 pomeni črno, 255 pa belo barvo).

Če za vsako svetlost od 0 do 255 preštejemo, koliko slikovnih elementov na sliki ima ravno to svetlost, in ta števila predstavimo s stolpci, dobimo *histogram* naše slike.

Izravnavanje histograma (*histogram equalization*) je postopek, pri katerem sliko popravimo tako, da postane njen histogram čim bolj „raven“ — torej da so vsi stolpci približno enako visoki.

Primer slike pred in po izravnavanju histograma (pod vsako različico slike je narisan tudi histogram):



Napiši program, ki prebere vhodno sliko I in izpiše izhodno sliko I' (enake velikosti kot I), za katero velja:

- Število slikovnih elementov najpogostejše barve na sliki I' je kvečjemu za 1 večje od števila slikovnih elementov najredkejše barve (tiste, ki ji pripada najmanj slikovnih elementov).
- Za vsak par slikovnih elementov (x_1, y_1) in (x_2, y_2) velja: če je slikovni element (x_1, y_1) na sliki I svetlejši od slikovnega elementa (x_2, y_2), je na sliki I' slikovni element (x_1, y_i) svetlejši ali pa enake barve kot (x_2, y_2), ni pa temnejši.

Če obstaja več slik I', ki ustrezajo gornjima pogojema, je vseeno, katero izpišeš.

Vhodna datoteka: v prvi vrstici sta celi števili h (višina slike) in w (širina slike), ločeni s presledkom. Obe sta večji ali enaki 1 in manjši ali enaki 200. Sledi h vrstic, v vsaki od njih pa je w celih števil, ločenih s po enim presledkom; i-to število v j-ti vrstici pove barvo slikovnega elementa na preseku i-tega stolpca in j-te vrstice slike. Vse barve so večje ali enake 0 in manjše ali enake 255.

Izhodna datoteka: vanjo zapiši h vrstic, v vsaki od njih pa naj bo w celih števil, ločenih s po enim presledkom. Ta števila naj opisujejo barve slikovnih elementov izhodne slike na enak način, kot je opisana vhodna slika v vhodni datoteki. Izhodna slika mora ustrezati zahtevam, navedenim v besedilu naloge.

Primer vhodne datoteke:

```
5 5
105 105 99 101 101
104 102 103 104 103
105 9 103 105 103
9 105 105 101 102
102 9 103 101 101
```

Ena od možnih pripadajočih izhodnih datotek:

```
19 20 3 4 5
17 9 12 18 13
21 0 14 22 15
1 23 24 6 10
11 2 16 7 8
```

4. Mafija (`mafija.in`, `mafija.out`)

V neki mafijski družini so člani urejeni hierarhično: vsakdo razen vrhovnega šefa ima natanko enega nadrejenega človeka in če sledimo povezavam od podrejenega do nadrejenega, lahko od vsakega člana družine sčasoma pridemo do vrhovnega šefa.

Denar v družini kroži takole. Tisti, ki nimajo nobenih podrejenih, morajo zbirati denar z raznimi kriminalnimi dejavnostmi in ves tako prisluženi denar oddati svojemu nadrejenemu. Ta mora ves denar, ki ga prejme od podrejenih, oddati *svojemu* nadrejenemu, on ga pošlje spet svojemu nadrejenemu in tako naprej, vse dokler ves denar ne konča v rokah vrhovnega šefa.

Vendar pa šef sumi, da nekateri člani družine goljufajo in ne pošljejo naprej svojemu nadrejenemu vsega denarja, ki so ga prejeli od svojih podrejenih. Šef je z izsiljevanjem in vohunjenjem uspel od vsakega člana pridobiti podatek o tem, koliko denarja je poslal svojemu nadrejenemu, zdaj pa prosi tebe, da mu **napišeš program**, ki bo ugotovil, kolikšna je največja vsota denarja, ki jo je utajil kakšen od članov družine (torej največja razlika med tem, koliko denarja je prejel od podrejenih, in tem, koliko ga je posredoval svojemu nadrejenemu). Za tiste, ki nimajo nobenega podrejenega, predpostavimo, da niso utajili nič svojega zaslužka (ker ne znamo oceniti, koliko denarja so v resnici zbrali s svojimi kriminalnimi dejavnostmi).

Vhodna datoteka: v prvi vrstici je celo število n ($0 < n \leq 100000$), ki pove, koliko članov ima družina. Člani družine so oštevilčeni s celimi števili od 1 do n. Sledi še n vrstic datoteke, ki za vsakega člana družine povedo, kdo je njegov nadrejeni in koliko denarja je ta član posredoval temu svojemu nadrejenemu. Tako torej $(i+1)$-va vrstica datoteke vsebuje dve celi števili, ločeni s presledkom; prvo od teh števil je številka člana, ki je neposredno nadrejen članu i, drugo pa je znesek denarja, ki ga je i oddal svojemu nadrejenemu (ta znesek je večji ali enak 0, obenem pa manjši ali enak vsoti zneskov, ki jih je član i prejel od svojih podrejenih). Pri vrhovnem šefu, ki nadrejenega nima, sta navedeni dve ničli. Vsota zneskov, ki jih posamezni član družine prejme od svojih podrejenih, pri nobenem članu ne presega 10^9 denarnih enot.

Izhodna datoteka: za vsakega člana (razen vrhovnega šefa) izračunaj razliko med zneskom, ki ga je prejel od podrejenih, in zneskom, ki ga je oddal nadrejenemu. Izpiši največjo vrednost te razlike po vseh članih.

Primer vhodne datoteke:

```
8
3 100
3 50
4 120
0 0
8 30
8 40
8 50
4 95
```

Pripadajoča izhodna datoteka:

```
30
```

Član 3 je od svojih podrejenih (1 in 2) prejel 150 denarnih enot, oddal pa 120, torej jih je utajil 30. Član 8 pa je od svojih podrejenih (5, 6 in 7) prejel 120 denarnih enot, oddal pa jih je 95, torej jih je utajil 25.

5. Tovornjaki (`tovornjaki.in`, `tovornjaki.out`)

Gneče na trajektih so običajen pojav in Jadrolinija vlaga veliko naporov v to, da bi jih zmanjšala. Običajno na obali stoji kolona tovornjakov in potrebno je čimprej ugotoviti, kako velik trajekt potrebujemo, da jih lahko naekrat prepeljemo.

Dano imamo torej zaporedje tovornjakov, ki bi jih radi razporedili na trajekt. Na trajektu bodo tovornjaki stali v treh pasovih in mi si lahko poljubno izberemo, na katerem pasu bo stal kateri tovornjak. Seveda pa skupna dolžina tovornjakov na nobenem od pasov ne sme presegati dolžine trajekta. **Napiši program**, ki bo za dano zaporedje dolžin tovornjakov poskusil ugotoviti, kako dolg je najkrajši trajekt, pri katerem bi se še dalo razporediti te tovornjake v tri pasove, ne da bi skupna dolžina tovornjakov na katerem od pasov presegla dolžino trajekta.

Vhodna datoteka: v prvi vrstici je celo število n ($1 \leq n \leq 100$), ki pove število tovornjakov. Sledi še n vrstic, v katerih so dolžine tovornjakov. Vsaka dolžina tovornjaka je celo število, večje ali enako 1 in manjše ali enako 100.

Izhodna datoteka: V prvo vrstico izpiši dolžino najkrajšega trajekta, na katerega je tvojemu programu uspelo razporediti tovornjake iz vhodne datoteke v skladu z zahtevami naloge. (Testni primeri bodo zasnovani tako, da ta dolžina najkrajšega trajekta nikoli ne bo večja od 100.) Sledi naj še n vrstic, ki opišejo nek konkreten razpored tovornjakov na ta najkrajši trajekt: v i-ti izmed teh vrstic naj bo število 1, 2 ali 3, ki pove, na kateri pas je bil razporejen i-ti tovornjak iz vhodne datoteke.

Točkovanje: če tvoja izhodna datoteka ne ustreza zgoraj opisanim zahtevam, dobiš pri tistem testnem primeru 0 točk, drugače pa je število točk odvisno od tega, kako dolg je tvoj trajekt v primerjavi z najkrajšim možnim. Recimo, da je pri nekem testnem primeru najkrajši možni trajekt dolg t^* enot, v tvoji izhodni datoteki pa je trajekt dolžine t enot. Potem dobiš pri tem testnem primeru $\max\{1, 10 - (t - t^*)\}$ točk. Z drugimi besedami, 10 točk dobiš le, če je tvoja rešitev najboljša možna ($t = t^*$); sicer pa se ti odbije toliko točk, za kolikor je tvoj trajekt daljši od najkrajšega možnega, vendar največ devet točk (tako da zagotovo dobiš vsaj eno točko).

Primer vhodne datoteke:

```
7
26
15
29
30
30
16
10
```

Pripadajoča izhodna datoteka:

```
55
1
2
1
2
3
3
2
```

Na trajekt dolžine 55 lahko razporedimo tovornjake takole: $26 + 29$ na en pas, $10 + 15 + 30$ na drugega in $16 + 30$ na tretjega. To je pri tem zaporedju tovornjakov tudi najkrajši možni trajekt.

1. tekmovanje IJS v znanju računalništva za srednješolce

6. maja 2006

REŠITVE NALOG ZA PRVO SKUPINO

1. Odstavki

V neki spremenljivki (v spodnji rešitvi je to PrejPrazna) si zapomnimo, če je bila prejšnja vrstica prazna ali ne. Na začetku postavimo PrejPrazna na false, ker bi se drugače naš program obnašal tako, kot da je pred prvo zares prebrano vrstico še neka prazna vrstica; če bi bila potem prva vrstica tudi prazna, bi jo program pobrisal, ker bi mislil, da je to zdaj že druga prazna vrstica zaporedoma (v resnici pa ni in je ne bi smel pobrisati).

Kakorkoli že, če sta trenutna in prejšnja vrstica prazni, trenutne vrstice ne bomo izpisovali. To bo zagotovilo, da bomo od vsake skupine dveh ali več zaporednih praznih vrstic izpisali le prvo, ostale pa preskočili in tako izpolnili zahtevo naloge. Potem pa si podatek o tem, ali je bila trenutna vrstica prazna ali ne, zapomnimo v spremenljivki PrejPrazna, saj bomo ta podatek potrebovali v naslednji iteraciji zanke, ko bomo prebrali naslednjo vrstico.

```
program Odstavki;
var S: string; PrejPrazna: boolean;
begin
  PrejPrazna := false;
  while not Eof do begin
    ReadLn(S);
    if not ((S = '') and PrejPrazna) then WriteLn(S);
    PrejPrazna := S = '';
  end; {while}
end. {Odstavki}
```

2. Sneg

V neki spremenljivki (v spodnji rešitvi je to Debelina) hranimo trenutno debelino snežne odeje. Za vsak dan moramo prebrati podatek o tem, za koliko se poveča in za koliko zmanjša, potem pa lahko izračunamo novo debelino in jo izpišemo.

```
program Sneg;
var i, Debelina, Povecanje, Zmanjsanje: integer;
begin
  Debelina := 0;
  for i := 1 to 365 do begin
    ReadLn(Povecanje, Zmanjsanje);
    Debelina := Debelina + Povecanje - Zmanjsanje;
    WriteLn(Debelina);
  end; {for i}
end. {Sneg}
```

3. Sudoku

Lahko si pomagamo z množicami (**set** v pascalu). Za vsako vrstico izračunamo množico vseh števil v tej vrstici; če ta množica ni enaka [1..9], mora v tej vrstici kakšno od števil od 1 do 9 manjkati. Pogoja, da se v vrstici nobeno število ne sme pojavljati več kot enkrat, pa nam ni treba še dodatno preverjati: če se kakšno pojavlja več kot enkrat, mora kakšno drugo manjkati (ker imamo devet števil, v vrstici pa devet kvadratkov) in bomo dano polje zavrnili že zaradi tega. Enako kot za vrstice naredimo tudi za stolpce in za male kvadrate velikosti 3×3.

```
program Sudoku;
  type SudokuPoljeT = array [1..9, 1..9] of 1..9;
  procedure Preberi(var T: SudokuPoljeT); external;
var T: SudokuPoljeT; V, S, K: array [0..8] of set of 1..9; i, j, a: integer; Ok: boolean;
begin
  for i := 0 to 8 do begin V[i] := []; S[i] := []; K[i] := [] end;
  Preberi(T); Ok := true;
  for i := 0 to 8 do for j := 0 to 8 do begin
    a := T[i + 1, j + 1]; V[i] := V[i] + [a]; S[j] := S[j] + [a];
    K[(i div 3) * 3 + j div 3] := K[(i div 3) * 3 + j div 3] + [a];
  end; {for i}
  for i := 0 to 8 do if (V[i] <> [1..9]) or (S[i] <> [1..9]) or (K[i] <> [1..9]) then Ok := false;
  if Ok then WriteLn('PRAVILNA') else WriteLn('NAPACNA');
end. {Sudoku}
```

Namesto množic bi lahko tudi prižigali bite v kakšni celoštevilski spremeljivki ali pa bi uporabili tabelo devetih booleanov.

Še ena možnost pa je na primer ta, da si napišemo podprogram, ki preveri, če se neko število pojavi v vsaki vrstici, stolpcu in kvadratu 3×3. V spodnji rešitvi je to podprogram Preveri. Potem ga moramo le še poklicati po enkrat za vsako število od 1 do 9:

```
program Sudoku;
  type SudokuPoljeT = array [1..9, 1..9] of 1..9;
  procedure Preberi(var T: SudokuPoljeT); external;
  function Preveri(var T: SudokuPoljeT; n: integer): boolean;
  var i, j, x, y: integer; Nasel: boolean;
  begin
    Preveri := false;
    for i := 1 to 9 do begin      { Preverimo i-to vrstico in i-ti stolpec. }
      Nasel := false; for j := 1 to 9 do if T[i, j] = n then Nasel := true;
      if not Nasel then exit;
      Nasel := false; for j := 1 to 9 do if T[j, i] = n then Nasel := true;
      if not Nasel then exit;
    end; {for i}
    for i := 0 to 2 do for j := 0 to 2 do begin        { Preverimo mali kvadrat 3 × 3 }
      Nasel := false;                { z zgornjim levim kotom v (3 * i + 1, 3 * j + 1). }
      for y := 1 to 3 do for x := 1 to 3 do if T[3 * i + y, 3 * j + x] = n then Nasel := true;
      if not Nasel then exit;
    end; {for j, i}
    Preveri := true;
  end; {Preveri}

var n: integer; T: SudokuPoljeT; Ok: boolean;
begin {Sudoku}
  Ok := true; Preberi(T);
  for n := 1 to 9 do if not Preveri(T, n) then begin Ok := false; break end;
  if Ok then WriteLn('PRAVILNA') else WriteLn('NAPACNA');
end. {Sudoku}
```

4. Naraščajoče besede

Najdaljšo doslej znano naraščajočo besedo hranimo v spremenljivki Naj. Ko prebiramo besede, lahko tiste, ki niso daljše od Naj, kar preskočimo — tudi če so naraščajoče, očitno ne morejo biti najdaljše naraščajoče besede v celem vhodnem zaporedju. Za tiste besede pa, ki so dovolj dolge, preverimo, če so naraščajoče. S števcem i se sprehodimo po besedi in za vsako črko preverimo, če je v abecednem vrstnem redu za prejšnjo; če ni, se ustavimo. Če na ta način pridemo do konca besede, pomeni, da je vsaka črka po abecedi res za prejšnjo, torej je beseda naraščajoča. Ker smo že prej ugotovili, da je tudi daljša od Naj, si jo zapomnimo kot novo najdaljšo doslej znano naraščajočo besedo. Ko na ta način obdelamo celotno vhodno zaporedje, moramo le še izpisati Naj.

```
program NarascajoceBesede;
var Beseda, Naj: string; i: integer;
begin
  Naj := '';
  while not Eof do begin
    ReadLn(Beseda);
    if Length(Beseda) > Length(Naj) then begin
      i := 2;
      while i <= Length(Beseda) do
        if Beseda[i] > Beseda[i − 1] then i := i + 1 else break;
      if i > Length(Beseda) then Naj := Beseda;
    end; {if}
  end; {while}
  WriteLn('Najdaljša naraščajoča beseda: "', Naj, '"');
end. {NarascajoceBesede}
```

5. Podnapisi

Preprosta rešitev lahko vsakič pregleda vse podnapise in poišče ustreznega:

```
function PodnapisZaSlicico(StevilkaSlicice: integer): string;
var i: integer;
begin {PodnapisZaSlicico}
  for i := 1 to SteviloPodnapisov do
    if (StevilkaSlicice >= PrvaSlicica(i)) and (StevilkaSlicice <= ZadnjaSlicica(i)) then
      begin PodnapisZaSlicico := Vsebina(i); exit end;
  PodnapisZaSlicico := '';
end; {PodnapisZaSlicico}
```

Učinkovitejša rešitev upošteva dejstvo, da sličice filma ponavadi pregledujemo po vrsti; zato je pogosto dober kar isti podnapis kot ob zadnjem klicu naše funkcije; zapomnimo si ga v neki globalni spremenljivki. Če ni pravi ta, bo pa pogosto dober naslednji ali pa prejšnji (če predvajamo film nazaj); precej mogoče je tudi, da smo na eni od sličic med dosedanjim in naslednjim podnapisom (in tam podnapisa ni, saj jih imamo podane po vrsti). Če nič od tega ne uspe, lahko pravi podnapis poiščemo z bisekcijo.

```
var ZadnjiPodnapis: integer;

procedure Inicializacija;
begin
  ZadnjiPodnapis := 0;
end; {Inicializacija}

function PodnapisZaSlicico(StevilkaSlicice: integer): string;

  procedure Vrni(StPodnapisa: integer);
  begin
    ZadnjiPodnapis := StPodnapisa;
    PodnapisZaSlicico := Vsebina(StPodnapisa);
  end; {Vrni}

var L, R, M: integer;
begin {PodnapisZaSlicico}
  PodnapisZaSlicico := '';
  if ZadnjiPodnapis > 0 then
    if StevilkaSlicice < PrvaSlicica(ZadnjiPodnapis) then begin
      if ZadnjiPodnapis = 1 then exit
      else if StevilkaSlicice > ZadnjaSlicica(ZadnjiPodnapis − 1) then exit
      else if StevilkaSlicice >= PrvaSlicica(ZadnjiPodnapis − 1) then
        begin Vrni(ZadnjiPodnapis − 1); exit end;
    end else if StevilkaSlicice > ZadnjaSlicica(ZadnjiPodnapis) then begin
      if ZadnjiPodnapis = SteviloPodnapisov then exit
      else if StevilkaSlicice < PrvaSlicica(ZadnjiPodnapis + 1) then exit
      else if StevilkaSlicice <= ZadnjaSlicica(ZadnjiPodnapis + 1) then
        begin Vrni(ZadnjiPodnapis + 1); exit end
```

```
    end else
      begin Vrni(ZadnjiPodnapis); exit end
  L := 1; R := SteviloPodnapisov;
  while L < R do begin
    { Na tem mestu velja: če je treba na tej sličici sploh prikazati kakšen podnapis,
      je to eden od podnapisov L..R. }
    M := (L + R) div 2;
    if StevilkaSlicice < PrvaSlicica(M) then R := M - 1
    else if StevilkaSlicice > ZadnjaSlicica(M) then L := M + 1
    else begin Vrni(M); exit end;
  end; {while}
  if L <= R then if (StevilkaSlicice >= PrvaSlicica(L))
                 and (StevilkaSlicice <= ZadnjaSlicica(L)) then Vrni(L);
end; {PodnapisZaSlicico}
```

Prostorsko malo potratnejša rešitev (besedilo naloge pravzaprav že pravi, da je takšna poraba prostora nesprejemljiva) pa je ta, da si ob inicializaciji v neki tabeli za vsako sličico zapomnimo ustrezni indeks podnapisa. Potem je PodnapisZaSlicico res zelo hitra, saj mora le pogledati v pravi element te tabele.

```
type TabelaT = packed array [1..1000000] of integer;
var Tabela: ↑TabelaT; StSlicic: integer;

procedure Inicializacija;
var i, t: integer;
begin
  if SteviloPodnapisov <= 0 then begin StSlicic := 0; exit end;
  StSlicic := ZadnjaSlicica(SteviloPodnapisov);
  GetMem(Tabela, SizeOf(integer) * StSlicic);
  for t := 1 to StSlicic do Tabela↑[t] := 0;
  for i := 1 to SteviloPodnapisov do
    for t := PrvaSlicica(i) to ZadnjaSlicica(i) do Tabela↑[t] := i;
end; {Inicializacija}

function PodnapisZaSlicico(StevilkaSlicice: integer): string;
var i: integer;
begin
  i := 0;
  if (StevilkaSlicice >= 1) and (StevilkaSlicice <= StSlicic)
  then i := Tabela↑[StevilkaSlicice];
  if i = 0 then PodnapisZaSlicico := ''
  else PodnapisZaSlicico := Vsebina(i);
end; {PodnapisZaSlicico}
```

REŠITVE NALOG ZA DRUGO SKUPINO

1. 1337ovščina

Vhodno besedilo berimo znak za znakom. Naj bo c trenutni znak in s niz, v katerem so vse možne zamenjave tega znaka (ločene s po enim presledkom). Če je na primer možnih pet zamenjav, vsebuje niz s štiri presledke. Če torej presledke preštejemo in prištejemo 1, dobimo ravno število možnih zamenjav znaka c. (Poseben primer je, če je s prazen niz; tedaj možnih zamenjav pač ni. Če bi tu šteli presledke in prišteli 1, bi dobili napačen vtis, da je možna ena zamenjava.)

Zdaj se moramo nekako odločiti, katero od naših n možnih zamenjav bi uporabili. Skupaj z možnostjo, da znak c pustimo kar v prvotni obliki (ga ne zamenjamo z ničimer), imamo torej $n + 1$ možnosti. Spodnja rešitev poskuša ravnati tako, da bi bile vse te možnosti enako verjetne. S stavkom m := Nakljucje(n + 1) − 1 dobimo neko naključno število od 0 do n. Pri $m = n$ bomo pustili vhodni znak c pri miru, drugače pa ga bomo zamenjali z eno od možnih zamenjav, in sicer s tisto, ki v nizu s stoji med m-tim in $(m + 1)$-vim presledkom. Tako pri $m = 0$ izberemo prvo možno zamenjavo znaka c, pri

$m = 1$ drugo in tako naprej. Zdaj se moramo le še sprehoditi po nizu s, šteti presledke in izpisati pravo zamenjavo.

```
program Leet;
var c: char; s: string; i, n, m: integer;
begin
  while not Eof do
    if Eoln then begin ReadLn; WriteLn; end
    else begin
      Read(c);
      s := MozneZamenjave(c);
      if Length(S) = 0 then n := 0 else n := 1;
      for i := 1 to Length(s) do if s[i] = ' ' then n := n + 1;
      m := Nakljucje(n + 1) - 1;
      if m = n then Write(c)
      else for i := 1 to Length(s) do
        if s[i] = ' ' then if m = 0 then break else m := m - 1
        else if m = 0 then Write(s[i]);
    end; {if}
end. {Leet}
```

Učinkovitejša rešitev ne bi vsakič sproti prežvekovala celega niza s, štela presledkov v njem in tako naprej, pač pa to naredila na začetku za vse možne znake c; za vsakega bi si zapomnila, koliko možnih zamenjav ima, in posamezne zamenjave shranila v neko tabelo. Potem bi morali po vsakem branju nekega znaka c le izpisati naključno izbran element c-jeve tabele možnih zamenjav.

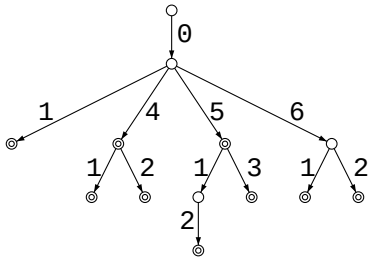
2. Predpone

Preprosta in ne najbolj učinkovita rešitev je, da gremo z zanko po vseh predponah in za vsako preverimo, če je to res predpona naše telefonske številke. Med tistimi, ki so, si zapomnimo najdaljšo. Spodnja rešitev primerja predpono z začetnimi znaki telefonske številke v notranji zanki **while**; če ta zazna neujemanje, se ustavi, če pa srečno pride skozi vseh DP znakov, kolikor jih je v trenutni predponi, vemo, da je to res predpona naše telefonske številke.

Da si prihranimo nekaj dela, si zapomnimo tudi dolžino najdaljše doslej znane predpone; če neka kasnejša predpona ni daljša od nje, jo lahko preskočimo, saj vemo, da četudi je to res predpona naše številke, gotovo ne bo postala nova najdaljša znana predpona. Podobno preskočimo tudi morebitne predolge predpone (daljše od cele telefonske številke).

```
function NajdaljsaPredpona(TelSt: string): string;
var i, pi, DP, DS, MaxD: integer; Predpona: string;
begin
  MaxD := 0; DS := Length(TelSt); NajdaljsaPredpona := '';
  for pi := 1 to StPredpon do begin
    Predpona := PovejPredpono(pi); DP := Length(Predpona);
    if (DP > DS) or (DP <= MaxD) then continue;
    i := 1; while i <= DP do if TelSt[i] = Predpona[i] then i := i + 1 else break;
    if i > DP then begin NajdaljsaPredpona := Predpona; MaxD := DP end;
  end; {for i}
end; {NajdaljsaPredpona}
```

Slabost te rešitve je, da mora iti pri vsaki telefonski številki po celem seznamu predpon. Če je predpon veliko, je to zamudno, pa še nepotrebno, saj večina predpon praviloma sploh ne nastopa na začetku naše telefonske številke (torej niso zares njene predpone). Hitreje bo šlo, če si ob inicializaciji vse predpone uredimo v drevo. Vsako vozlišče drevesa ima lahko do deset poddreves, ki ustrezajo posameznim števkam od 0 do 9. Vsaka pot od korena drevesa do nekega vozlišča tako ustreza nekemu zaporedju števk, v vozlišču pa imejmo podatek o tem, ali je to zaporedje števk tudi res ena od predpon ali ne. Primer drevesa za predpone 01, 04, 041, 042, 05, 0512, 053, 061 in 062 (z dvojnimi krožci so označena tista vozlišča, pri katerih se res konča neka predpona):



Podprogram NajdaljsaPredpona lahko zdaj jemlje števke z začetka dane telefonske številke in se spušča po drevesu s korena navzdol; vsakič gre v tisto poddrevo, na katerega kaže povezava, označena s trenutno števko. (Če take povezave ni, vemo, da ni nobene predpone, ki bi pokrivala ves doslej prebrani del naše telefonske številke, zato lahko nehamo.) Na vsakem koraku pogledamo, če se pri trenutnem vozlišču končuje kakšna predpona, in če se, si jo zapomnimo kot najdaljšo doslej znano predpono naše dane telefonske številke.

Drevo zgradi podprogram Inicializacija, ki začne s praznim drevesom (takim, ki vsebuje le koren) in potem vanj eno za drugo dodaja vse predpone. Za vsako predpono moramo poskrbeti, da obstaja v drevesu pot od korena navzdol, pri kateri se števke na povezavah zložijo ravno v to predpono. Če ustrezne povezave še ne obstajajo, bo pač treba dodati kakšno novo vozlišče.

```
type VozlisceP = ↑VozlisceT;
     VozlisceT = record
       JePredpona: boolean;
       Otroci: array [0..9] of VozlisceP;
     end; {VozlisceT}

var Koren: VozlisceP;

{ Ta podprogram ustvari novo vozlišče in ga doda med otroke vozlišča Oce (če Oce ni nil). }
function DodajVozlisce(Oce: VozlisceP; Stevka: integer): VozlisceP;
var V: VozlisceP; i: integer;
begin
  New(V); V↑.JePredpona := false;
  for i := 0 to 9 do V↑.Otroci[i] := nil;
  if Oce <> nil then Oce↑.Otroci[Stevka] := V;
  DodajVozlisce := V;
end; {DodajVozlisce}

procedure Inicializacija;
var i, j, Stevka: integer; Predpona: string; V: VozlisceP;
begin
  Koren := DodajVozlisce(nil, 0);
  for i := 1 to StPredpon do begin
    Predpona := PovejPredpono(i);
    V := Koren;
    for j := 1 to Length(Predpona) do begin
      Stevka := Ord(Predpona[j]) - Ord('0');
      if V↑.Otroci[Stevka] <> nil then V := V↑.Otroci[Stevka]
      else V := DodajVozlisce(V, Stevka);
    end; {for j}
    V↑.JePredpona := true;
  end; {for i}
end; {Inicializacija}

function NajdaljsaPredpona(TelSt: string): string;
var Predpona: string; V: VozlisceP; i: integer;
begin
  V := Koren; Predpona := ''; NajdaljsaPredpona := '';
  for i := 1 to Length(TelSt) do begin
    V := V↑.Otroci[Ord(TelSt[i]) - Ord('0')];
    if V = nil then break;
```

```
      Predpona := Predpona + TelSt[i];
      if V↑.JePredpona then NajdaljsaPredpona := Predpona;
    end; {for i}
end; {NajdaljsaPredpona}
```

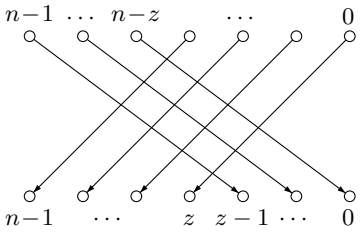
3. Cestne lučke

Podprogram Ugasni mora le ugasniti vse lučke, torej postaviti stanje na 0. Podprogram Utripaj lahko v neskončni zanki prižiga in ugaša lučke, vmes pa čaka; ko bo treba z utripanjem prenehati, ga bo že prekinil sistem (tako obljublja besedilo naloge.)

```
procedure Ugasni;
begin
  Nastavi(0);
end; {Ugasni}

procedure Utripaj;
begin
  while true do begin
    Nastavi(UTRIPA); Pocakaj;
    Nastavi(0); Pocakaj;
  end; {while}
end; {Utripaj}
```

Nekaj več dela pa je s potovanjem lučk v levo in desno. Naj bo n število vseh lučk; mislimo si neko n-bitno število, v katerem vsak bit pove stanje ene lučke (bit 0 predstavlja skrajno desno lučko, bit $n-1$ pa skrajno levo). Kako se to število spremeni, če ciklično zamaknemo vzorec na lučkah za z mest v levo? Tule je primer za $n = 7$, $z = 3$:



Vidimo torej, da se vsebina spodnjih $n - z$ bitov (od 0 do $n - z - 1$) premakne v bite od z do $n - 1$, vsebina zgornjih z bitov (od $n - z$ do $n - 1$) pa se premakne v bite od 0 do $z - 1$.

Do spodnjih nekaj bitov nekega števila x lahko pridemo s pomočjo operatorja **and**. Tako na primer z izrazom x **and** 1111_2 dobimo spodnje štiri bite števila x, ostali pa so ugasnjeni (ker so bili ugasnjeni tudi v desnem operandu, 1111_2). V splošnem, če hočemo dobiti spodnjih k bitov števila x, moramo x z**and**ati z operandom, ki ima spodnjih k bitov prižganih, ostale pa ugasnjene. Če bi tako število $111\ldots1_2$ povečali za 1, vidimo, da pride povsod do prenosa in dobili bi rezultat $10\ldots0_2$ (enica in k ničel), to pa je ravno 2^k. Naš operand $111\ldots1_2$ je torej enak $2^k - 1$. Do števila 2^k pa lahko pridemo z izrazom oblike 1 **shl** k. Na ta način si pripravi spodnji podprogram v spremenljivki Spodnji spodnjih $n - z$ bitov števila POTUJE.

Do zgornjih z bitov pridemo še enostavneje — število POTUJE preprosto zamaknemo za $n - z$ bitov v desno (biti od n-tega naprej so bili, kot pravi naloga, v konstanti POTUJE tako ali tako ugasnjeni in se nam zdaj ni treba posebej ubadati s tem, kako bi jih ugasnili). Tako dobimo v spodnjih z bitih spremenljivke Zgornji vsebino zgornjih z bitov konstante POTUJE.

Zdaj nam ostane le še, da spodnjih $n - z$ bitov prvotnega vzorca POTUJE (ki so zdaj v Spodnji) zamaknemo za z mest v levo in na tako izpraznjena mesta bite postavimo vsebino zgornjih z bitov vzorca POTUJE (ki so zdaj na spodnjih z bitih spremenljivke Zgornji). To potem ponavljamo v neskončni zanki in zamik bodisi povečujemo (če hočemo premikanje v desno) ali pa zmanjšujemo (če hočemo premikanje v levo).

```
procedure PotujoceLucke(Smer: integer);
var Zamik, Spodnji, Zgornji: integer;
begin
  Zamik := 0;
  while true do begin
    { Spodnjih SteviloLuck − Zamik bitov. }
    Spodnji := POTUJE and ((1 shl (SteviloLuck − Zamik)) − 1);
    { Zgornjih Zamik bitov. }
    Zgornji := POTUJE shr (SteviloLuck − Zamik);
    { Nastavimo novo stanje. Zgornjih Zamik bitov pride na dno. }
    Nastavi((Spodnji shl Zamik) or Zgornji); Pocakaj;
    { Premaknimo Zamik. }
    Zamik := Zamik + Smer;
    if Zamik < 0 then Zamik := SteviloLuck − 1;
    if Zamik = SteviloLuck then Zamik := 0;
  end; {while}
end; {PotujoceLucke}

procedure VLevo; begin PotujoceLucke(1) end;
procedure VDesno; begin PotujoceLucke(−1) end;
```

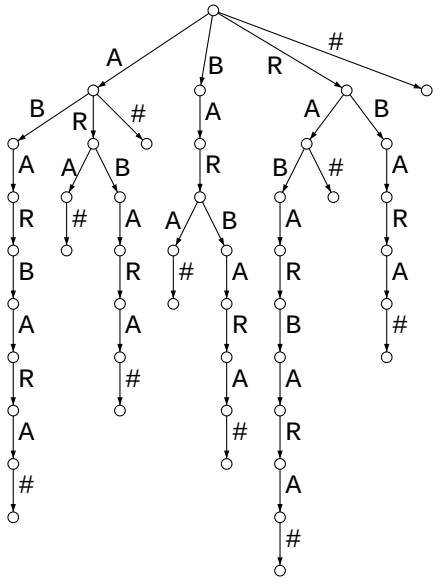
Gornja rešitev uporablja nestandardne razširitve pascala (ki so sicer v dandanašnjih narečjih zelo razširjene): **and** in **or** uporablja kot aritmetična operatorja (na bitih), medtem ko sta v standardnem pascalu definirana le kot logična operatorja; operatorjev **shl** in **shr** pa standardni pascal sploh ne pozna.

Oglejmo si zdaj še primer rešitve, ki ne uporablja takšnih nestandardnih razširitev. Namesto tega si bomo pomagali z množenjem in deljenjem s potencami števila 2. Če število delimo z 2^k, ga v bistvu zamaknemo za k bitov v desno; če ga množimo z 2^k, ga s tem zamaknemo za k bitov v levo. Če izračunamo njegov ostanek po deljenju z 2^k, pa dobimo pravzaprav spodnjih k bitov prvotnega števila. Tako lahko premikamo kose nekega števila za določeno število mest v levo in desno.

```
procedure PotujoceLucke(Smer: integer);
var Stanje, i, m: integer;
begin
  Stanje := POTUJE; m := 1;
  for i := 1 to SteviloLuck − 1 do m := m * 2;
  while true do begin
    Nastavi(Stanje);
    if Smer < 0 then { v desno }
      Stanje := m * (Stanje mod 2) + (Stanje div 2)
    else { v levo }
      Stanje := 2 * (Stanje mod m) + (Stanje div m);
    Pocakaj;
  end; {while}
end; {PotujoceLucke}
```

4. Drevo končnic

(a) Drevo končnic za niz `RABARBARA` je takšno:



(b) Naj bo v neko vozlišče našega drevesa končnic; označimo število končnic, ki se končujejo v poddrevesu s korenom pri v (v to poddrevo šteje tudi vozlišče v sámo), s $f(v)$. To je ravno enako številu listov v tem poddrevesu. Vrednost $f(v)$ lahko za vsa vozlišča izračunamo tako, da rekurzivno pregledamo drevo od spodaj navzgor in seštevamo vrednosti f-ja po otrocih vsakega vozlišča (v listih pa je $f(v) = 1$).

Mislimo si zdaj nek niz $p = p_1 p_2 \dots p_t$, ki se pojavlja r-krat kot podniz niza s. To pomeni, da za r različnih začetnih indeksov $i_1, i_2, \dots, i_r$ velja $s_{i_j} s_{i_j+1} \dots s_{i_j+t-1} = p_1 \dots p_t$. To pa pomeni, da se vse končnice $s_{i_j} s_{i_j+1} \dots s_n$ (za $j = 1, \dots, r$) začnejo na niz p. Z drugimi besedami: število pojavitev p-ja kot podniza v s-ju je ravno enako številu s-jevih končnic, ki se začnejo na p. Te končnice pa bomo našli tako, da se po našem drevesu sprehodimo od korena navzdol in sledimo povezavam, kot jih določajo črke p-ja. Če v nekem trenutku ni primerne povezave, pomeni, da se sploh nobena s-jeva končnica ne začne na p, zato se p v s-ju sploh ne pojavlja. Če pa se naš sprehod ustavi v nekem vozlišču v, vemo, da se vsaka končnica s-ja, ki se končuje nekje v v-jevem poddrevesu (in nobena, ki se ne nahaja v njem!), začenja na niz p; takih končnic je, kot vemo, $f(v)$, to pa je potemtakem tudi število pojavitev p-ja v s-ju.

Če torej hočemo najti vse nize, ki se pojavljajo v s-ju vsaj k-krat, se moramo sprehajati od korena drevesa navzdol v vse smeri, dokler ne pridemo do kakšnega vozlišča, ki ima $f(v) < k$; tam se ustavimo. Za vsa vozlišča, ki pa imajo $f(v) \geq k$, pa lahko izpišemo niz, ki ga tvorijo oznake povezav od korena do vozlišča v. No, preden ga izpišemo, preverimo še, če je ta niz dolg vsaj m znakov, saj naloga pravi, da nas krajši nizi ne zanimajo.

Opisani postopek prikazuje spodnji podprogram, ki pa ne predpostavlja, da so vrednosti $f(v)$ že izračunane in shranjene v drevesu; zato jih sproti računa sam, kar pa med drugim pomeni, da mora v vsakem primeru pregledati celotno drevo.

```
const MaxOtrok = 5;
type VozlisceP = ↑VozlisceT;
     VozlisceT = record
       Oznaka: char; { oznaka na povezavi, ki kaže na to vozlišče }
       StOtrok: integer; { število otrok }
       Otroci: array [1..MaxOtrok] of VozlisceP; { kazalci na otroke }
     end; {VozlisceT}

procedure IzpisiPogostePodnize(Koren: VozlisceP; MinPogostost, MinDolzina: integer);

  function Rekurzija(V: VozlisceP; PotDoslej: string): integer;
  var i, StKoncnic: integer; Otrok: VozlisceP;
  begin
    if V↑.StOtrok = 0 then begin Rekurzija := 1; exit end;
    StKoncnic := 0;
```

```
    for i := 1 to V↑.StOtrok do begin
      Otrok := V↑.Otroci[i];
      StKoncnic := StKoncnic + Rekurzija(Otrok, PotDoslej + Otrok↑.Oznaka);
    end; {for i}
    if StKoncnic >= MinPogostost then
      if Length(PotDoslej) >= MinDolzina then WriteLn(PotDoslej, ' (', StKoncnic, ')');
    Rekurzija := StKoncnic;
  end; {Rekurzija}

begin {IzpisiPogostePodnize}
  if Koren <> nil then Rekurzija(Koren, '');
end; {IzpisiPogostePodnize}
```

V gornji rešitvi podprogram Rekurzija pregleda poddrevo, ki se začne v vozlišču V, in vrne število listov v njem. V parametru PotDoslej mu je treba podati niz, dobljen s stikanjem oznak vseh povezav na poti od korena do vozlišča V. Če je V list, vemo, da je oznaka na povezavi do njega kar posebni znak #, torej se PotDoslej tudi konča na # in je nima smisla izpisovati, saj to ni podniz tistega prvotnega niza, ki uporabnika zares zanima (torej tistega pred dodajanjem znaka #). Če pa je V notranje vozlišče, moramo z rekurzivnimi klici ugotoviti, koliko listov je v njegovem poddrevesu; to pa nam tudi pove, kolikokrat se PotDoslej pojavi v prvotnem nizu. Če je to število pojavitev dovolj veliko in če je niz PotDoslej dovolj dolg, ga izpišemo.

5. Razvajeni zvezdniki

Naloga pravi, da je vsak zvezdnik navedel dve množici in kot pogoj za svoje sodelovanje zahteval, da v oddaji sodelujejo vsi zvezdniki iz prve množice in nobeden od zvezdnikov iz druge množice. Označimo pri zvezdniku i prvo od teh dveh množic z A_i in drugo z B_i.

Če bi radi v oddajo vključili zvezdnike $\{1, 2, \ldots, k\}$, iz gornjih omejitev sledi, da morajo biti v oddajo vključeni tudi vsi zvezdniki iz $A_1 \cup A_2 \cup \ldots \cup A_k$ (označimo to unijo z $\hat{A}_k$) in nobeden od zvezdnikov iz $B_1 \cup B_2 \cup \ldots \cup B_k$ (označimo to unijo z $\hat{B}_k$). Da bomo res lahko uporabili le zvezdnike od 1 do k in nobenih drugih, mora biti torej množica zahtevanih zvedznikov $\hat{A}_k$ pomnožica množice $\{1, \ldots, k\}$; množica „prepovedanih“ zvezdnikov $\hat{B}_k$ pa ne sme vsebovati nobenega od zvezdnikov iz $\{1, \ldots, k\}$.

Pogoj, da je $\hat{A}_k$ podmnožica množice $\{1, \ldots, k\}$, je enakovreden pogoju, da je največji element množice $\hat{A}_k$ manjši ali enak k. Podobno je pogoj, da $\hat{B}_k$ nima skupnih elementov z množico $\{1, \ldots, k\}$, enakovreden pogoju, da je najmanjši element množice $\hat{B}_k$ večji od k.

Naš postopek lahko pregleduje različne k-je v naraščajočem vrstnem redu. Ko se k povečuje, pridobivata množici $\hat{A}_k$ in $\hat{B}_k$ nove elemente, starih pa nikoli ne izgubljata; zato se najmanjši element $\hat{B}_k$ sčasoma zgolj povečuje. Zato, če pri nekem k opazimo, da je najmanjši element množice $\hat{B}_k$ manjši ali enak k, bo to veljalo tudi pri vseh večjih k (ker se bo tisti najmanjši element le še zmanjševal, k pa se bo povečeval); čim je torej ta pogoj prekršen, vemo, da niti prvih k zvezdnikov na otok ne more odpotovati skupaj, še toliko manj pa lahko zato potuje skupaj kakšna večja skupina zvezdnikov, ki vsebuje vseh prvih k zvezdnikov. Zato lahko v takem primeru naš postopek takoj prekinemo.

Če pa s tem pogojem ni težav, nam ostane le še pogoj, da je največji element množice $\hat{A}_k$ manjši ali enak k; če je to res, pomeni, da oddajo lahko organiziramo z zvezdniki $\{1, \ldots, k\}$. Med k-ji, ki ustrezajo temu pogoju, si zapomnimo največjega; to je rešitev naše naloge.

Zdaj tudi vidimo, da množic $\hat{A}_k$ in $\hat{B}_k$ sploh ni treba shranjevati v celoti, ampak je dovolj že, če si zapomnimo največji element $\hat{A}_k$ (MaxZaht v spodnjem programu) in najmanjši element $\hat{B}_k$ (MinPrep v spodnjem programu).

```
var z, k, MaxK, MaxZaht, MinPrep: integer;
begin
  MaxK := 0; MaxZaht := 0; MinPrep := n + 1;
  for k := 1 to n do begin
    za vsakega z ∈ B_k do if z < MinPrep then MinPrep := z;
    if MinPrep <= k then break;
```

```
    za vsakega z ∈ A_k do if z > MaxZaht then MaxZaht := z;
    if MaxZaht <= k then MaxK := k;
  end; {for k}
  WriteLn(MaxK);
end.
```

REŠITVE NALOG ZA TRETJO SKUPINO

1. Optična miška

Ker so naše slike majhne, je dovolj dobra že najpreprostejša rešitev: pregledamo vse možne zamike, torej vse pare (dx, dy), za katere sta dx in dy med -3 in $+3$ (takih parov je $7 \times 7 = 49$) in pri vsakem pregledamo vse slikovne elemente, ki so pri tem zamiku vidni na obeh slikah, stari in novi. Točka, ki jo na stari sliki vidimo na koordinatah (x, y), je na novi sliki vidna na koordinatah $(x - dx, y - dy)$. Na primer: če se miška premakne v desno ($dx > 0$), se stvari na sliki premaknejo v levo, torej k manjšim x-koordinatam; zato tisto, kar smo prej videli pri x, zdaj vidimo $x - dx$. Podobno je s premiki v navpični smeri.

V poštev pridejo seveda le tisti (x, y) s stare slike, ki so pri trenutnem zamiku tudi zares vidni na novi sliki. Za koordinati na novi sliki, $(x - dx, y - dy)$, mora torej veljati: $0 \le x - dx < n$ in $0 \le y - dy < n$, če imamo opravka s slikami velikosti $n \times n$. Zanimale nas bodo torej x-koordinate od $\max\{0, -dx\}$ do $\min\{n - 1, n + dx - 1\}$, podobno pa je tudi pri y-koordinatah.

Pri vsakem slikovnem elementu, ki ga vidimo na obeh slikah, izračunajmo absolutno vrednost razlike v svetlosti tega elementa na obeh slikah. Na koncu nas bo zanimalo povprečje teh absolutnih vrednosti po vseh slikovnih elementih, ki se jih vidi na obeh slikah. Da ne bomo imeli težav s kakšnimi zaokrožitvenimi napakami, povprečij ne bomo izračunali eksplicitno, ampak jih bomo hranili v obliki ulomka a/b, pri čemer je a vsota absolutnih vrednosti razlik, b pa število slikovnih elementov, ki so prisotni na obeh slikah. Če je najmanjše doslej znano povprečje a^*/b^*, moramo potem pri vsakem naslednjem zamiku preveriti, če je a/b mogoče manjše od a^*/b^*. Pogoj $a/b < a^*/b^*$ lahko preoblikujemo v $ab^* < a^*b$ in se tako izognemo deljenju.

```
program OpticnaMiska;
const cxMax = 64; cyMax = 64; dxMax = 3; dyMax = 3;
var cx, cy, dx, dy, dxNaj, dyNaj, x, y: integer; T, U: text;
    OcenaA, OcenaB, NajA, NajB, i, StPrimerov: integer;
    SP, SN: array [0..cyMax − 1, 0..cxMax − 1] of integer;
begin
  Assign(T, 'miska.in'); Reset(T); ReadLn(T, StPrimerov);
  Assign(U, 'miska.out'); Rewrite(U);
  while StPrimerov > 0 do begin
    ReadLn(T); StPrimerov := StPrimerov − 1;
    Read(T, cx); cy := cx;
    { Preberimo obe sliki. }
    ReadLn(T); for y := 0 to cy − 1 do
      begin for x := 0 to cx − 1 do Read(T, SP[y, x]);   ReadLn(T) end;
    ReadLn(T); for y := 0 to cy − 1 do
      begin for x := 0 to cx − 1 do Read(T, SN[y, x]);   ReadLn(T) end;
    { Primerjajmo sliki in določimo premik. }
    NajA := −1; NajB := −1;
    for dy := − dyMax to dyMax do for dx := − dxMax to dxMax do begin
      OcenaA := 0; OcenaB := 0;
      y := dy; if y < 0 then y := 0;
      while (y < cy) and (y − dy < cy) do begin
        x := dx; if x < 0 then x := 0;
        while (x < cx) and (x − dx < cx) do begin
          OcenaA := OcenaA + Abs(SP[y, x] − SN[y − dy, x − dx]);
          OcenaB := OcenaB + 1; x := x + 1;
```

```
          end; {while x}
          y := y + 1;
        end; {while y}
        { Ocena trenutnega premika je OcenaA / OcenaB, ocena najboljšega doslej znanega
          pa je NajA / NajB. Če je trenutni boljši, si ga zapomnimo. }
        if (NajB < 0) or (OcenaA * NajB < OcenaB * NajA) then
          begin NajA := OcenaA; NajB := OcenaB; dxNaj := dx; dyNaj := dy end;
      end; {for dx, dy}
      WriteLn(U, dxNaj, ' ', dyNaj);
    end; {while}
    Close(T); Close(U);
end. {OpticnaMiska}
```

Postopki, ki jih miške res uporabljajo za ugotavljanje premikov, so omenjeni npr. v Wikipedijinem članku Optical Flow.

2. Spletne knjigarne

Če bi knjigarne zaračunavale enako poštnino za prvo knjigo kot za vse ostale, bi bila ta naloga zelo preprosta. Dovolj bi bilo, če bi za vsako knjigo j in vsako knjigarno i pogledali, koliko bi nas stala ta knjiga z dostavo vred, če bi jo naročili pri tej knjigarni: to je skupaj $b_i + c_{ji}$ enot. Vsako knjigo bi potem naročili pri tisti knjigarni, kjer je ta vsota najmanjša.

Pri naši nalogi pa so stvari malo bolj zapletene, ker je poštnina za prvo knjigo pri knjigarni i lahko dražja, namreč a_i namesto b_i. Na ta strošek lahko pogledamo tudi takole: za to, da bomo od knjigarne i sploh lahko kaj naročili, ji moramo plačati konstantno vsoto $a_i - b_i$ denarja; potem pa bomo za vsako naročeno knjigo (tudi prvo) plačali tej knjigarni še b_i denarnih enot za poštnino.

Vnaprej je težko reči, od katerih knjigarn se splača naročati knjige. Mogoče ima neka knjigarna sicer ugodno nizke zneske $b_i + c_{ji}$, vendar ima tako visok $a_i - b_i$, da je bolje, če od nje sploh ničesar ne naročamo. Ker je knjigarn malo, si lahko privoščimo pregledati kar vse možne množice knjigarn. Pri k knjigarnah imamo $2^k - 1$ nepraznih množic knjigarn. Za vsako tako množico M si mislimo, da smo se že sprijaznili s tem, da bomo od vsake od teh knjigarn mogoče kaj naročili, kar nas bo skupaj stalo $\sum_{i \in M}(a_i - b_i)$ enot denarja. Potem moramo ugotoviti le še to, pri kateri od teh knjigarn je posamezna knjiga najcenejša (z dostavo vred). Tako bomo za knjige plačali še nadaljnjih $\sum_{j=1}^{n} \min_{i \in M}(b_i + c_{ji})$ enot denarja.

Ker bomo sčasoma pregledali vse možne množice knjigarn, bomo prej ali slej prišli tudi do tiste, pri kateri je res dosežena optimalna rešitev. Spodnji program predstavi množico M kar s celim številom, v katerem bit $i-1$ pove, ali je knjigarna i v tej množici ali ne. V tabeli Cj si pripravi vrednosti $\min_{i \in M}(b_i + c_{ji})$, ki povedo najmanjšo ceno (z dostavo vred) za posamezno knjigo j, gledano po vseh knjigarnah i iz M.

```
program SpletneKnjigarne;
const MaxK = 10; MaxN = 10000; MaxCena = 10000;
var Ai, Bi: array [1..MaxK] of integer;
    Cji: array [1..MaxN, 1..MaxK] of integer;
    Cj: array [1..MaxN] of integer;
    i, j, k, n, M, Cena, NajCena: integer; T: text;
begin
  { Preberimo vhodno datoteko. }
  Assign(T, 'knjigarne.in'); Reset(T); ReadLn(T, n, k);
  for i := 1 to k do Read(T, Ai[i]);   ReadLn(T);
  for i := 1 to k do Read(T, Bi[i]);   ReadLn(T);
  for j := 1 to n do
    begin for i := 1 to k do Read(T, Cji[j, i]);   ReadLn(T) end;
  { Preglejmo vse neprazne množice knjigarn. }
  NajCena := (MaxCena + 1) * (n + n + k);
  for M := 1 to (1 shl k) - 1 do begin
    Cena := 0;
    for j := 1 to n do Cj[j] := 2 * MaxCena + 1;
```

```
    for i := 1 to k do if (M and (1 shl (i − 1))) <> 0 then begin
      Cena := Cena + Ai[i] − Bi[i];
      for j := 1 to n do if Cj[j] > Bi[i] + Cji[j, i] then
        Cj[j] := Bi[i] + Cji[j, i];
    end; {for i}
    for j := 1 to n do Cena := Cena + Cj[j];
    if Cena < NajCena then NajCena := Cena;
  end; {for M}

  { Izpišimo rezultat. }
  Assign(T, 'knjigarne.out'); Rewrite(T); WriteLn(T, NajCena); Close(T);
end. {SpletneKnjigarne}
```

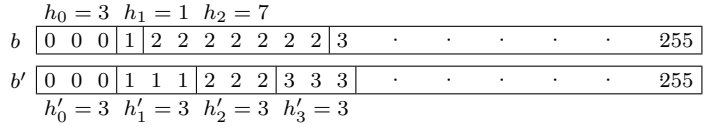
3. Izravnavanje histogramov

Naj bo h_c (za $0 \le c \le 255$) število slikovnih elementov (pikslov) barve c na vhodni sliki I, h'_c pa število slikovnih elementov barve c na izhodni sliki I'. Histogram $h = (h_0, \ldots, h_{255})$ lahko določimo s pregledom vhodne slike I, histogram $h' = (h'_0, \ldots, h'_{255})$ pa določimo na podlagi zahteve, da se smeta najvišji in najnižji stolpec razlikovati največ za 1. Če ima naša slika vsega skupaj n slikovnih elementov, lahko za začetek postavimo vsak stolpec histograma na višino n div 256. Tako smo že porabili $256 \cdot (n \text{ div } 256)$ od vseh n slikovnih elementov; ostalo jih je še n mod 256, kar je manj kot 256. Torej lahko n mod 256 stolpcev dvignemo za 1 in tako porabimo vse slikovne elemente, naš izhodni histogram h' pa še vedno ustreza zahtevi, da najvišji stolpec ni več kot za 1 višji od najnižjega.

Recimo, da smo že našli primerno sliko I', ki ima histogram h' in ustreza tudi drugi od zahtev naše naloge, torej: če je nek slikovni element na sliki I svetlejši od nekega drugega, mora biti na sliki I' tudi svetlejši ali pa iste barve, nikakor pa ne temnejši. Ta zahteva v bistvu pravi, da morajo najtemnejši slikovni elementi vhodne slike postati tudi najtemnejši elementi izhodne slike, najsvetlejši vhodne slike pa najsvetlejši izhodne slike in podobno.

Mislimo si, da bi za vsak slikovni element naše slike zapisali urejen par $(b_\bullet, b'_\bullet)$, pri čemer je $b_\bullet$ barva tega elementa na vhodni sliki, $b'_\bullet$ pa na izhodni sliki. Vse tako urejene pare uredimo naraščajoče po $b_\bullet$, znotraj vsake vrednosti $b_\bullet$ pa še naraščajoče po $b'_\bullet$. V tako urejenem zaporedju pare oštevilčimo od 0 do $n-1$; i-ti par je torej (b_i, b'_i). Zaradi načina urejanja vemo, da je zaporedje $b = (b_0, \ldots, b_{n-1})$ nepadajoče. Toda ker I' ustreza zahtevam naloge, vemo, da je tudi zaporedje $b' = (b'_0, \ldots, b'_{n-1})$ nepadajoče: če to ne bi bilo res, bi bil torej nekje nek $b'_i > b'_{i+1}$, kar pa je zaradi našega načina urejanja zaporedja parov $(b_\bullet, b'_\bullet)$ možno le, če sta b_i in b_{i+1} različna; to pa je možno le v obliki $b_i < b_{i+1}$, saj smo zaporedje parov uredili po $b_\bullet$-jih. Tedaj bi torej imeli dva slikovna elementa, od katerih bi bil prvi na vhodni sliki temnejši od drugega ($b_i < b_{i+1}$), na drugi pa svetlejši ($b'_i > b'_{i+1}$), kar bi bilo v protislovju z zahtevo iz naloge.

Ker poznamo histograma obeh slik, seveda tudi vemo, da je prvih h_0 členov zaporedja b enakih 0, naslednjih h_1 členov tega zaporedja je enakih 1 in tako naprej; podobno je prvih h'_0 členov zaporedja b' enakih 0, naslednjih h'_1 členov je enakih 1 in tako naprej. Obe zaporedji torej pravzaprav že v celoti poznamo in ju lahko v mislih napišemo eno nad drugim, na primer takole:



Obenem pa zaradi načina, kako smo do teh zaporedij prišli, še vedno velja, da vsak par istoležnih elementov (b_i, b'_i) opisuje spremembo barve v enem od slikovnih elementov naše slike (torej pove, da se je nek slikovni element barve b_i spremenil v barvo b'_i). Ker je na vhodni sliki h_0 slikovnih elementov barve 0, imamo v našem zaporedju b vrednosti $b_i = 0$ pri $i = 0, 2, \ldots, h_0 - 1$, pripadajoče vrednosti b'_i pa nam povedo, kako moramo pobarvati te slikovne elemente na izhodni sliki. Naslednjih h_1 členov zaporedja b (torej

za $i = h_0, \ldots, h_0 + h_1 - 1$) ima $b_i = 1$ in pripadajoče vrednosti b'_i nam povedo, kako pobarvati na izhodni sliki tiste slikovne elemente, ki so na vhodni sliki barve 1. Tako lahko nadaljujemo, dokler ne pobarvamo cele izhodne slike:

```
1    pripravi zaporedji b in b', kot je bilo opisano zgoraj;
2    i := 0;
3    za vsako barvo c od 0 do 255 ponovi:
4        za vsak slikovni element (x, y), ki je na stari sliki I barve c:
5            vemo, da je b_i = c; na izhodni sliki pobarvaj ta element z barvo b'_i;
6            i := i + 1;
```

Pri tem postopku že vidimo, da zaporedja b ni treba hraniti ekspliticno, saj ga pravzaprav sploh nikjer ne uporabljamo. Še lepše pa je, da tudi zaporedja b' ni treba hraniti eksplicitno, saj dostopamo do njega po naraščajočih indeksih i. Ker vemo, da ima prvih h'_0 členov tega zaporedja vrednost 0, naslednjih h'_1 členov vrednost 1 in tako naprej, je dovolj že, če si zapomnimo, na katerem od teh intervalov leži trenutni i in kako daleč mu še manjka do konca tega intervala. Tako pridemo do naslednjega postopka:

```
1    i := 0; c' := 0; ν := h'_0;
2    za vsako barvo c od 0 do 255 ponovi:
3        za vsak slikovni element (x, y), ki je na stari sliki I barve c:
4            vemo, da je indeks i v zaporedju b na območju barve c (torej b_i = c);
5            vemo tudi, da je indeks i v zaporedju b' na območju barve c' in da se
                 naslednje območje začne pri indeksu i + ν;
6            while ν = 0 do begin c' := c' + 1; ν := h'_{c'} end;
7            pobarvaj (x, y) na izhodni sliki I' z barvo c';
8            i := i + 1; ν := ν − 1;
```

Notranja zanka **while** torej poskrbi za to, da c' povečamo, ko pridemo v zaporedju b' do indeksa, kjer se vrednost členov tega zaporedja poveča. Tako je c' po koncu te zanke zagotovo enak b'_i.

Gornji postopek bi načeloma za vsako barvo od 0 do 255 izvedel po en prehod čez celo sliko, da bi našel slikovne elemente te barve in jih ustrezno prebarval. Še elegantnejši in hitrejši postopek pa dobimo, če vse barve prebarvamo istočasno. Gornji postopek, ki pregleduje barve eno za drugo od temnejših k svetlejšim, bi imel opravka z barvo c takrat, ko bi imel števec i vrednosti od $h_0+h_1+\ldots+h_{c-1}$ do $h_0+h_1+\ldots+h_{c-1}+(h_c-1)$. Če bi torej znali za vsakega od začetnih indeksov $h_0 + \ldots + h_{c-1}$, pri katerih se začne naše delo s slikovnimi elementi barve c, ugotoviti, kakšna je takrat v gornjem postopku vrednost spremenljivk c' in ν (recimo tema vrednostma $c'[c]$ in $\nu[c]$), bi lahko potem v enem samem prehodu čez sliko vzporedno izvajali vseh 256 iteracij glavne zanke gornjega postopka — ko naletimo na slikovni element barve c, izvedemo eno iteracijo notranje zanke (vrstice 4–8) znotraj tiste iteracije zunanje zanke, ki se nanaša na to barvo c.

Do vrednosti $c'[c]$ in $\nu[c]$ pa tudi ni težko priti; še enkrat moramo v mislih izvesti gornji postopek, le da iz njega pobrišemo vse ukvarjanje s konkretnimi slikovnimi elementi in iskanje le-teh po sliki: zdaj je pomembno le, kolikokrat bi se notranja zanka (vrstice 4–8) izvedla pri posameznem c in kako bi se zaradi tega spremenili vrednosti c' in ν. Tako pridemo do naslednjega postopka za inicializacijo tabel $c'[c]$ in $\nu[c]$:

```
1    c' := 0; ν := h'_0;
2    za vsako barvo c od 0 do 255 ponovi:
3        c'[c] := c'; ν[c] := ν;
4        ν := ν − h_c;
5        while ν ≤ 0 and c' < 255 do begin c' := c' + 1; ν := ν + h'_{c'} end;
```

Barvanje stare slike v novo lahko zdaj poteka takole:

```
1    za vsak slikovni element (x, y):
2        naj bo c barva tega elementa na stari sliki;
3        while ν[c] = 0 do begin c'[c] := c'[c] + 1; ν[c] := h'_{c'[c]} end;
4        pobarvaj (x, y) na izhodni sliki I' z barvo c'[c];
5        ν[c] := ν[c] − 1;
```

Tako smo dobili učinkovit in eleganten postopek, ki potrebuje le dva prehoda po sliki (enega, da določi histogram h, in drugega, da sliko prebarva oz. ustvari izhodno sliko) in ima tako na sliki z n slikovnimi elementi in B barvami (v našem primeru je $B = 256$) časovno zahtevnost $O(n + B)$.

Zapišimo našo rešitev še v obliki programa. Tabeli Hist in CiljHist hranita histograma h in h', tabeli Nova in NovaKoliko pa hranita vrednosti $c'[c]$ in $\nu[c]$.

```
program IzravnavanjeHistograma;
const MaxSirina = 200; MaxVisina = 200;
var T: text;
    Slika: array [0..MaxVisina − 1, 0..MaxSirina − 1] of integer;
    Hist, CiljHist, Nova, NovaKoliko: array [0..255] of integer;
    x, y, c, cNova, Koliko, Sirina, Visina: integer;
begin
  { Preberimo vhodno sliko. }
  Assign(T, 'histogram.in'); Reset(T); ReadLn(T, Visina, Sirina);
  for y := 0 to Visina − 1 do for x := 0 to Sirina − 1 do Read(T, Slika[y, x]);
  Close(T);
  { Izračunajmo histogram prvotne slike. }
  for c := 0 to 255 do Hist[c] := 0;
  for y := 0 to Visina − 1 do for x := 0 to Sirina − 1 do
    Hist[Slika[y, x]] := Hist[Slika[y, x]] + 1;
  { V tabeli CiljHist pripravimo enega od možnih histogramov nove slike. Druge možne
    histograme, ki bi tudi ustrezali zahtevam naloge, bi lahko dobili tako, da bi
    druga zanka povečala za 1 kakšne druge elemente, ne pa ravno prvih nekaj. }
  for c := 0 to 255 do CiljHist[c] := (Sirina * Visina) div 256;
  for c := 0 to ((Sirina * Visina) mod 256) − 1 do CiljHist[c] := CiljHist[c] + 1;
  { Vsaka barva prvotne slike se preslika v eno ali več zaporednih barv nove slike.
    Izračunajmo za začetek najtemnejšo barvo Nova[c], v katero se bo preslikala posamezna
    barva c prvotne slike, pa še to, koliko pikslov barve c se bo preslikalo v barvo Nova[c]. }
  cNova := 0; Koliko := CiljHist[cNova];
  for c := 0 to 255 do begin
    Nova[c] := cNova; NovaKoliko[c] := Koliko;
    Koliko := Koliko − Hist[c];
    while (Koliko <= 0) and (cNova < 255) do
      begin cNova := cNova + 1; Koliko := Koliko + CiljHist[cNova] end;
  end; {for c}
  { Pojdimo po sliki in popravljajmo barve pikslov. Barva c na stari sliki postane barva Nova[c]
    na novi sliki. Če pa ima ta barva že dovolj pikslov, začnimo uporabljati naslednjo. }
  for y := 0 to Visina − 1 do for x := 0 to Sirina − 1 do begin
    c := Slika[y, x];
    while NovaKoliko[c] = 0 do
      begin Nova[c] := Nova[c] + 1; NovaKoliko[c] := CiljHist[Nova[c]] end;
    Slika[y, x] := Nova[c]; NovaKoliko[c] := NovaKoliko[c] − 1;
  end; {for x, y}
  { Izpišimo rezultat. }
  Assign(T, 'histogrami.out'); Rewrite(T);
  for y := 0 to Visina − 1 do for x := 0 to Sirina − 1 do begin
    Write(T, Slika[y, x]); if x = Sirina − 1 then WriteLn(T) else Write(T, ' ');
  end; {for x, y}
  Close(T);
end. {IzravnavanjeHistograma}
```

4. Mafija

Za vsakega člana družine vodimo podatek o razliki med njegovimi prejemki in tem, kar je dal nadrejenemu. Vsaka vrstica vhodne datoteke nam pove, da je član i dal svojemu nadrejenemu (recimo mu j) nek znesek denarja; ta znesek torej prištejemo j-jevemu elementu tabele in odštejemo od i-jevega elementa. Na koncu imamo tako v tej tabeli za vsakega člana ravno količino denarja, ki ga je utajil. Poiskati moramo največji

element tabele in ga izpisati; pri tem pa moramo paziti, da ne upoštevamo elementa, ki se nanaša na šefa družine: on nima nadrejenega, zato le prejema in v naši tabeli utaj se mu lahko nabere zelo velik znesek (ki pa seveda ne pomeni utaje).

```
program Mafija;
const MaxN = 100000;
var Utajil: array [1..MaxN] of integer;
    i, j, Znesek, n, MaxUtaja, Sef: integer; T: text;
begin
  Assign(T, 'mafija.in'); Reset(T); ReadLn(T, n);
  for i := 1 to n do Utajil[i] := 0;
  for i := 1 to n do begin
    ReadLn(T, j, Znesek); Assert(0 <= Znesek);
    if j = 0 then Sef := i
    else begin Utajil[i] := Utajil[i] - Znesek; Utajil[j] := Utajil[j] + Znesek end;
  end; {for i}
  Close(T);
  { Poiščimo največjo utajo. }
  MaxUtaja := 0;
  for i := 1 to n do if (i <> Sef) and (Utajil[i] > MaxUtaja) then MaxUtaja := Utajil[i];
  { Izpišimo rezultat. }
  Assign(T, 'mafija.out'); Rewrite(T); WriteLn(T, MaxUtaja); Close(T);
end. {Mafija}
```

5. Tovornjaki

Pri tej nalogi ni nujno priti do najboljše možne rešitve; dovolj za nekaj točk je že, da se ji vsaj čim bolj približamo. Izkaže se, da lahko blizu dobrim rešitvam pridemo že z naključnim razporejanjem tovornjakov na pasove. Vsak tovornjak naključno razporedimo na enega od treh pasov in na koncu pogledamo, kako dolg trajekt potrebujemo. To velikokrat ponovimo in na koncu izpišemo tisti razpored, ki je dal najkrajši trajekt.

```
program Tovornjaki;
const MaxStTov = 100; StPasov = 3;
var
  Dolzina, Pas, NajPas: array [1..MaxStTov] of integer;
  DolPasu: array [1..StPasov] of integer;
  Trajekt, NajTrajekt: integer; i, t, StTov: integer; T: text;
begin
  { Preberimo vhodno datoteko. }
  Assign(T, 'tovornjaki.in'); Reset(T); ReadLn(T, StTov);
  NajTrajekt := 0;
  for i := 1 to StTov do begin
    ReadLn(T, Dolzina[i]);
    NajTrajekt := NajTrajekt + Dolzina[i]; NajPas[i] := 1;
  end; {for i}
  Close(T);
  { Preizkusimo veliko naključnih razporedov. }
  for t := 1 to 10000 do begin
    for i := 1 to StPasov do DolPasu[i] := 0;
    for i := 1 to StTov do begin
      Pas[i] := Random(StPasov) + 1;
      DolPasu[Pas[i]] := DolPasu[Pas[i]] + Dolzina[i];
    end; {for i}
    Trajekt := 0;
    for i := 1 to 3 do if DolPasu[i] > Trajekt then Trajekt := DolPasu[i];
    if Trajekt < NajTrajekt then
      begin NajTrajekt := Trajekt; for i := 1 to StTov do NajPas[i] := Pas[i] end;
  end; {for t}
  { Izpišimo najboljši razpored. }
  Assign(T, 'tovornjaki.out'); Rewrite(T); WriteLn(T, NajTrajekt);
```

```
  for i := 1 to StTov do WriteLn(T, NajPas[i]);
  Close(T);
end. {Tovornjaki}
```

Pri naših testnih primerih je tovornjakov malo in njihove dolžine so majhna naravna števila, tako da gornji postopek praktično vedno najde trajekt, ki je le malo daljši od najkrajšega možnega; pogosto pa najde sploh najkrajši možni trajekt.

Še bolje se obnese različica gornje rešitve, ki razporeja tovornjake „požrešno“ namesto naključno: vsak tovornjak pošljemo na tisti pas, na katerem je skupna dolžina dosedanjih tovornjakov najmanjša. Dolžina dobljenega trajekta je pri tem postopku v splošnem odvisna od tega, v kakšnem vrstnem redu pregledujemo tovornjake, zato je pametno preizkusiti več naključnih vrstnih redov in izpisati najboljšo izmed vseh dobljenih rešitev:

```
program Tovornjaki;
const MaxStTov = 100; StPasov = 3;
var Dolzina, Pas, NajPas, VrstniRed: array [1..MaxStTov] of integer;
    DolPasu: array [1..StPasov] of integer;
    Trajekt, NajTrajekt: integer; i, j, k, p, StTov: integer; T: text;
begin
  { Preberimo vhodno datoteko. }
  Assign(T, 'tovornjaki.in'); Reset(T); ReadLn(T, StTov);
  NajTrajekt := 0;
  for i := 1 to StTov do begin
    ReadLn(T, Dolzina[i]);
    NajTrajekt := NajTrajekt + Dolzina[i]; NajPas[i] := 1;
  end; {for i}
  Close(T);
  { Preizkusimo veliko naključnih vrstnih redov. }
  for k := 1 to 1000 do begin
    { Sestavimo nek naključen vrstni red tovornjakov. }
    for i := 1 to StTov do begin
      j := Random(i) + 1;
      if j < i then VrstniRed[i] := VrstniRed[j];
      VrstniRed[j] := i;
    end; {for i}
    { Preglejmo tovornjake v tem vrstnem redu, vsakega dajmo na najkrajši pas. }
    for i := 1 to StPasov do DolPasu[i] := 0;
    for i := 1 to StTov do begin
      j := VrstniRed[i]; Pas[j] := 1;
      for p := 2 to StPasov do if DolPasu[p] < DolPasu[Pas[j]] then Pas[j] := p;
      DolPasu[Pas[j]] := DolPasu[Pas[j]] + Dolzina[j];
    end; {for i}
    { Če je to najboljši razpored doslej, si ga zapomnimo. }
    Trajekt := 0;
    for i := 1 to 3 do if DolPasu[i] > Trajekt then Trajekt := DolPasu[i];
    if Trajekt < NajTrajekt then
      begin NajTrajekt := Trajekt; for i := 1 to StTov do NajPas[i] := Pas[i] end
  end; {for k}
  { Izpišimo najboljši razpored. }
  Assign(T, 'tovornjaki.out'); Rewrite(T); WriteLn(T, NajTrajekt);
  for i := 1 to StTov do WriteLn(T, NajPas[i]);
  Close(T);
end. {Tovornjaki}
```

Ta postopek je pri naših poskusih in na naših testnih primerih vedno odkril najkrajši možni trajekt. V splošnem pa nam niti naključni niti požrešni postopek ne dajeta zagotovila, da bodo njuni trajekti res najkrajši možni.

Oglejmo si še postopek, ki je časovno in prostorsko znatno zahtevnejši od gornjih dveh, vendar pa vedno zagotovo najde najkrajši možni trajekt.

Stanje trajekta lahko predstavimo z urejeno trojico števil (p_1, p_2, p_3), ki povedo skupno dolžino vseh tovornjakov na vsakem od treh pasov. V naši vhodni datoteki

je neko zaporedje n tovornjakov z dolžinami $d_1, \ldots, d_n$. Če je mogoče prvih i tovornjakov razporediti v tri pasove tako, da je vsota dolžin tovornjakov na prvem pasu p_1, na drugem p_2 in na tretjem p_3, bomo rekli, da je stanje (p_1, p_2, p_3) *dosegljivo* s prvimi i tovornjaki. (Takoj lahko opazimo, da nobeno stanje ne more biti dosegljivo s prvimi i tovornjaki za več različnih vrednosti i — če je dosegljivo s prvimi i, to pomeni, da je $p_1 + p_2 + p_3$ enako vsoti dolžin prvih i tovornjakov; če bi bilo hkrati dosegljivo tudi s prvimi i' tovornjaki za nek $i' > i$, bi to pomenilo, da je $p_1 + p_2 + p_3$ enako tudi vsoti dolžin prvih i' tovornjakov, torej je vsota dolžin tovornjakov od $i + 1$ do i' enaka 0, kar pa je nemogoče.)

Stanje (p_1, p_2, p_3) je možno le na trajektu, ki je dolg vsaj $\max\{p_1, p_2, p_3\}$ enot. Če bi pregledali vsa stanja, dosegljiva z n tovornjaki, bi torej lahko za vsako ugotovili, kako dolg trajekt zahteva, in izpisali tisto z najkrajšim trajektom.

S preprostim rekurzivnim podprogramom (Razvij v spodnjem programu) si lahko v neki tabeli (StTov) pripravimo podatke o tem, s koliko tovornjaki je dosegljivo posamezno stanje (če je sploh dosegljivo). Rekurzija temelji na opažanju, da če je stanje (p_1, p_2, p_3) dosegljivo s prvimi $i - 1$ tovornjaki, potem so stanja $(p_1 + d_i, p_2, p_3)$, $(p_1, p_2 + d_i, p_3)$ in $(p_1, p_2, p_3 + d_i)$ dosegljiva s prvimi i tovornjaki. Ob vsakem stanju si tudi zapomnimo (tabela KjeZadnji), na kateri pas smo pri tem stanju uvrstili zadnji (i-ti) tovornjak. Začnemo pa lahko ta postopek pri $i = 0$ tovornjakih, ko vemo, da je dosegljivo le stanje $(0, 0, 0)$.

S tabelo KjeZadnji si pomagamo, da na koncu rekonstruiramo celoten razpored tovornjakov po pasovih. Če je (p_1, p_2, p_3) stanje, ki nam da najkrajši trajekt za vseh n tovornjakov, lahko s pogledom v KjeZadnji ugotovimo, na katerem pasu stoji zadnji (n-ti) tovornjak v razporedu, ki pripelje do tega stanja; dolzino tega tovornjaka potem odštejemo od enega od števil p_1, p_2, p_3 (odvisno od pasu, na katerem je bil tovornjak) in tako dobimo stanje pred prihodom zadnjega tovornjaka; zanj spet pogledamo v KjeZadnji in tako ugotovimo, kje mora stati $(n - 1)$-vi tovornjak in tako naprej.

```
program Tovornjaki;
const MaxDolzTraj = 100; MaxDolzTov = MaxDolzTraj; MaxStTov = 100;
type TabelaP = ↑TabelaT;
     TabelaT = packed array [0..MaxDolzTraj, 0..MaxDolzTraj, 0..MaxDolzTraj] of integer;
var n: integer; StTov, KjeZadnji: TabelaP; Dolzine: array [1..MaxStTov] of integer;

  { Klic tega podprograma pomeni, da je stanje (d1, d2, d3) dosegljivo s prvimi k tovornjaki,
    pri čemer je zadnji od njih na pasu p. }
  procedure Razvij(d1, d2, d3, k, p: integer);
  var d: integer;
  begin
    if (d1 > MaxDolzTraj) or (d2 > MaxDolzTraj) or (d3 > MaxDolzTraj) then exit;
    if StTov↑[d1, d2, d3] >= k then exit; { Do tega stanja smo že prišli nekoč prej. }
    StTov↑[d1, d2, d3] := k; KjeZadnji↑[d1, d2, d3] := p;
    if k < n then begin
      d := Dolzine[k + 1]; Razvij(d1 + d, d2, d3, k + 1, 1);
      Razvij(d1, d2 + d, d3, k + 1, 2); Razvij(d1, d2, d3 + d, k + 1, 3);
    end; {if}
  end; {Razvij}

  procedure NajdiResitev(var d1, d2, d3: integer);
  var i1, i2, i3: integer;
  begin
    for i1 := 0 to MaxDolzTraj do for i2 := 0 to i1 do for i3 := 0 to i2 do
      if StTov↑[i1, i2, i3] = n then begin d1 := i1; d2 := i2; d3 := i3; exitend;
  end; {NajdiResitev}

var d1, d2, d3, i, r, Dolzina: integer; Pasovi: array [1..MaxStTov] of integer; T: text;
begin
  Assign(T, 'tovornjaki.in'); Reset(T); ReadLn(T, n);
  New(StTov); New(KjeZadnji);
  for d1 := 0 to MaxDolzTraj do for d2 := 0 to MaxDolzTraj do
    for d3 := 0 to MaxDolzTraj do StTov↑[d1, d2, d3] := -1;
  for i := 1 to n do ReadLn(T, Dolzine[i]);
```

```
  Close(T);
  Razvij(0, 0, 0, 0, 0);
  NajdiResitev(d1, d2, d3);
  r := d1; if d2 > r then r := d2; if d3 > r then r := d3;
  for i := n downto 1 do begin
    Pasovi[i] := KjeZadnji↑[d1, d2, d3];
    if Pasovi[i] = 1 then d1 := d1 − Dolzine[i]
    else if Pasovi[i] = 2 then d2 := d2 − Dolzine[i]
    else d3 := d3 − Dolzine[i];
  end; {for i}
  Dispose(StTov); Dispose(KjeZadnji);
  Assign(T, 'tovornjaki.out'); Rewrite(T); WriteLn(T, r);
  for i := 1 to n do WriteLn(T, Pasovi[i]);
  Close(T);
end. {Tovornjaki}
```

Viri nalog: predpone — Gorazd Božič; sudoku — Primož Gabrijelčič; cestne lučke — Boris Gašperin; mafija — Uroš Jovanovič; sneg — Aleš Košir; odstavki, 1337ovščina, optična miška — Mark Martinec; končniška drevesa — Mojca Miklavec; tovornjaki — Miha Vuk; naraščajoče besede, podnapisi, razvajeni zvezdniki, spletne knjigarne, izravnavanje histogramov — Janez Brank. Vprašanja, pripombe, komentarji, popravki ipd. v zvezi z nalogami so dobrodošli: ⟨*janez@brank.org*⟩.