18. tekmovanje ACM v znanju računalništva za srednješolce

25. marca 2023

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 vejitev, zank 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;
```

(Opomba: samo zato, ker je tu primer psevdokode, to še ne pomeni, da moraš tudi ti pisati svoje odgovore v psevdokodi.)

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 (s tem je mišljeno predvsem, naj ima rešitev učinkovit algoritem; drobne tehnične optimizacije niso tako pomembne). Za manjše sintaktične napake se ne odbije veliko točk. Priporočljivo in zaželeno je, da so tvoje rešitve napisane pregledno in čitljivo. Če je na listih, ki jih oddajaš, več različic rešitve za kakšno nalogo, jasno označi, katera je tista, ki naj jo ocenjevalci upoštevajo.

Če naloga zahteva branje ali obdelavo 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[201]; 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 dvesto 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 in drugi 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;
}
```

Še isti trije primeri v pythonu:

```
# Branje dveh števil in izpis vsote:
import sys

a, b = sys.stdin.readline().split()
a = int(a); b = int(b)
print(f"{a} + {b} = {a + b}")
```

```
# Branje standardnega vhoda po vrsticah:
import sys

i = d = 0
for s in sys.stdin:
  s = s.rstrip('\n') # odrežemo znak za konec vrstice
  i += 1; d += len(s)
  print(f"{i}. vrstica: \"{s}\"")
print(f"{i} vrstic, {d} znakov.")
```

```
# Branje standardnega vhoda znak po znak:
import sys

i = 0
while True:
  c = sys.stdin.read(1)
  if c == "": break # EOF
  sys.stdout.write(c)
  if c != '\n': i += 1
print(f"Skupaj {i} znakov.")
```

Še isti trije primeri v javi:

```
// Branje dveh števil in izpis vsote:
import java.io.*;
import java.util.Scanner;

public class Primer1
{
   public static void main(String[] args) throws IOException
   {
      Scanner fi = new Scanner(System.in);
      int i = fi.nextInt(); int j = fi.nextInt();
      System.out.println(i + " + " + j + " = " + (i + j));
   }
}
```

```
// Branje standardnega vhoda po vrsticah:
import java.io.*;

public class Primer2
{
   public static void main(String[] args) throws IOException
   {
      BufferedReader fi = new BufferedReader(new InputStreamReader(System.in));
      int i = 0, d = 0; String s;
      while ((s = fi.readLine()) != null) {
         i++; d += s.length();
         System.out.println(i + ". vrstica: \"" + s + "\""); }
      System.out.println(i + " vrstic, " + d + " znakov.");
   }
}
```

```
// Branje standardnega vhoda znak po znak:
import java.io.*;

public class Primer3
{
   public static void main(String[] args) throws IOException
   {
      InputStreamReader fi = new InputStreamReader(System.in);
      int i = 0, c;
      while ((c = fi.read()) >= 0) {
         System.out.print((char) c); if (c != '\n' && c != '\r') i++; }
      System.out.println("Skupaj " + i + " znakov.");
   }
}
```

18. tekmovanje ACM v znanju računalništva za srednješolce

25. marca 2023

NALOGE ZA PRVO SKUPINO

Odgovore lahko pišeš/rišeš na papir ali pa jih natipkaš z računalnikom ali pa oddaš del odgovorov na papirju in del prek računalnika. Vse te možnosti so enakovredne. Če oddajaš kaj na papirju, napiši na vsak oddani list svoje ime. Pri delu si lahko pomagaš s prevajalniki in razvojnimi orodji, ki so na voljo na tvojem računalniku, vendar bomo tvoje odgovore v vsakem primeru pregledali in ocenili ročno (ne glede na to, ali si jih oddal prek računalnika ali na papirju), zato manjše napake v sintaksi ali pri klicih funkcij standardne knjižnice niso tako pomembne, kot bi bile na tekmovanjih z avtomatskim ocenjevanjem.

Tekmovanje bo potekalo na strežniku `https://rtk.fri.uni-lj.si/`, kjer dobiš naloge in oddajaš svoje odgovore. Uporabniška imena in gesla vas bodo čakala na mizi v učilnici. Pri oddaji preko računalnika odpreš dotično nalogo v spletni učilnici in rešitev natipkaš oz. prilepiš v polje za programsko kodo. Med tipkanjem se rešitev na približno dve minuti samodejno shrani. Poleg tega lahko sam med pisanjem rešitve izrecno zahtevaš shranjevanje rešitve s pritiskom na gumb „Shrani spremembe“. Ker je vgrajeni urejevalnik dokaj preprost in ne omogoča označevanja kode z barvami, predlagamo, da rešitev pripraviš v kakšnem drugem urejevalniku na računalniku (Visual Studio Code, Geany, Lazarus) in jo nato prekopiraš v okno spletnega urejevalnika. Naj te ne moti, da se bodo barvne oznake kode pri kopiranju izgubile.

Ko si bodisi zadovoljen z rešitvijo ter si zaključil nalogo ali ko želiš začasno prekiniti pisanje rešitve naloge ter se lotiti druge naloge, uporabi gumb „Shrani spremembe“ in nato klikni na „Nazaj na seznam nalog“, da se vrneš v glavni meni. (Oddano rešitev lahko kasneje še spreminjaš.) Za vsak slučaj priporočamo, da pred oddajo shraniš svoj odgovor tudi v datoteko na svojem lokalnem računalniku.

Med reševanjem lahko vprašanja za tekmovalno komisijo postavljaš prek zasebnih sporočil na tekmovalnem strežniku (ikona oblačka zgoraj desno) ali pa vprašaš člane komisije, ki bodo prisotni v učilnicah. Prek zasebnih sporočil bomo pošiljali tudi morebitna pojasnila in popravke, če bi se izkazalo, da so v besedilu nalog kakšne nejasnosti ali napake. Zato med reševanjem redno preverjaj, če so se pojavila kakšna nova zasebna sporočila. Če imaš težave z računalnikom ali s povezavo s spletnim strežnikom za oddajo nalog in komunikacijo s tekmovalno komisijo, se nemudoma obrni na nadzornika v učilnici, ki bo zagotovil drug računalnik. **Če zaradi morebitnih težav pri oddajanju rešitev na strežnik želiš, da ocenimo odgovore v datotekah na lokalnem disku tvojega računalnika, o tem obvezno obvesti nadzorno osebo v svoji učilnici, še preden odideš iz nje.**

Svoje odgovore dobro utemelji. Če pišeš izvorno kodo programa ali podprograma, **OBVEZNO** tudi v nekaj stavkih z besedami opiši idejo, na kateri temelji tvoja rešitev. Če ni v nalogi drugače napisano, lahko tvoje rešitve predpostavljajo, da so vhodni podatki brez napak (da ustrezajo formatu in omejitvam, kot jih podaja naloga). Zaželeno je, da so tvoje rešitve poleg tega, da so pravilne, tudi učinkovite; bolj učinkovite rešitve dobijo več točk (s tem je mišljeno predvsem, naj ima rešitev učinkovit algoritem; drobne tehnične optimizacije niso tako pomembne). **Nalog je pet** in pri vsaki nalogi lahko dobiš od 0 do 20 točk. Liste z nalogami lahko po tekmovanju obdržiš.

Rešitve in rezultati bodo objavljeni na `https://rtk.ijs.si/`.

1. Neurejene besede

Mojca in Peter sta urednika šolskega časopisa. Kot vsak zaupanja vreden časopis mora tudi šolski časopis imeti rubriko z ugankami. Po pregledu ostalih časopisov sta se Mojca in Peter odločila, da bosta v šolskem časopisu dijakom v reševanje ponudila „zmešane citate“. Pri tej vrsti ugank je podan stavek, kjer so črke v posameznih besedah premešane, tako da nimajo nobenega smisla, naloga reševalca pa je, da ugotovi pravilen vrstni red črk v besedah, ki mu dajo smiseln citat.

Ker se Mojca in Peter ukvarjata z urejanjem šolskega časopisa, ne pa s programiranjem, sta se obrnila nate, ki obiskuješ programerski krožek. Prosita te, da jima **napišeš program**, ki bo iz citatov generiral uganke. Tvoj program kot vhod prebere citat (lahko ga prebere s standardnega vhoda ali iz datoteke `vhod.txt`, karkoli ti je lažje), izpiše pa naj „zmešani citat“, v katerem so črke vsake besede naključno premešane, ostali znaki citata (presledki in ločila) pa ostanejo nespremenjeni. Predpostavi, da je citat dolg največ 100 znakov, da leži v celoti v eni vrstici in da je posamezna beseda sestavljena le iz črk angleške abecede.

Predpostavi, da je za generiranje naključnih števil na voljo funkcija `Random(n)`, ki vrne naključno celo število od 0 do $n - 1$ (pri čemer so vsa števila enako verjetna).

Nekaj primerov:

vhod: `Danes je lep, topel dan.`
možen izhod: `nDsae ej lpe, peotl dan.`

vhod: `Pes, ki laja, ne grize.`
možen izhod: `sPe, ik jaal, ne zireg.`

vhod: `cDdDc cddcdc CCD... cdcdd ddDc? ccddc`
možen izhod: `cDcdD ddcccd DCC... dddcc Dcdd? cdccd`

2. Kibi, mebi

Velikost datotek ali pomnilnika običajno merimo v bajtih (B), pri zapisu večjih vrednosti pa si naredimo število preglednejše oz. lažje razumljivo tako, da uporabimo multiplikativne predpone K (kilo-), M (mega-), G (giga-) itd. Tako predstavlja en kilobajt 1024 bajtov (1 KB = 1024 B), megabajt je 1024 kilobajtov (1 MB = 1024 KB) in tako naprej, vsaka naslednja predpona (kot jih določa in poimenuje npr. industrijski standard JEDEC) je za faktor 1024 večja od prejšnje. Te predpone po vrsti so: K, M, G, T, P (za naš namen se ustavimo pri P, čeprav obstajajo tudi višje).

Napiši podprogram (oz. funkcijo), ki bo dobil kot argument velikost neke datoteke v bajtih kot nenegativno celo število, potem pa izpisal to vrednost, po potrebi okrajšano z uporabo najnižje možne predpone tako, da število števk zapisa ne bo večje kot štiri. Če je število tako veliko, da zanj uporabimo najvišjo predpono, potem za tak primer omejitev na štiri števke ne velja.

Če ti je lažje, lahko namesto podprograma napišeš program, ki naj prebere število iz vhodne datoteke ali s standardnega vhoda.

Izpišemo vedno le celi del (brez morebitnih decimalk), temu naj sledi črka B (= bajt), pred katero naj po potrebi stoji črka predpone.

Če število bajtov ni mnogokratnik vrednosti predpone (in bi pri deljenju ostale decimalke), zaokrožimo število navzgor, npr. 234,03 KB izpišemo kot 235 KB, 234,00 KB pa kot 234 KB.

Za potrebe te naloge lahko predpostaviš, da imajo številski podatkovni tipi tvojega programskega jezika neomejen obseg in natančnost.

Primeri:

podatek	izpis
0	`0 B`
5678	`5678 B`
2097152	`2048 KB`
2097153	`2049 KB`
12897500	`13 MB`
128975000	`124 MB`

3. Lučka

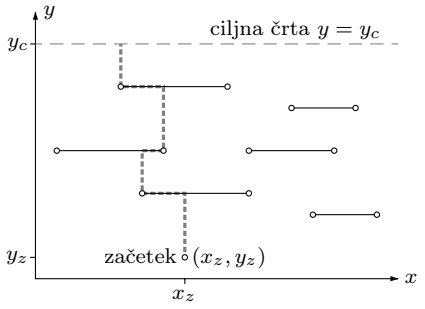
Od podjetja, ki proizvaja namizne lučke, si dobil nalogo napisati program, ki bo nastavil lučko na največjo možno svetlost. Vsaka lučka ima več stopenj svetlosti, zaradi varčevanja pa se je podjetje odločilo, da se svetlost krmili z le eno tipko. Vsakič ko pritisnemo na tipko, se svetlost poveča, razen če je lučka že na največji svetlosti, v tem primeru pa se svetlost ponastavi na najnižjo stopnjo. Ker ne vemo, na kateri stopnji je lučka in koliko stopenj ima, imamo na voljo senzor svetlosti, ki nam pove trenutno svetlost lučke. **Napiši program**, ki bo ob koncu delovanja nastavil lučko na največjo možno svetlost.

Na voljo imaš funkciji PritisniTipko() in PreveriSvetlost(). Funkcija PritisniTipko() simulira pritisk na tipko (in ne vrne ničesar), funkcija PreveriSvetlost() pa vrne trenutno svetlost lučke kot naravno število (za vsako stopnjo svetlosti vedno vrne enako vrednost). Ti dve funkciji sta že napisani in ju ne implementiraš ti. Za vse točke mora tvoj program uporabiti čim manj klicev funkcije PritisniTipko().

4. Oviratlon

Tekmovalec na oviratlonu želi preteči travnik od juga proti severu (torej od manjših y-koordinat proti večjim). Začne na koordinati (x_z, y_z), njegov cilj pa je doseči y-koordinato y_c. Pri tem mu pot ovira n vodoravnih ovir; ovira i (za $i = 1, 2, \ldots, n$) leži na y-koordinati y_i in se po x-koordinati razteza od x_{i1} do x_{i2} (pri čemer je $x_{i1} < x_{i2}$). Vse ovire ležijo po y-koordinati med začetkom in ciljem, torej za vse i velja $y_z < y_i < y_c$.

Tekmovalec bo tekel v ravni črti proti severu, dokler se ne zaleti v oviro. Obšel jo bo po levi ali desni strani, odvisno od tega, katero krajišče mu je bližje (če sta obe krajišči enako oddaljeni od njega, bo izbral levo krajišče, torej tisto z manjšo x-koordinato), in nadaljeval pot na drugi strani od krajišča ovire zopet proti severu. Ovire se med seboj ne dotikajo in se ne prekrivajo (in so dovolj daleč narazen, da gre tekmovalec vedno lahko med njimi oz. okrog njih). Ovire niso nujno podane v kakšnem posebnem vrstnem redu.

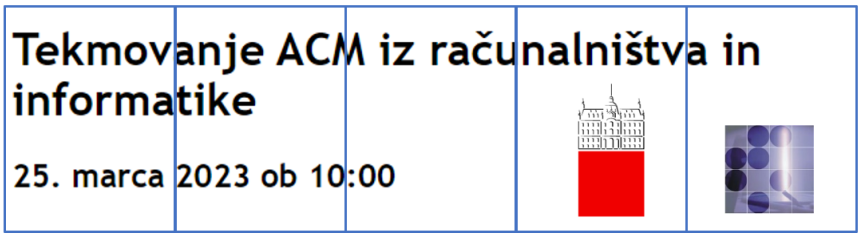


Primer koordinatnega sistema z ovirami (vodoravne daljice) in tekmovalčevo potjo (debela črtkana črta). Tekmovalec začne svojo pot v (x_z, y_z) in nadaljuje, dokler ne doseže ciljne črte $y = y_c$.

Opiši postopek (ali napiši program, če ti je lažje), ki izračuna dolžino poti, ki jo bo tekmovalec pretekel. Kot vhodne podatke tvoj postopek dobi števila x_z, y_z, y_c in n ter (za vsak i od 1 do n) y_i, x_{i1} in x_{i2}. Poleg tega, da opišeš postopek, oceni tudi njegovo časovno zahtevnost (število operacij, ki jih izvede, v odvisnosti od števila ovir n). Ovir je lahko do 100 000, zato je zaželeno, da je tvoj postopek čim bolj učinkovit.

5. Videostena

Na eni od imenitnejših srednjih šol v Ljubljani so se odločili, da bi v telovadnico želeli postaviti velik zaslon, na katerem bi predvajali reklame in druga sporočila, podobno kot vidimo pri športnih tekmovanjih. Ker nimajo denarja za nakup novih, so iz računalniške učilnice nabrali stare, še delujoče računalniške zaslone in jim dodali poceni računalnike, ki krmilijo zaslon in se pogovarjajo z nadzornim računalnikom. Zaslone so pritrdili ob rob telovadnice, pokonci, drugega zraven drugega. Nadzorni računalnik bo celotno sliko „razrezal“ in dele pošiljal na ustrezni zaslon. Primer:



zaslon 140 zaslon 378 zaslon 52 zaslon 6 zaslon 839

Seveda za ta podvig potrebujejo računalniški krožek. Tam so napisali program, ki na posameznem zaslonu ugotovi oba sosednja zaslona in to sporoči nadzornemu računalniku. Te podatke dobiš kot zaporedje trojic števil, kjer drugo število pove številko zaslona, ki sporoča to trojico, prvo število je številka njegovega levega soseda (ali -1, če levega soseda sploh ni), tretje število pa je številka njegovega desnega soseda (ali -1, če desnega soseda sploh ni). Te trojice so navedene v nekem naključnem vrstnem redu, kot kaže naslednji primer za zgornjo sliko:

```
378 52 6
6 839 -1
52 6 839
-1 140 378
140 378 52
```

Napiši program, ki iz teh podatkov ugotovi vrstni red zaslonov. Številke zaslonov so naravna števila z območja od 1 do 1000, vendar ne nujno točno od 1 do števila zaslonov (kar vidimo tudi na gornjem primeru). Podatke lahko tvoj program bere s standardnega vhoda ali pa iz datoteke `vhod.txt` (karkoli ti je lažje). (Pozor: čeprav je v primeru zgoraj pet zaslonov, naj tvoja rešitev deluje tudi za primere z več ali manj kot petimi zasloni.)

Mogoče je tudi, da v vhodnih podatkih manjka vrstica za kakšen zaslon; v tem primeru naj tvoj program izpiše sporočilo o napaki.

18. tekmovanje ACM v znanju računalništva za srednješolce

25. marca 2023

NALOGE ZA DRUGO SKUPINO

Odgovore lahko pišeš/rišeš na papir ali pa jih natipkaš z računalnikom ali pa oddaš del odgovorov na papirju in del prek računalnika. Vse te možnosti so enakovredne. Če oddajaš kaj na papirju, napiši na vsak oddani list svoje ime. Pri delu si lahko pomagaš s prevajalniki in razvojnimi orodji, ki so na voljo na tvojem računalniku, vendar bomo tvoje odgovore v vsakem primeru pregledali in ocenili ročno (ne glede na to, ali si jih oddal prek računalnika ali na papirju), zato manjše napake v sintaksi ali pri klicih funkcij standardne knjižnice niso tako pomembne, kot bi bile na tekmovanjih z avtomatskim ocenjevanjem.

Tekmovanje bo potekalo na strežniku `https://rtk.fri.uni-lj.si/`, kjer dobiš naloge in oddajaš svoje odgovore. Uporabniška imena in gesla vas bodo čakala na mizi v učilnici. Pri oddaji preko računalnika odpreš dotično nalogo v spletni učilnici in rešitev natipkaš oz. prilepiš v polje za programsko kodo. Med tipkanjem se rešitev na približno dve minuti samodejno shrani. Poleg tega lahko sam med pisanjem rešitve izrecno zahtevaš shranjevanje rešitve s pritiskom na gumb „Shrani spremembe“. Ker je vgrajeni urejevalnik dokaj preprost in ne omogoča označevanja kode z barvami, predlagamo, da rešitev pripraviš v kakšnem drugem urejevalniku na računalniku (Visual Studio Code, Geany, Lazarus) in jo nato prekopiraš v okno spletnega urejevalnika. Naj te ne moti, da se bodo barvne oznake kode pri kopiranju izgubile.

Ko si bodisi zadovoljen z rešitvijo ter si zaključil nalogo ali ko želiš začasno prekiniti pisanje rešitve naloge ter se lotiti druge naloge, uporabi gumb „Shrani spremembe“ in nato klikni na „Nazaj na seznam nalog“, da se vrneš v glavni meni. (Oddano rešitev lahko kasneje še spreminjaš.) Za vsak slučaj priporočamo, da pred oddajo shraniš svoj odgovor tudi v datoteko na svojem lokalnem računalniku.

Med reševanjem lahko vprašanja za tekmovalno komisijo postavljaš prek zasebnih sporočil na tekmovalnem strežniku (ikona oblačka zgoraj desno) ali pa vprašaš člane komisije, ki bodo prisotni v učilnicah. Prek zasebnih sporočil bomo pošiljali tudi morebitna pojasnila in popravke, če bi se izkazalo, da so v besedilu nalog kakšne nejasnosti ali napake. Zato med reševanjem redno preverjaj, če so se pojavila kakšna nova zasebna sporočila. Če imaš težave z računalnikom ali s povezavo s spletnim strežnikom za oddajo nalog in komunikacijo s tekmovalno komisijo, se nemudoma obrni na nadzornika v učilnici, ki bo zagotovil drug računalnik. **Če zaradi morebitnih težav pri oddajanju rešitev na strežnik želiš, da ocenimo odgovore v datotekah na lokalnem disku tvojega računalnika, o tem obvezno obvesti nadzorno osebo v svoji učilnici, še preden odideš iz nje.**

Svoje odgovore dobro utemelji. Če pišeš izvorno kodo programa ali podprograma, **OBVEZNO** tudi v nekaj stavkih z besedami opiši idejo, na kateri temelji tvoja rešitev. Če ni v nalogi drugače napisano, lahko tvoje rešitve predpostavljajo, da so vhodni podatki brez napak (da ustrezajo formatu in omejitvam, kot jih podaja naloga). Zaželeno je, da so tvoje rešitve poleg tega, da so pravilne, tudi učinkovite; bolj učinkovite rešitve dobijo več točk (s tem je mišljeno predvsem, naj ima rešitev učinkovit algoritem; drobne tehnične optimizacije niso tako pomembne). **Nalog je pet** in pri vsaki nalogi lahko dobiš od 0 do 20 točk. Liste z nalogami lahko po tekmovanju obdržiš.

Rešitve in rezultati bodo objavljeni na `https://rtk.ijs.si/`.

1. Stoli

Imamo n stolov v vrsti in n oseb; oboji so oštevilčeni od 1 do n (velja $n \leq 1000$). Osebe bi se rade druga za drugo posedle na stole. O vsaki osebi vemo, na kateri stol bi se najraje usedla (oseba i na stol S_i), poznamo pa tudi njihove želje v primeru, da je ta stol že zaseden. Če je želeni stol že zaseden, se oseba poskuša usesti na najbližji stol, ki ustreza tem zahtevam. Nekatere osebe so se pripravljene premakniti levo od stola, nekatere pa desno od stola. Ker bodo tisti, ki se niso uspeli usesti na želeni stol, slabe volje, imamo za vsako osebo i podano tudi število R_i (ki je večje ali enako 0), ki pove, koliko mest levo in desno od njenega novega sedišča v trenutku, ko se bo usedla, ne sme biti nikogar, da ne bo ta zlovoljna oseba širila negativne energije (oseba, ki se ni mogla usesti na želeni stol, se torej v izbrano smer premika tako daleč, dokler ni v njeni okolici dovolj prostih stolov). Tisti, ki ne morejo zasesti nobenega stola po svojih željah, v jezi odkorakajo domov in ne zasedejo nobenega stola.

Napiši program, ki ugotovi, kakšna je končna razporeditev ljudi na stole. Podatke o osebah naj prebere s standardnega vhoda. V prvi vrstici dobi število oseb n. Sledi n vrstic, za vsako osebo po ena; v i-ti od teh vrstic so s presledkom ločena števila S_i, R_i ter črka `L` ali `D`, ki predstavlja smer, v katero se želi oseba i premakniti, če je njen želeni stol S_i že zaseden.

Tvoj program naj izpiše n s presledkom ločenih števil, kjer i-to število predstavlja i-ti stol. Če na stolu ni nikogar, izpiši 0, sicer pa številko osebe (od 1 do n), ki sedi na njem.

Primer: recimo, da imamo 8 stolov; naslednja tabela kaže, kako bi se posedle prve tri osebe, če bi imele takšne želje, kot je navedeno v levem delu tabele.

oseba i	stol S_i	razmik R_i	smer	Stanje stolov po prihodu te osebe
	začetno stanje →			0 0 0 0 0 0 0 0
1	5	7	L	0 0 0 0 1 0 0 0
2	5	3	L	2 0 0 0 1 0 0 0
3	5	1	L	2 0 3 0 1 0 0 0

2. Tehtnica

Imamo tehtnico z dvema skodelicama. V levo skodelico postavimo paket z neko celoštevilčno težo, za uravnoteženje pa imamo na voljo n uteži, ki jih lahko razporedimo po obeh skodelicah. Uteži imajo teže, ki so potence števila tri, torej npr. za $n = 5$ imamo uteži s težami 1, 3, 9, 27, 81 enot, od vsake po en primerek. Vsako od uteži lahko postavimo ali v nasprotno skodelico od paketa ali v isto skodelico z njim ali pa je ne uporabimo pri uravnoteženju tehtnice.

Napiši program, ki bo prebral število uteži n, potem pa za vsako možno nenegativno celoštevilčno težo paketa, ki se jo še da uravnotežiti z danimi utežmi, izpisal, katere uteži moramo pri tem postaviti v levo skodelico in katere v desno.

Primer izpisa:

```
()    <=> ()    = 0
()    <=> (1)   = 1
(1)   <=> (3)   = 2
()    <=> (3)   = 3
()    <=> (1 3) = 4
(1 3) <=> (9)   = 5
(3)   <=> (9)   = 6
...
```

Točen format izpisa ni predpisan, tudi vrstni red ne, važno je le, da je iz izpisa nedvoumno jasno, katere uteži so položene v levo in katere v desno skodelico za vsako možno težo paketa.

3. Konkordanca

Dano je dolgo besedilo v datoteki, razdeljeno na vrstice, dolge po največ 100 znakov. **Napiši program** ali podprogram (oz. funkcijo), ki za dani niz s (sestavljen samo iz črk) poišče vse pojavitve tega niza v besedilu in vsako pojavitev izpiše skupaj z zadnjimi 30 znaki pred njo in prvimi 30 znaki za njo. Pri tem konec vrstice obravnavaj kot presledek. Predpostaviš lahko, da besedilo vsebuje le znake ASCII (črke angleške abecede, števke in ločila). Besedilo lahko bereš s standardnega vhoda ali pa iz datoteke `vhod.txt` (karkoli ti je lažje). Besedilo je lahko dolgo, zato v pomnilniku ni nujno dovolj prostora za hranjenje celotnega besedila.

Primer za niz „`drevak`“:

```
          (30 znakov)                        (30 znakov)
rogi bi bil najrajši planil v drevak in se odpeljal v konec Dolge
vredno hoditi iskat.    Nekaj drevakov se je potopilo. Le rilci so
l k vodi. Odbrali so najlepši drevak, izplali vodo, prijeli za ves
bilo jesti."      "Vidim tuje drevake," je spet spregovoril Jelen.
jak je slišal reči Sulca:    "drevak je brez veslača. Je že komu z
opil Ostrorogi v pripravljeni drevak. Vesla so se potopila v skalj
 Sinovi so pa ročno potegnili drevak na pol na breg, ko ga ni bilo
ščave. S trdine pa so veslali drevaki proti koliščem. Veliko je
obrvi. Pogled mu je obstal na drevaku, ki se je pravkar odtrgal iz
```

Opomba: če je najdena pojavitev niza preblizu začetka ali konca datoteke, lahko manjkajoče znake okolice niza nadomestiš s presledki ali pa jih ne izpišeš.

4. Nedeljiva hramba

Mikrokrmilnik periodično odčitava vrednost neke meritve, ki je nenegativno celo število. To število se da shraniti v 32-bitno spremenljivko (to so štirje bajti). Uporabnik lahko kadarkoli izklopi mikrokrmilnik in ga kasneje vklopi nazaj, lahko tudi začasno izpade napajanje. Pomembno pa je, da se vrednost zadnje meritve ohrani tudi ob izpadu, torej da ima mikrokrmilnik ob ponovnem zagonu dostop do zadnjega podatka, ki ga je pred izpadom uspel shraniti.

Lasten hitri pomnilnik za hranjenje podatka prek izpada ni uporaben, ker se njegova vsebina izgubi. Tudi dostopa do diska z datotekami ni. Pač pa imamo na voljo manjši zunanji pomnilnik, ki je sposoben trajno ohranjati svojo vsebino. Ta pomnilnik je velik 1024 bajtov (dovolj in preveč za naš namen, torej ni treba biti pretirano varčen), enota branja in pisanja vanj je en bajt, ta se hrani na naslovu med 0 in 1023.

Konstrukcija zunanjega pomnilnika zagotavlja, da se pisanje bajta izvede celovito (atomično, nedeljivo), tudi če bi sredi operacije pisanja izpadlo napajanje — vrednost bajta se bo torej bodisi zapisala v celoti na zahtevani naslov ali pa se ne bo zapisala in bo ohranjena prejšnja vrednost na tem naslovu. Ne more se torej zgoditi, da bi se zapisalo npr. le nekaj bitov od osmih ali da bi se vsebina kako drugače pokvarila.

Za dostop do trajnega pomnilnika sta na voljo funkciji:

- ShraniBajt(naslov, vrednost)
- PreberiBajt(naslov)

Obe imata kot prvi argument pomnilniški naslov (celo število med 0 in 1023). Funkcija ShraniBajt zapiše na dani naslov en bajt z dano vrednostjo (celo število med 0 in 255), funkcija PreberiBajt pa z danega naslova en bajt prebere in ga vrne (rezultat je torej tudi celo število med 0 in 255). Za tidve funkciji torej predpostavi, da že obstajata, in ju ne poskušaj implementirati sam.

(*Nadaljevanje na naslednji strani.*)

Težava, ki jo moramo rešiti, je, da je podatek z naše meritve dolg štiri bajte in tudi zanj bi potrebovali atomičen način zapisovanja, t.j. da se ob morebitnem izpadu sredi pisanja štirih bajtov ne bi zgodilo, da bi nekaj bajtov obdržalo staro vrednost, nekaj pa novo, saj bi tako dobili ob branju napačno vrednost (niti staro, niti novo, ampak nekaj pomešanega).

Premisli, kako bi lahko zagotovil atomičen zapis merilne vrednosti v zunanji pomnilnik, in **opiši** podatkovno strukturo in postopek, ki bi to lahko zagotovila.

Napiši tudi tri funkcije:

- NastaviZacetnoStanje() — ta funkcija bo poklicana le ob prvem vklopu mikrokrmilnika in nikoli več. Njena naloga je, da v zunanjem pomnilniku pripravi veljavno začetno podatkovno strukturo, kot si si jo zamislil in opisal.
- ShraniPodatek(podatek) — ta funkcija naj shrani podatek (32-bitno nenegativno celo število) v zunanji pomnilnik in pri tem pazi, da se stari in novi podatek ne bi pomešala, tudi če sredi klica te funkcije izpade delovanje mikrokrmilnika.
- PreberiPodatek() — ta funkcija naj prebere in vrne zadnjo shranjeno vrednost iz zunanjega pomnilnika.

5. Prisotnost

Podjetje je poslalo skupino programerjev na konferenco o najnovejših tehnologijah. Kljub temu, da nekatera izmed n predavanj na konferenci potekajo istočasno, pričakuje direktor podjetja, da bo vsako predavanje poslušala vsaj ena oseba iz delegacije, ki bo lahko pridobljeno znanje predala naprej v podjetju. Za vsako predavanje na konferenci je znano, v katerem časovnem intervalu poteka; i-to predavanje (za $i = 1, 2, \ldots, n$) poteka v intervalu $[z_i, k_i]$, torej od vključno časa z_i do vključno časa k_i.

Zaposleni so se odločili, da si bodo naknadno pogledali posnetke predavanj, pričakovanja direktorja pa bodo zadovoljili z vpisom na seznam prisotnih, ki je na voljo ves čas posameznega predavanja. Raje bodo šli na ogled mesta, med ogledom pa se bo vsake toliko časa nekdo vrnil na prizorišče konference. Kdo bo ta nesrečnež, bodo žrebali vsakič znova. Nesrečni izbranec se bo vpisal med prisotne na vseh predavanjih, ki takrat potekajo, in se takoj vrnil k skupini (za vpis med prisotne torej porabi 0 časa).

Opiši postopek, ki izračuna, najmanj kolikokrat bo moral nekdo od njih tako prekiniti turistični ogled, da bo na vseh predavanjih vsaj eden od njih vpisan med prisotne. Kot vhodne podatke tvoj postopek dobi število predavanj n in čase $z_1, k_1, z_2, k_2, \ldots, z_n, k_n$. Časi so podani kot cela števila v nekih zelo majhnih enotah, zato so lahko to precej velika cela števila. Število predavanj n je lahko do 10^6, zato naj bo tvoj postopek učinkovit.

18. tekmovanje ACM v znanju računalništva za srednješolce

25. marca 2023

PRAVILA TEKMOVANJA ZA TRETJO SKUPINO

Vsaka naloga zahteva, da napišeš program, ki prebere neke vhodne podatke, izračuna odgovor oz. rezultat ter ga izpiše. Programi naj berejo vhodne podatke s standardnega vhoda in izpisujejo svoje rezultate na standardni izhod. Vaše programe bomo pognali po večkrat, vsakič na drugem testnem primeru. Besedilo vsake naloge natančno določa obliko (format) vhodnih in izhodnih podatkov. Tvoji programi lahko predpostavijo, da se naši testni primeri ujemajo s pravili za obliko vhodnih podatkov, ti pa moraš zagotoviti, da se bo izpis tvojega programa ujemal s pravili za obliko izhodnih podatkov.

Tvoji programi naj bodo napisani v programskem jeziku pascal, C, C++, C#, java, python ali rust, mi pa jih bomo preverili s prevajalniki FreePascal 3.0.4, GNUjevima `gcc` in `g++` 10.3.0 (ta verzija podpira C++17, novejše različice standarda C++ pa le delno), prevajalnikom za javo iz JDK 17, s prevajalnikom Mono 6.8 za C#, s prevajalnikom `rustc` 1.57 za rust in z interpreterjem za python 3.8.

Na spletni strani `https://putka-rtk.acm.si/contests/rtk-2023-3/` najdeš opise nalog v elektronski obliki. Prek iste strani lahko oddaš tudi rešitve svojih nalog. Pred začetkom tekmovanja lahko poskusiš oddati katero od nalog iz arhiva `https://putka-rtk.acm.si/tasks/s/test-sistema/list/`. Uporabniško ime in geslo za Putko sta enaki kot za računalnike. Med tekmovanjem lahko vprašanja za tekmovalno komisijo postavljaš prek foruma na Putki (povezava pod „Pogovor o nalogi“ v okvirju „Osnovne informacije“ desno od besedila posamezne naloge).

Sistem na spletni strani bo tvojo izvorno kodo prevedel in pognal na več testnih primerih. Za vsak testni primer se bo izpisalo, ali je program pri njem odgovoril pravilno ali ne. Če se bo tvoj program s kakšnim testnim primerom ukvarjal predolgo ali pa porabil preveč pomnilnika (točne omejitve so navedene na ocenjevalnem sistemu pri besedilu vsake naloge), 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 (za podrobne nastavitve prevajalnikov na ocenjevalnem strežniku glej `https://putka-rtk.acm.si/info/`. Tvoji programi naj uporabljajo le standardne knjižnice svojega programskega jezika in naj ne delajo z datotekami na disku. Dovoljena je uporaba literature (papirnate), ne pa računalniško berljivih pripomočkov (razen tega, kar je že na voljo na tekmovalnem računalniku in na ocenjevalnem strežniku), prenosnih računalnikov, prenosnih telefonov itd.

Preden oddaš kak program, ga najprej prevedi in testiraj na svojem računalniku, oddaj pa ga šele potem, ko se ti bo zdelo, da utegne pravilno rešiti vsaj kakšen testni primer.

Ocenjevanje

Vsaka naloga ti lahko prinese od 0 do 100 točk. Vsak oddani program se preizkusi na več testnih primerih; pri tretji nalogi je testnih primerov 20 in vsak je vreden po 5 točk, pri prvi in peti pa je testnih primerov po 10 in vsak je vreden po 10 točk. Pri posameznem testnem primeru dobi program vse točke, če je izpisal pravilen odgovor, sicer pa 0 točk (izjema je tretja naloga, kjer so možne tudi delne točke pri posameznem testnem primeru). Druga in četrta naloga imata točkovanje po podnalogah, kjer dobi program vse točke za posamezno podnalogo, če pravilno reši vse testne primere tiste podnaloge, sicer pa pri tej podnalogi dobi 0 točk.

Nato se točke po vseh testnih primerih oz. podnalogah 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. Liste z nalogami lahko po tekmovanju obdržiš.

Primer naloge (ne šteje k tekmovanju)

Napiši program, ki s standardnega vhoda prebere dve celi števili (obe sta v prvi vrstici, ločeni z enim presledkom) in izpiše desetkratnik njune vsote na standardni izhod.

Primer vhoda:

```
123 456
```

Ustrezen izhod:

```
5790
```

Primeri rešitev:

- V pascalu:

```
program PoskusnaNaloga;
var i, j: integer;
begin
  ReadLn(i, j);
  WriteLn(10 * (i + j));
end. {PoskusnaNaloga}
```

- V C-ju:

```
#include <stdio.h>
int main()
{
  int i, j; scanf("%d %d", &i, &j);
  printf("%d\n", 10 * (i + j));
  return 0;
}
```

- V C++:

```
#include <iostream>
using namespace std;
int main()
{
  int i, j; cin >> i >> j;
  cout << 10 * (i + j) << '\n';
}
```

(*Opomba:* namesto `'\n'` lahko uporabimo `endl`, vendar je slednje ponavadi počasneje.)

- V pythonu:

```
import sys
L = sys.stdin.readline().split()
i = int(L[0]); j = int(L[1])
print("%d" % (10 * (i + j)))
```

- V javi:

```
import java.io.*;
import java.util.Scanner;
public class Poskus
{
  public static void main(String[] args)
                              throws IOException
  {
    Scanner fi = new Scanner(System.in);
    int i = fi.nextInt(); int j = fi.nextInt();
    System.out.println(10 * (i + j));
  }
}
```

- V C#:

```
using System;
class Program
{
  static void Main(string[] args)
  {
    string[] t = Console.In.ReadLine().Split(' ');
    int i = int.Parse(t[0]), j = int.Parse(t[1]);
    Console.Out.WriteLine("{0}", 10 * (i + j));
  }
}
```

18. tekmovanje ACM v znanju računalništva za srednješolce

25. marca 2023

NALOGE ZA TRETJO SKUPINO

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

1. Padalski izlet

Klub športnih padalcev organizira izlet za začetnike, kjer bodo padalsko izkušnjo podali novincem s skokom v tandemu. Cilj izleta je, da čim več začetnikov okusi skok s padalom. Za ta namen mora biti vsak začetnik v paru z enim izkušenim padalcem. Drugi problem izleta je prevoz — za vsakega udeleženca vemo, ali pride z avtom ali ne. Tisti, ki ne pridejo z avtom, se morajo pridružiti nekomu, ki pride z avtom. Udeleženci, ki so brez prevoza ali pa so začetniki brez mentorja, se izleta žal ne morejo udeležiti.

Dobili smo seznam prijavljenih. Za vsakega voznika vemo, koliko ljudi lahko vzame v avto. Prav tako za vsakega izkušenega padalca vemo, s koliko začetniki lahko skoči v tandemu. **Napiši program**, ki ugotovi, največ koliko začetnikov lahko okusi skok s padalom.

Opomba: ni treba, da se oseba pripelje z avtom skupaj z istim soudeležencem, s katerim potem tudi skače; lahko se pripelje z enim in skače z drugim.

Vhodni podatki: v prvi vrstici je naravno število n, število udeležencev. Nato sledi n vrstic; v i-ti od njih je opis i-tega izmed udeležencev z dvema celima številoma a_i in b_i, ločenima s presledkom. Pri tem a_i pove, koliko sopotnikov lahko i-ta oseba vzame v svoj avto. Če je $a_i = -1$, potem ta oseba nima avtomobila in se mora pridružiti nekomu drugemu. Podobno b_i pomeni, koliko so-skakalcev lahko vzame i-ta oseba. Če je $b_i = -1$, je ta oseba začetnik in se mora pridružiti bolj izkušenemu mentorju. Če je $a_i = 0$, to pomeni, da ima oseba prevoz zase, ne more pa vzeti sopotnikov; in podobno $b_i = 0$ pomeni, da ta oseba lahko skoči s padalom, ne more pa skočiti skupaj z nobenim začetnikom.

Omejitve: $1 \leq n \leq 10^5$; za vsak $i = 1, \ldots, n$ velja $-1 \leq a_i \leq 10^5$ in $-1 \leq b_i \leq 10^5$. V prvih 20 % testnih primerov bo veljalo $1 \leq n \leq 15$; v naslednjih 30 % testnih primerov bo veljalo $1 \leq n \leq 1000$; v naslednjih 30 % testnih primerov bo veljalo $a_i, b_i \in \{-1, 1\}$.

Izhodni podatki: izpiši eno samo celo število — koliko največ začetnikov (torej ljudi, za katere je $b_i = -1$) lahko pride na izlet.

Primer vhoda:

```
5
2 1
-1 1
-1 -1
-1 -1
1 -1
```

Pripadajoči izhod:

```
2
```

Še en primer vhoda:

```
8
-1 3
0 -1
-1 2
-1 -1
2 -1
-1 -1
-1 2
-1 -1
```

Pripadajoči izhod:

```
3
```

Komentar: v prvem primeru lahko na izlet vzamemo štiri osebe, izpustimo pa enega izmed začetnikov brez prostora v avtu. V drugem primeru vzamemo 1., 2., 5. osebo in eno izmed 4., 6. in 8. osebe, kar vključuje tri začetnike.

2. Uličnе luči

Dana je ulica dolžine m; položaje na njej torej lahko opišemo z x-koordinatami od 0 do m, ki merijo oddaljenost posamezne točke od levega krajišča ulice. Na ulici stoji n luči, pri čemer i-ta od njih (za $i = 1, 2, \ldots, n$) stoji na x-koordinati p_i in ima svetilnost c_i. Luči lahko osvetlimo različno močno: izberemo si neko konstanto a (ki je za vse luči enaka) in potem posamezna luč osvetljuje ulico v razdalji $a \cdot c_i$ od točke, kjer stoji; i-ta luč torej osvetljuje vse točke x z območja $p_i - a \cdot c_i \leq x \leq p_i + a \cdot c_i$. **Napiši program**, ki poišče najmanjši celoštevilski a, pri katerem bo osvetljena celotna ulica (torej: vsako točko x z območja $0 \leq x \leq m$ mora osvetljevati vsaj ena luč).

Vhodni podatki: v prvi vrstici sta celi števili n (število luči) in m (dolžina ulice), ločeni s presledkom. Sledi n vrstic, ki po vrsti opisujejo luči; i-ta od teh vrstic vsebuje celi števili p_i in c_i, ločeni s presledkom. Luči niso nujno podane v kakšnem posebnem vrstnem redu.

Omejitve: $1 \leq n \leq 10^6$; $1 \leq m \leq 10^9$; za vsak $i = 1, 2, \ldots, n$ velja $0 \leq p_i \leq m$ in $1 \leq c_i \leq m$. Vsi p_i so med seboj različni, torej nobeni dve luči nista na isti koordinati.

Točkovanje. Pri tej nalogi je pet podnalog, ki se razlikujejo po dodatnih omejitvah:

- (10 točk) $n \leq 1000$ in $m \leq 1000$;
- (20 točk) $n \leq 1000$;
- (25 točk) $c_i \leq 10$;
- (20 točk) $n \leq 2 \cdot 10^5$;
- (25 točk) brez dodatnih omejitev vhodnih podatkov.

Pri posamezni podnalogi dobiš vse točke, če pravilno rešiš vse testne primere pri njej, sicer pa pri njej ne dobiš nobene točke.

Izhodni podatki: izpiši eno samo vrstico, vanjo pa najmanjše celo število a, pri katerem je osvetljena celotna ulica.

Primer vhoda:

```
100 3
90 8
10 20
70 30
```

Pripadajoči izhod:

```
2
```

Opomba: pri tem primeru bi bila ulica v celoti osvetljena že pri $a = 5/4$, toda ker naloga zahteva celoštevilsko rešitev, je pravilni odgovor $a = 2$.

3. Špijonaža

V neki tajni službi dela n vohunov, ki so oštevilčeni s celimi števili od 1 do n. Vohuni so organizirani hierarhično: vsak vohun ima natanko enega neposredno nadrejenega, izjema je le vohun številka 1, ki nima nadrejenega in torej stoji na vrhu hierarhije.

Vohun številka 1 je prejel pomemben dokument, z vsebino katerega bi zdaj rad seznanil vse ostale vohune. Zaradi varnosti bodo vohuni širili dokument postopoma, po korakih, pri čemer se v vsakem koraku zgodi ena od naslednjih stvari:

(1) vohun, ki trenutno ima svoj izvod dokumenta, lahko naredi eno kopijo tega dokumenta in to kopijo izroči enemu od svojih neposredno podrejenih vohunov, vendar le takemu, ki takšne kopije ni prejel že kdaj prej;

(2) vohun, ki trenutno ima svoj izvod dokumenta, lahko ta izvod uniči.

V posameznem trenutku lahko torej obstaja več izvodov dokumenta — v koraku tipa 1 se število izvodov poveča za 1, v koraku tipa 2 pa zmanjša za 1. Vohuni bi radi poskrbeli, da bo vsak od njih nekoč prejel svoj izvod dokumenta, pri tem pa želijo, da hkrati obstaja čim manj izvodov (torej: da bi bilo največje število dokumentov, ki bodo kdajkoli obstajali istočasno, najmanjše možno). **Napiši program**, ki prebere podatke o hierarhiji vohunov in izpiše zaporedje korakov, s katerim lahko to dosežejo.

Vhodni podatki: v prvi vrstici je število vohunov n. Sledi $n-1$ vrstic, od katerih i-ta vsebuje številko vohuna, ki je neposredno nadrejen vohunu številka $i+1$. (To so torej podatki o neposredno nadrejenih za vohune od 2 do n; za vohuna 1 takega podatka ni, ker zanj že vemo, da neposredno nadrejenega sploh nima.) Zagotovljeno je, da je vohun številka 1 posredno ali neposredno nadrejen vsem ostalim.

Izhodni podatki: v prvo vrstico izpiši dve celi števili, ločeni s presledkom. Prvo naj bo število korakov v tvojem zaporedju, drugo pa največje število izvodov dokumenta, ki pri tvoji rešitvi kdajkoli obstajajo istočasno. Sledi naj za vsak korak tvojega zaporedja po ena vrstica, ki vsebuje dve celi števili (ločeni s presledkom), ki opisujeta ta korak:

- `1` v — korak tipa 1: vohun v prejme svoj izvod dokumenta od svojega neposredno nadrejenega vohuna;
- `2` v — korak tipa 2: vohun v uniči svoj izvod dokumenta.

Če je možnih več enako dobrih rešitev, je vseeno, katero od njih izpišeš.

Alternativa: če hočeš, lahko izpišeš le prvo vrstico, pri čemer pa namesto števila korakov napišeš -1 (glej drugi primer spodaj). Takšna rešitev dobi pri trenutnem testnem primeru 60 % točk.

Omejitve: $1 \leq n \leq 10^5$. Pri prvih 30 % testnih primerov velja tudi $n \leq 8$. Pri naslednjih 20 % testnih primerov velja $n \leq 1000$.

Primer vhoda:	Eden od možnih pripadajočih izhodov:

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

Še en primer vhoda:	Izhod za 60 % točk:

`10`
`5`
`1`
`7`
`1`
`5`
`1`
`5`
`4`
`7`
	`-1 3`

Komentar: pri rešitvi prvega primera obstajata po največ dva izvoda dokumenta hkrati. Po prvem koraku imata svoj izvod vohuna 1 in 2; v drugem koraku vohun 2 svoj izvod uniči; po tretjem koraku pa imata svoj izvod vohuna 1 in 3. Tako je vsak vohun nekoč prejel svoj izvod dokumenta, nikoli pa nista obstajala več kot dva izvoda dokumenta hkrati.

4. Valj

Dan je valj, na plašču katerega je karirasta mreža velikosti $n \times m$ (n vrstic in m stolpcev). Vsak kvadratek je pobarvan z neko barvo. Zanima nas, ali lahko valj obkrožimo, če se premikamo samo po kvadratkih ene barve. **Napiši program**, ki ugotovi, za katere barve to lahko storimo.

Valj z neko potjo obkrožimo, če pot začnemo in končamo v isti točki, pri tem pa se premaknemo naokrog celega valja. Pri tem ni nujno, da se gibljemo zgolj v eno smer (glej drugi primer spodaj).

Obhod v desno definiramo kot pot, ki se začne in konča na istem polju in kjer je vsaka navpična oz. „poldnevnik“ (meja med dvema stolpcema v mreži) valja prečkana enkrat več v desno kot v levo. Definicija obhoda v levo je simetrična. Med kvadratki se lahko premikamo čez stranice, ne pa po diagonalah.

Vhodni podatki: v prvi vrstici sta celi števili n in m, višina in širina mreže. Sledi n vrstic s po m celimi števili $a_{i,j}$, kjer vsaka vrednost predstavlja svojo barvo.

Izhodni podatki: izpiši seznam vseh barv, po katerih lahko obkrožimo valj; urejene naj bodo naraščajoče in ločene s presledkom. Če se obhoda ne da narediti po poljih nobene barve, izpiši „`mavrica`“ (brez narekovajev).

Omejitve: $2 \leq n \leq 1000$; $2 \leq m \leq 1000$; $0 \leq a_{i,j} \leq 10^6$.

Točkovanje. Pri tej nalogi so štiri podnaloge, ki se razlikujejo po dodatnih omejitvah:

- (30 točk) $2 \leq n \leq 5$ in $2 \leq m \leq 5$;
- (20 točk) $2 \leq n \leq 100$ in $2 \leq m \leq 100$;
- (20 točk) na valju bosta le dve barvi, 0 in 1;
- (30 točk) brez dodatnih omejitev vhodnih podatkov.

Pri posamezni podnalogi dobiš vse točke, če pravilno rešiš vse testne primere pri njej, sicer pa pri njej ne dobiš nobene točke.

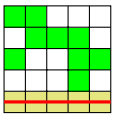
Primer vhoda:

```
5 5
1 1 0 0 0
0 1 1 1 0
1 0 0 1 1
0 0 0 1 0
2 2 2 2 2
```

Pripadajoči izhod:

```
2
```

Slika:



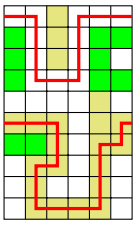
Še en primer vhoda:

```
10 6
0 0 2 0 0 0
1 0 2 0 1 1
1 0 2 0 1 0
1 0 0 0 1 1
0 0 0 0 2 0
2 2 2 0 2 2
1 1 2 0 2 2
0 2 2 0 2 0
0 2 0 0 2 0
0 2 2 2 2 0
```

Pripadajoči izhod:

```
0 2
```

Slika:



Rešitvi obeh primerov sta prikazani na slikah na desni; rdeče črte prikazujejo možne poti okrog valja.

5. Urejanje z medianami

V skladišču je v vrsti n podstavkov, oštevilčenih od 1 do n. Na vsakem podstavku stoji po en zaboj, zaboji pa so različno težki (nobena dva nista enako težka). Zaboje bi radi uredili po teži, pri čemer nam je vseeno, ali bodo urejeni naraščajoče (najlažji zaboj na podstavku 1, drugi najlažji na podstavku 2, ..., najtežji na podstavku n) ali padajoče (najtežji zaboj na podstavku 1, drugi najtežji na podstavku 2, ..., najlažji na podstavku n). Teže posameznih zabojev ne poznamo; za urejanje zabojev bomo morali uporabiti sistem robotskih rok v skladišču, ki podpira le dva tipa operacij:

1. lahko mu podamo številki dveh podstavkov in zahtevamo, naj zamenja zaboja na teh dveh podstavkih (tako da zaboj z enega podstavka pride na drugega in obratno);

2. lahko mu podamo številke treh podstavkov in zahtevamo, naj nam pove, na katerem od teh treh podstavkov stoji zaboj, ki je po teži srednji med zaboji na teh treh podstavkih (torej ki ni niti najlažji niti najtežji izmed njih).

Dodatna težava je, da se podstavki, ki sodelujejo v operaciji drugega tipa, pri tem nekoliko obrabijo. Definirajmo *obrabo* podstavka kot število operacij drugega tipa, pri katerih je bil ta podstavek udeležen. **Napiši program**, ki uredi zaboje po teži (lahko naraščajoče ali padajoče), pri tem pa pazi, da obraba nobenega podstavka ne bo prevelika.

To je interaktivna naloga; tvoj program se bo z ocenjevalnim strežnikom „pogovarjal“ tako, da bo bral s standardnega vhoda in pisal na standardni izhod. Ta pogovor naj poteka po naslednjih korakih:

1. Na začetku preberi s standardnega vhoda eno vrstico, v kateri bo celo število n (in nič drugega).

2. Nato lahko izvedeš 0 ali več operacij. Operacijo izvedeš tako, da na standardni izhod izpišeš vrstico takšne oblike:

   - $a\ b\ -1$ — če hočeš zamenjati zaboja na podstavkih a in b;
   - $a\ b\ c$ — če hočeš poizvedeti, na katerem od podstavkov a, b in c je zaboj, ki je po teži srednji med temi tremi zaboji. Pri tej operaciji moraš nato s standardnega vhoda prebrati vrstico, v kateri boš dobil odgovor (eno od števil a, b in c).

   Pri tem morajo biti a, b in (pri operaciji drugega tipa) c različna cela števila z območja od 1 do n.

3. Na koncu izpiši vrstico, v kateri naj bo le celo število -1, in prenehaj z izvajanjem.

Opozorilo: po vsaki izpisani vrstici splakni standardni izhod (*flush*), da bodo podatki res sproti prišli do ocenjevalnega sistema.

Omejitve: $1 \leq n \leq 1000$. Pri prvih 50 % testnih primerov bo veljalo tudi $n \leq 100$.

Točkovanje: če se tvoj program ne drži zgoraj opisanega protokola, ne dobi pri tem testnem primeru nobene točke; enako velja tudi, če izvede več kot 40 000 operacij ali če na koncu izvajanja zaboji niso urejeni niti naraščajoče niti padajoče. Sicer pa je število točk odvisno od obrabe najbolj obrabljenega podstavka (torej od tega, v koliko največ operacijah drugega tipa je sodeloval kakšen podstavek). Če je ta maksimalna obraba največ 20, dobiš pri tem testnem primeru 10 točk; če je od 21 do 100, dobiš 9 točk; če je od 101 do 500, dobiš 8 točk; če je od 501 do 2000, dobiš 7 točk; če pa je več kot 2000, dobiš 5 točk.

(*Nadaljevanje na naslednji strani.*)

Primer:

Tvoj program izpiše	Sistem izpiše	Komentar
	`4`	v skladišču so štirje zaboji
`2 3 4`	`4`	med podstavki 2, 3, 4 je srednji po teži zaboj na podstavku 4
`3 4 -1`		zamenjamo zaboja na podstavkih 3 in 4
`1 2 3`	`1`	med podstavki 1, 2, 3 je srednji po teži zaboj na podstavku 1
`2 1 -1`		zamenjamo zaboja na podstavkih 1 in 2
`-1`		končali smo z urejanjem

Zaboji so na koncu tega primera urejeni tako, kot naloga zahteva.

18. tekmovanje ACM v znanju računalništva za srednješolce

25. marca 2023

REŠITVE NALOG ZA PRVO SKUPINO

1. Neurejene besede

Črke posamezne besede lahko naključno premešamo takole: pojdimo v zanki po črkah besede od leve proti desni; ko smo pri k-ti črki, izberimo naključno število r z območja od 1 do k (vsa z enako verjetnostjo) in zamenjajmo k-to ter r-to črko besede (mogoče je torej tudi, da dobimo $r = k$ in torej k-ta črka ostane, kjer je bila). Če imamo n znakov dolgo besedo, je te znake načeloma mogoče premešati na $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n$ različnih načinov (ni sicer nujno, da vsi ti načini dajo različne besede — na primer, če v besedi `bob` zamenjamo prvo in tretjo črko, ostane enaka); z indukcijo po n se lahko prepričamo, da lahko prej opisani postopek premeša to besedo na vseh teh $n!$ načinov in da so vsi enako verjetni.

Zdaj torej znamo premešati znake ene besede; ker pa dobimo pri tej nalogi niz, v katerem je lahko tudi več besed, bomo postopek iz prejšnjega odstavka ovili v še eno zanko, ki pregleda celoten niz in obdela vse besede v njem. Oglejmo si implementacijo te rešitve v C++. Zunanja zanka se z i premika po znakih vhodnega niza; notranja zanka obdela trenutno besedo in ob koncu pusti i na naslednjem ne-črkovnem znaku, ki ga potem ++i v zunanji zanki preskoči. V primeru, ko je na indeksu i že ob začetku notranje zanke neki ne-črkovni znak, notranja zanka ne naredi ničesar (zunanja zanka pa se nato premake na naslednji znak).

```
#include <string>
#include <iostream>
#include <utility>
using namespace std;

void PremesajCrke(string &s)
{
  // Sprehodimo se po vhodnem nizu.
  for (int i = 0, n = s.length(); i < n; ++i)
    // Premešajmo znake naslednje besede.
    for (int od = i; i < n && isalpha(s[i]); ++i)
      // Postavimo znak i na naključno mesto med „od“ in „i“.
      swap(s[i], s[od + Random(i - od + 1)]);
}

int main()
{
  string s; getline(cin, s); // Preberimo vhodni niz.
  PremesajCrke(s); // Premešajmo črke vsake besede.
  cout << s << endl; return 0; // Izpišimo rezultat.
}
```

Tu smo uporabili funkcijo Random, za katero naloga pravi, da je že podana. Lahko pa namesto tega uporabimo generator naključnih števil iz standardne knjižnice:

```
#include <random>

void PremesajCrke2(string &s)
{
  random_device rnd;
  // Sprehodimo se po vhodnem nizu.
  for (int i = 0, n = s.length(); i < n; ++i)
```

```
    // Premešajmo znake naslednje besede.
    for (int od = i; i < n && isalpha(s[i]); ++i)
      // Postavimo znak i na naključno mesto med „od“ in „i“.
      swap(s[i], s[uniform_int_distribution(od, i)(rnd)]);
}
```

Tudi za mešanje znakov posamezne besede imamo v standardni knjižnici koristno funkcijo, std::shuffle; kot parametra ji moramo podati iteratorja, ki kažeta na prvi znak besede in prvi znak za besedo:

```
void PremesajCrke3(string &s)
{
  random_device rnd;
  // Sprehodimo se po vhodnem nizu.
  for (int i = 0, n = s.length(); i < n; ++i) {
    // Poglejmo, kje se konča trenutna beseda.
    int od = i; while (i < n && isalpha(s[i])) ++i;
    // Premešajmo znake trenutne besede.
    shuffle(s.begin() + od, s.begin() + i, rnd); }
}
```

Oglejmo si še rešitev v pythonu. Ker niza ne moremo spreminjati, ga lahko najprej predelamo v seznam, kjer je vsak znak samostojen element:

```
def PremesajCrke(s):
  i = 0; n = len(s)
  s = list(s) # Spremenimo s v seznam, da ga bomo lahko spreminjali.
  while i < n: # Sprehodimo se po vhodnem nizu.
    # Sprehodimo se po znakih naslednje besede.
    od = i
    while i < n and s[i].isalpha():
      # Postavimo znak s[i] na naključen indeks med „od“ in „i“.
      r = od + Random(i − od + 1)                                    (†)
      s[r], s[i] = s[i], s[r]
      i += 1
    i += 1 # Preskočimo trenutni ne-črkovni znak.
  return "".join(s) # Staknimo znake spet v niz.

# Preberimo niz in ga izpišimo s premešanimi znaki besed.
print(PremesajCrke(input()))
```

Tudi tu bi lahko uporabili generator naključnih števil iz pythonove knjižnice; na začetek programa bi morali dodati **import** random, nato pa vrstico (†) zamenjati z:

```
      r = random.randrange(od, i + 1)
```

Za ljubitelje pythonove standardne knjižnice je tu še enovrstična rešitev. Besede v vhodnem nizu lahko poiščemo z regularnim izrazom \w+ (torej zaporedja ene ali več črk); s funkcijo sub iz modula re lahko potem vsako besedo zamenjamo z nizom, ki ga pripravi naša vgnezdena funkcija, ki jo podamo z lambda-izrazom. Ta funkcija dobi kot parameter objekt m tipa Match, ki nam kot m[0] vrne trenutno besedo; s funkcijo sample iz modula random (ki iz danega zaporedja naključno izbira elemente brez vračanja) lahko pripravimo seznam, v katerem so vsi znaki te besede v naključno premešanem vrstnem redu, nato pa jih moramo le še stakniti skupaj v nov niz (z metodo join). Tako smo dobili naslednjo rešitev:

```
import random, re

def PremesajCrke2(s):
  return re.sub(r"\w+", lambda m: "".join(random.sample(m[0], len(m[0]))), s)
```

2. Kibi, mebi

Nalogo lahko načeloma rešimo z nekaj pogojnimi stavki: če je velikost — recimo ji x — manjša ali enaka 9999, jo lahko izpišemo kar v bajtih; sicer, če je manjša ali enaka $9999 \cdot 2^{10}$, jo lahko izpišemo v kilobajtih; sicer, če je manjša ali enaka $9999 \cdot 2^{20}$, jo lahko izpišemo v megabajtih; in tako naprej. Naloga pravi, da če x pri pretvorbi v večje enote ni celo število, ga moramo zaokrožiti navzgor na naslednje celo število. Običajno celoštevilsko deljenje pa zaokroža bodisi navzdol (npr. operator // v pythonu) bodisi proti 0 (npr. v C/C++ in podobnih jezikih), kar v našem primeru tudi pomeni navzdol; da dobimo zaokrožanje navzgor, lahko izkoristimo dejstvo, da (za celoštevilska x in d, kjer je $d > 0$) velja $\lfloor (x + d - 1)/d \rfloor = \lceil x/d \rceil$.

```
if (x <= 9999) cout << x << " B" << endl;
else if (x <= 9999 * 1024) cout << ((x + 1024 − 1) / 1024) << " KB" << endl;
else ...   // in tako naprej.
```

Namesto množenja in deljenja s 1024 (in njegovimi potencami) pa lahko uporabimo tudi operatorja za zamikanje bitov, << in >>, saj se pri množenju s 1024 (kar je 2^{10}) število zamakne za 10 bitov v levo, pri deljenju s 1024 pa za 10 bitov v desno. Tako dobimo naslednjo rešitev:

```
#include <iostream>
using namespace std;

void Izpisi(uintmax_t x)
{
  constexpr uintmax_t M = 9999;
  if (x <= M) cout << x << " B";
  else if (x <= (M << 10)) cout << ((x + (uintmax_t(1) << 10) − 1) >> 10) << " KB";
  else if (x <= (M << 20)) cout << ((x + (uintmax_t(1) << 20) − 1) >> 20) << " MB";
  else if (x <= (M << 30)) cout << ((x + (uintmax_t(1) << 30) − 1) >> 30) << " GB";
  else if (x <= (M << 40)) cout << ((x + (uintmax_t(1) << 40) − 1) >> 40) << " TB";
  else cout << ((x + (uintmax_t(1) << 50) − 1) >> 50) << " PB";
}
```

Uporabili smo največji razpoložljivi celoštevilski tip, **uintmax_t** (ki je načeloma dolg vsaj 64 bitov), da bo naloga delovala tudi za velike vrednosti x (na primer: 10000 PB je približno $2^{63,3}$ bajtov). Naloga sicer tega od nas ne zahteva, saj pravi, da smemo predpostaviti, da imajo številski podatkovni tipi neomejen obseg.

Še ena možnost je, da x pretvarjamo v večje enote postopoma: izkoristimo dejstvo, da je $\lceil x/d^2 \rceil = \lceil \lceil x/d \rceil / d \rceil$, torej lahko v vsakem koraku delimo x s 1024 in rezultat zaokrožimo navzgor. Tako dobimo:

```
void Izpisi2(uintmax_t x)
{
  if (x <= 9999) { cout << x << " B"; return; }
  x = (x + 1023) / 1024; if (x <= 9999) { cout << x << " KB"; return; }
  x = (x + 1023) / 1024; if (x <= 9999) { cout << x << " MB"; return; }
  x = (x + 1023) / 1024; if (x <= 9999) { cout << x << " GB"; return; }
  x = (x + 1023) / 1024; if (x <= 9999) { cout << x << " TB"; return; }
  x = (x + 1023) / 1024; cout << x << " PB";
}
```

Ker pa so si posamezne vrstice naše rešitve tako podobne, jo lahko elegantno zapišemo tudi z zanko:

```
void Izpisi3(uintmax_t x)
{
  // Pripravimo si tabelo predpon.
  static constexpr const char *predpone[] = {"", "K", "M", "G", "T", "P"};
  static constexpr int stPredpon = sizeof(predpone) / sizeof(predpone[0]);

  // Delimo x s 1024 (če se deljenje ne izide, zaokrožimo navzgor),
  // dokler ne pade pod 10000 ali pa ne pridemo do zadnje predpone.
```

```
  int i = 0; while (x > 9999 && i + 1 < stPredpon) x = (x + 1023) / 1024, ++i;
  // Izpišimo x s trenutno predpono.
  cout << x << " " << predpone[i] << "B" << endl;
}
```

Zapišimo našo rešitev še v pythonu:

```
def IzpisiVelikost(x):
  for predpona in "|K|M|G|T|P".split("|"):
    # Poglejmo, če je x dovolj kratek (ali pa smo pri zadnji predponi).
    if x <= 9999 or predpona == "P": break
    # Pretvorimo x v naslednjo večjo enoto.
    x = (x + 1023) // 1024
  # Izpišimo x v izbranih enotah.
  print("%d %sB" % (x, predpona))
```

3. Lučka

Ko pritisnemo na tipko, se svetlost lučke poveča — razen če je bila že pred pritiskom na najvišji stopnji, tedaj pa se zmanjša. Najvišjo stopnjo svetlosti prepoznamo torej po tem, da če takrat pritisnemo na tipko, se svetlost zmanjša namesto poveča. Tako lahko torej v zanki pritiskamo na tipko in merimo svetlost, dokler ne opazimo, da se je po zadnjem pritisku svetlost zmanjšala. Takrat vemo, da je bila lučka pred tem zadnjim pritiskom na najvišji možni svetlosti; zdaj, po pritisku, pa je na *najnižji* svetlosti. Toda naloga od nas zahteva, da mora biti ob koncu izvajanja našega programa lučka na najvišji svetlosti, torej moramo še enkrat v zanki pritiskati tipko, dokler spet ne dosežemo najvišje svetlosti.

```
int main()
{
  int maxSvetlost = PreveriSvetlost();
  // Pritiskajmo tipko, dokler se svetlost še povečuje.
  while (true) {
    PritisniTipko();
    if (PreveriSvetlost() <= maxSvetlost) break;
    maxSvetlost = PreveriSvetlost(); }
  // Pritiskajmo tipko, dokler spet ne dosežemo najvišje svetlosti.
  while (PreveriSvetlost() != maxSvetlost)
    PritisniTipko();
  return 0;
}
```

Zapišimo to rešitev še v pythonu:

```
maxSvetlost = PreveriSvetlost()
# Pritiskajmo tipko, dokler se svetlost še povečuje.
while True:
  PritisniTipko()
  if PreveriSvetlost() <= maxSvetlost: break
  maxSvetlost = PreveriSvetlost()
# Pritiskajmo tipko, dokler spet ne dosežemo najvišje svetlosti.
while PreveriSvetlost() != maxSvetlost:
  PritisniTipko()
```

4. Oviratlon

Tekmovalec se premika v smeri naraščajočih y-koordinat, zato bo imela vsaka naslednja ovira, na katero bo naletel, višjo y-koordinato kot prejšnja. Zato je koristno za začetek urediti vse ovire naraščajoče po y_i (kajti besedilo naloge pravi, da v vhodnih podatkih niso nujno urejene) in jih potem pregledovati v tem vrstnem redu.

Pri vsaki oviri se zdaj vprašajmo, ali se bo naš tekmovalec zaletel vanjo ali ne. Če je tekmovalec trenutno na (x, y), ovira pa gre od (x_{i1}, y_i) do (x_{i2}, y_i), se bo zaletel vanjo v primeru, ko je $x_{i1} < x < x_{i2}$; takrat moramo pogledati, katero krajišče mu je bližje — torej katera od razdalj $x - x_{i1}$ in $x_{i2} - x$ je manjša — in tekmovalca premakniti v tisto krajišče. Tako lahko postopoma sledimo njegovi poti in tudi seštevamo dolžine vseh premikov; na koncu pa ne pozabimo prišteti še dolžine premika od zadnjega položaja do ciljne črte.

Zapišimo ta postopek s psevdokodo:

uredi ovire naraščajoče po y-koordinati in jih v tem vrstnem redu oštevilči;
$x := x_z$; $y := y_z$; $d := 0$; (* *Trenutni položaj tekmovalca in dolžina poti.* *)
for $i := 1$ **to** n: (* *Pojdimo v zanki po vseh ovirah.* *)
  (* *Ali se tekmovalec zaleti v to oviro?* *)
  if $y_i < y$ **or** $x_{i1} < x$ **or** $x_{i2} > x$ **then continue**;
  (* *Premaknimo ga do ovire.* *)
  $d := d + y_i - y$; $y := y_i$;
  (* *Premaknimo ga v bližje krajišče ovire (oz. v levo, če sta obe enako oddaljeni).* *)
  if $x - x_{i1} \le x_{i2} - x$ **then** $d := d + x - x_{i1}$; $x := x_{i1}$;
                else $d := d + x_{i2} - x$; $x := x_{i2}$;
(* *Pretečeni poti dodajmo še razdaljo do ciljne črte.* *)
return $d + y_c - y$;

Časovna zahtevnost tega postopka je $O(n \log n)$ zaradi urejanja ovir po y-koordinati; ko so ovire enkrat urejene, nam preostanek postopka vzame le $O(n)$ časa, saj imamo v naši zanki z vsako oviro le konstantno mnogo dela.

5. Videostena

Ko prebiramo podatke s standardnega vhoda, lahko v neko tabelo ali vektor zapisujemo za vsak zaslon številko njegovega desnega soseda. Poleg tega si tudi zapomnimo, kateri zaslon ni imel levega soseda (torej je imel tam -1), kajti tisto je potem najbolj levi zaslon. Ko preberemo vse vhodne podatke, začnemo pri najbolj levem zaslonu in se potem s pomočjo prej omenjene tabele na vsakem koraku premaknemo s trenutnega zaslona na njegovega desnega soseda; tako lahko sledimo zaslonom od leve proti desni in jih izpisujemo. Ustavimo se, ko trenutni zaslon nima desnega soseda (torej ko je tam -1).

Naloga pravi, da moramo paziti še na možnost, da podatki za kakšen zaslon manjkajo. Ena možnost je, da manjka najbolj levi zaslon; to bomo prepoznali po tem, da pri nobenem zaslonu kot njegov levi sosed ni navedeno število -1. Če pa manjka kak kasnejši zaslon, bomo to opazili na koncu pri sprehajanju po zaslonih od leve proti desni: kot desnega soseda prejšnjega zaslona lahko dobimo neki zaslon, za katerega nismo prebrali podatka o *njegovem* desnem sosedu. V naši tabeli desnih sosedov moramo torej znati ločiti med primerom, ko neki zaslon nima desnega soseda, in primerom, ko za neki zaslon nimamo podatka o tem, kdo je njegov desni sosed oz. ali ga sploh ima. V ta namen spodnji program uporablja vrednost 0 za manjkajoče podatke in -1 za primere, ko vemo, da desnega soseda ni.

```
#include <cstdio>
#include <vector>                                                    // (1)
using namespace std;

int main()
{
  enum { M = 1000, MANJKA = 0 };
  vector<int> naslednji(M + 1, MANJKA); // naslednji[z] = desni sosed zaslona z    // (2)
  int prvi = MANJKA; // najbolj levi zaslon

  while (true)
  {
    // Preberimo podatke o še enem zaslonu.
    int levi, z, desni; if (scanf("%d %d %d", &levi, &z, &desni) != 3) break;
    if (levi < 0) prvi = z; // Če nima levega soseda, je to najbolj levi zaslon.
```

```
    naslednji[z] = desni;   // Zapomnimo si njegovega desnega soseda.
  }
  // Naštejmo zaslone od leve proti desni.
  if (prvi == MANJKA) { fprintf(stderr, "Manjka najbolj levi zaslon.\n"); return 1; }
  for (int z = prvi; z >= 1; z = naslednji[z])
  {
    if (naslednji[z] == MANJKA) {                                          // (3)
      fprintf(stderr, "Manjka zaslon %d.\n", z); return 2; }
    printf("%d ", z);
  }
  printf("\n"); return 0;
}
```

Če bi bile številke zaslonov večje (npr. do 10^9 namesto le do 1000), bi takšna rešitev s tabelo oz. vektorjem porabila preveč pomnilnika. Takrat bi bilo bolje uporabiti razpršeno tabelo oz. slovar, na primer razred map iz C++ove standardne knjižnice. Vse, kar bi morali v zgornji rešitvi spremeniti, sta vrstici (1) in (2):

```
  #include <map>                                                           // (1)
  ⋮
  map<int, int> naslednji; // naslednji[z] = desni sosed zaslona z         // (2)
```

Vrstica (3) bi še vedno delovala pravilno, kajti če ključa z takrat v slovarju še ni, ga bo map::**operator** [] dodal s pripadajočo vrednostjo 0 (kar je ravno enako naši konstanti MANJKA).

Oglejmo si še rešitev v pythonu. Tudi tu bomo uporabili slovar, saj je delo z njimi v pythonu zelo preprosto:

```
import sys
prvi = −1; naslednji = {}
# Preberimo podatke o zaslonih.
for vrstica in sys.stdin:
  [levi, z, desni] = [int(s) for s in vrstica.split()]
  if levi < 0: prvi = z    # Če nima levega soseda, je to najbolj levi zaslon.
  naslednji[z] = desni     # Zapomnimo si njegovega desnega soseda.
# Naštejmo zaslone od leve proti desni.
if prvi < 0: sys.stderr.write("Manjka najbolj levi zaslon.\n"); sys.exit(1)
z = prvi
while z > 0:
  if z not in naslednji: sys.stderr.write("Manjka zaslon %d.\n" % z); sys.exit(2)
  sys.stdout.write("%d " % z); z = naslednji[z]
sys.stdout.write("\n")
```

REŠITVE NALOG ZA DRUGO SKUPINO

1. Stoli

Ker je n (število stolov in ljudi) pri tej nalogi majhen, lahko dogajanje preprosto odsimuliramo. Pri tem bomo vzdrževali tabelo oz. vektor celih števil, ki za vsak stol povedo, kdo sedi na njem (če sploh kdo; vrednost 0 pomeni prazen stol). V glavni zanki berimo podatke o osebah; pri vsaki najprej preverimo, če je stol S_i prost; če je, lahko sede nanj, sicer pa moramo iti v notranji zanki po stolih od S_i v želeno smer in šteti, koliko prostih stolov skupaj vidimo.

Če zagledamo skupino $2R_i+1$ strnjenih prostih stolov, lahko oseba i sede na srednjega od teh stolov; če pa pridemo do začetka oz. konca vrste (odvisno od tega, ali smo šli levo ali desno), preverimo, ali smo takrat videli vsaj $R_i + 1$ strnjenih stolov, in če smo jih, se lahko oseba i usede na $(R_i + 1)$-vega od njih (gledano iz smeri, iz katere smo prišli). Naloga namreč ne zahteva, da mora biti na vsaki strani stola, kamor bi se i usedel, R_i

praznih stolov, pač pa le, da nihče drug ne sme sedeti manj kot R_i mest levo ali desno; temu pogoju lahko ustrežemo bodisi s praznimi stoli bodisi s stem, da stolov sploh ni (ker smo že na začetku oz. koncu naše vrste n stolov).

```
#include <iostream>
#include <vector>
using namespace std;

int main()
{
  // Preberimo n in pripravimo vektor praznih stolov.
  int n; cin >> n; vector<int> stoli(n + 1, 0);

  // Preberimo in obdelajmo podatke o osebah.
  for (int i = 1; i <= n; ++i)
  {
    // Preberimo naslednjo osebo.
    int Si, Ri; char smer; cin >> Si >> Ri >> smer;
    int d = (smer == 'D') ? 1 : -1;

    // Ali lahko sede na želeni stol?
    if (stoli[Si] == 0) { stoli[Si] = i; continue; }

    // Sicer se premikajmo v želeno smer.
    for (int stol = Si, stProstih = 0; 1 <= stol && stol <= n; stol += d)
    {
      // Štejmo, kako dolgo strnjeno skupino prostih stolov imamo.
      if (stoli[stol] == 0) ++stProstih; else stProstih = 0;

      // Ali lahko sede tako, da bo imel na vsaki strani Ri prostih stolov?
      if (stProstih >= 2 * Ri + 1) { stoli[stol - d * Ri] = i; break; }

      // Če smo prišli do začetka/konca vrstice, je dovolj že, če je
      // vsaj Ri prostih stolov v smeri, iz katere smo prišli.
      if (stol == (d > 0 ? n : 1) && stProstih >= Ri + 1) {
        stoli[stol - d * (stProstih - Ri - 1)] = i; break; }
    }
  }

  // Izpišimo končno stanje.
  for (int j = 1; j <= n; ++j) cout << stoli[j];
  cout << endl; return 0;
}
```

Ta rešitev ima časovno zahtevnost $O(n^2)$. Če bi hoteli učinkovitejšo rešitev za večje n, bi bilo koristno vzdrževati strnjene skupine prostih stolov v neki primerno uravnoteženi drevesasti podatkovni strukturi (npr. drevo intervalov ali pa rdeče-črno drevo), kjer bi morali v vsakem notranjem vozlišču tudi vzdrževati dolžino najdaljše skupine v poddrevesu tistega vozlišča. Tako bi lahko v $O(\log n)$ časa poiskali najbližjo dovolj dolgo skupino; ko se oseba i usede na enega od stolov te skupine, pa bi skupina razpadla na dve krajši, torej bi jo morali pobrisati iz drevesa in dodati tisti dve, na kateri je razpadla; tudi to bi šlo v $O(\log n)$ časa. Ker bi morali to narediti pri vsaki osebi, bi imela takšna rešitev časovno zahtevnost $O(n \log n)$.

2. Tehtnica

Za vsako utež imamo tri možnosti: lahko jo damo v levo skodelico, lahko v desno, lahko pa je sploh ne uporabimo (to lahko prikladno opišemo s števili -1, 1 in 0). Ker imamo n uteži, se nam tako nabere $3 \cdot 3 \cdot \ldots \cdot 3 = 3^n$ kombinacij tega, kaj naredimo s katero utežjo. Vse te možnosti lahko pregledamo z rekurzivnim podprogramom, ki za trenutno utež k (s težo 3^k) v zanki preizkusi vse tri možnosti in za vsako izvede vgnezden rekurzivni klic, ki bo pregledal vse možne kombinacije tega, kaj narediti s preostalimi utežmi. Spotoma računajmo tudi vsoto že uporabljenih uteži — tiste na desni strani prištevajmo, tiste na levi pa odštevajmo. Rekurzija se neha gnezditi, ko pridemo do $k = 0$ (utež s težo 1); takrat izpišemo trenutni razpored uteži in vsoto njihovih tež.

Paziti moramo, da ne izpišemo razporedov z negativno vsoto. To bi lahko načeloma preverili tik pred izpisom, še bolje pa je, če si pomagamo z naslednjim opažanjem:

najtežja uporabljena utež mora biti na desni strani, da jo bomo prišteli, kajti ona je več kot še enkrat težja od vseh lažjih uteži skupaj (utež k je težka 3^k, vse lažje uteži skupaj pa so težke $1+3+9+\ldots+3^{k-1}=\sum_{i=0}^{k-1}3^i=(3^k-1)/2$, torej za slabo polovico uteži k) in če jo odštejemo, niti vse lažje uteži skupaj ne bodo mogle spraviti vsote nazaj nad 0 (niti do 0).

Spodobi se tudi razmisliti, da naša rešitev ne izpiše kakšne teže po večkrat. Ali je mogoče isto vsoto tež dobiti pri dveh različnih razporedih uteži? Pa vzemimo v mislih dva takšna razporeda in naj bo k najtežja utež, ki jo tadva razporeda uporabita različno. Zaradi tega med njunima težama nastopi razlika vsaj 3^k; te razlike pa lažje uteži ne morejo izničiti, kajti tudi če jih en razpored vse uporabi v nasprotni skodelici kot drugi, bo razliko v težah to spremenilo za največ $2\sum_{i=0}^{k-1}3^i=3^k-1$, torej manj od 3^k. Čim se torej razporeda pri rabi neke uteži razlikujeta, bo med njunima težama nastala razlika, ki je z lažjimi utežmi ne bomo mogli izničiti; torej imata različna razporeda neizogibno tudi različno težo, zato se ne more zgoditi, da bi naš rekurzivni postopek izpisal isto težo po večkrat.

```
#include <iostream>
#include <vector>
using namespace std;

int n; // število uteži
vector<int> utezi, kako; // kako[k] pove, kako smo uporabili utež s težo utezi[k]

void Rekurzija(int k, int vsotaDoslej)
{
  // Preglejmo vse tri možnosti glede tega, kako uporabimo utež k.
  // Pazimo le na to, da prve (= najtežje) uporabljene uteži ne smemo
  // odštevati, pač pa le prištevati, saj bo drugače končna vsota negativna.
  for (kako[k] = (vsotaDoslej == 0 ? 0 : -1); kako[k] <= 1; ++kako[k])
  {
    int vsota = vsotaDoslej + kako[k] * utezi[k];
    if (k > 0) { Rekurzija(k - 1, vsota); continue; }
    // Pri k = 0 izpišimo rezultate.
    cout << "( ";
    for (int i = 0; i < n; ++i) if (kako[i] < 0) cout << utezi[i] << " ";
    cout << ") <=> ( ";
    for (int i = 0; i < n; ++i) if (kako[i] > 0) cout << utezi[i] << " ";
    cout << ") = " << vsota << endl;
  }
}

int main()
{
  // Preberimo število uteži in izračunajmo njihove teže.
  cin >> n; kako.resize(n); utezi.resize(n);
  utezi[0] = 1; for (int i = 1; i < n; ++i) utezi[i] = 3 * utezi[i - 1];
  // Z rekurzijo preglejmo vse možnosti.
  Rekurzija(n - 1, 0); return 0;
}
```

Ker gre naša rekurzija po padajočih k (od težjih uteži k lažjim) in pri vsakem k po naraščajočih kako[k], bodo tudi teže tako dobljenih razporedov nastajale v naraščajočem vrstnem redu, od 0 do največje možne teže $(3^n-1)/2$.

3. Konkordanca

Recimo, da je niz s, čigar pojavitve nas zanimajo, dolg n znakov in da hočemo pri vsaki pojavitvi izpisati še prejšnjih in naslednjih $d=30$ znakov. Po vhodni datoteki se bomo sprehajali z „oknom“, dolgim $m=n+2d$ znakov — na sredi okna je torej n znakov, kjer bomo gledali, ali se tam pojavi niz s ali ne, levo in desno od tega pa je še po d znakov, ki jih potrebujemo za izpis. Na vsakem koraku torej preverimo, ali se s pojavlja pri trenutnem položaju okna; če se, ga izpišemo; nato pa v vsakem primeru premaknemo

okno za eno mesto naprej — takrat pride na desni v okno nov znak, na levi pa en znak izpade iz okna.

Da bo ta rešitev učinkovita, je koristno za predstavitev okna uporabiti neke vrste krožno tabelo (*ring buffer*): okno bo sicer tabela m znakov, pri čemer pa zdaj za vsak k velja, da k-ti znak vhodne datoteke (kadar je prisoten v oknu) hranimo v tej tabeli na indeksu $k \bmod m$. To pomeni, da ko se okno premakne naprej po vhodni datoteki, ni treba v naši tabeli v resnici ničesar spremeniti, le zapomniti si je treba, kje v tabeli se začne vsebina okna.

Oglejmo si implementacijo takšne rešitve v jeziku C++. Spremenljivka p bo hranila tisti indeks v tabeli okno, kjer gledamo, ali se tam nahaja pojavitev niza s. Pred p je torej v oknu še d znakov levega kontesta, od p naprej pa $n + d$ znakov (dovolj za eno pojavitev s-ja in za d znakov desnega konteksta); okno tako pravzaprav sestavljajo znaki okno$[(p + i) \bmod m]$ za $-d \leq i < n + d$. Nekaj pazljivosti je potrebne še na koncu datoteke, kjer morda od p-ja naprej sploh ni več $n + d$ znakov, ker se datoteka konča že prej. Če je od p-ja naprej celo manj kot n znakov, potem gotovo ne bomo našli nobene pojavitve s-ja več in lahko takoj končamo; sicer pa le pazimo pri izpisu desnega konteksta, da bomo namesto manjkajočih znakov napisali presledke.

Za branje iz vhodne datoteke poskrbimo na začetku vsake iteracije glavne zanke, kjer skušamo poskrbeti, da bomo imeli v oknu od p naprej še $n+d$ znakov (manj kot to pa le, če se datoteka tam že konča). Načeloma to pomeni, da po vsakem premiku okna naprej (torej: po vsakem povečanju p-ja za 1) preberemo po en nov znak iz vhodne datoteke, le na začetku izvajanja programa bomo prebrali do $n + 2d$ znakov, da napolnimo celo okno.

```
#include <cstdio>
#include <string>
using namespace std;

void Konkordanca(const string &s, int d = 30)
{
  // Pripravimo okno z dovolj prostora za en izvod niza s
  // in še d znakov levo in desno od njega.
  int n = s.length();
  int m = 2 * d + n; string okno(m, ' ');

  // p je trenutni položaj v oknu, kjer preverjamo, ali se tam začne
  // pojavitev s-ja. „stVeljavnih“ je število veljavnih znakov od p naprej.
  int p = 0, stVeljavnih = 0;
  while (true)
  {
    // Preberimo naslednji znak.
    int c = fgetc(stdin);
    if (c != EOF) {
      okno[(p + stVeljavnih++) % m] = (c == '\n' ? ' ' : c);
      if (stVeljavnih < n + d) continue; }

    // Tu imamo od p-ja naprej bodisi n + d veljavnih znakov
    // bodisi ves preostanek datoteke, če ga je manj kot n + d znakov.
    // Poglejmo, ali se tu začne pojavitev s-ja.
    if (stVeljavnih < n) break;
    int i = 0; while (i < n && s[i] == okno[(p + i) % m]) ++i;

    // Če smo našli pojavitev s-ja, jo izpišimo s kontekstom vred.
    if (i == n) {
      for (int i = -d; i <= n + d; ++i)
        fputc(i >= stVeljavnih ? ' ' : okno[(p + i + m) % m], stdout);
      fputc('\n', stdout); }

    p = (p + 1) % m; --stVeljavnih; // Premaknimo se naprej po oknu.
  }
}
```

To, ali se na sredi okna (pri njegovem trenutnem položaju) nahaja pojavitev niza s, smo preverjali preprosto z zanko, ki primerja istoležne znake v oknu in v s-ju. To bi se dalo

seveda še izboljšati s prijemi iz raznih znanih algoritmov, kot so Knuth-Morris-Prattov, Boyer-Mooreov ali Rabin-Karpov, vendar poudarek naše naloge ni na tem.

4. Nedeljiva hramba

Podatek, ki ga želimo shraniti, je dolg štiri bajte, mi pa lahko atomično zapisujemo le po en bajt naenkrat; torej ne moremo zagotoviti, da nas ne bo med shranjevanjem kaj prekinilo, še preden bomo zapisali vse štiri bajte. Če nočemo, da v pomnilniku takrat ostane neka mešanica stare in nove vrednosti, to pomeni, da ne smemo nobenega dela stare vrednosti spremeniti ali povoziti, dokler ni nova vrednost v celoti zapisana v pomnilnik. Koristno je torej, če nove vrednosti ne pišemo čez staro, pač pa nekam drugam; v pomnilniku moramo imeti pripravljen prostor za dve vrednosti hkrati. Eno od njiju hranimo na primer na naslovih od 0 do 3, drugo pa od 4 do 7. Poleg tega moramo nekje — recimo kar na naslovu 8 — hraniti še podatek o tem, katera od obeh vrednosti je novejša.

Podprogram ShraniPodatek bo torej pogledal na naslov 8, katera od vrednosti je novejša, in potem z novim podatkom ne bo povozil nje, pač pa tisto drugo; in šele ko bo novi podatek v celoti shranil, bo na naslovu 8 zapisal, katera vrednost je zdaj novejša. Podprogram PreberiPodatek pa najprej prebere bajt z naslova 8, da vidi, katera vrednost je novejša, in potem prebere vse štiri bajte od tam in jih združi v eno samo 32-bitno celoštevilsko vrednost.

```
void NastaviZacetnoStanje()
{
  ShraniBajt(8, 0);
}

void ShraniPodatek(unsigned int podatek)
{
  // Novi podatek zapišimo na drugo lokacijo od dosedanjega.
  int kam = (PreberiBajt(8) == 0 ? 4 : 0);

  // Zapišimo ga po en bajt naenkrat.
  for (int i = 0; i < 4; ++i, podatek >>= 8)
    ShraniBajt(kam + i, podatek & 0xff);

  // Zapomnimo si, kam smo ga zapisali.
  ShraniBajt(8, kam);
}

unsigned int PreberiPodatek()
{
  // Poglejmo, kje lahko preberemo najnovejši podatek.
  int odKod = PreberiBajt(8);

  // Preberimo ga po en bajt naenkrat.
  unsigned int podatek = 0;
  for (int i = 0; i < 4; ++i)
    podatek = (podatek << 8) | PreberiBajt(odKod + 3 - i);

  return podatek;
}
```

Če uporabnik pokliče PreberiPodatek pred prvim klicem ShraniPodatek, bo dobil neko vrednost, ki je bila od prej pač slučajno v pomnilniku na naslovih od 0 do 3. Še ena možnost bi bila, da bi NastaviZacetnoStanje inicializiral bajt 8 na neko tretjo vrednost (niti 0 niti 4), kar bi PreberiPodatek lahko preveril in v tem primeru javil napako (sprožil izjemo ali kaj podobnega).

5. Prisotnost

Uredimo za začetek predavanja naraščajoče po času konca in jih v tem vrstnem redu oštevilčimo; odslej bomo torej predpostavili, da je $k_1 \leq k_2 \leq \ldots \leq k_n$. Prvi (najzgodnejši) vpis se ne sme zgoditi kasneje kot ob času k_1, saj se kasneje ne bo več mogoče vpisati med prisotne na prvem predavanju. Nobene koristi pa ni od tega, da bi se ta

vpis zgodil kaj prej kot ob k_1, saj vsako predavanje, ki bi ga pokrili ob času $t < k_1$, poteka tudi še v času k_1 (kajti če ne bi, bi to pomenilo, da se je končalo pred časom k_1, mi pa smo predavanja uredili po času konca in torej vemo, da se nobeno predavanje ne konča prej kot ob k_1).

Ko smo tako izbrali čas prvega vpisa in z njim pokrili nekaj predavanj (vsaj tisto od z_1 do k_1, morda pa še kakšno drugo), lahko ta predavanja v mislih pobrišemo, saj nam je vseeno, ali jih bodo kasnejši vpisi še kaj pokrivali ali ne; dovolj je že, da so bila pokrita enkrat. Tako nam ostane neka manjša množica predavanj, pri kateri lahko razmišljamo enako kot na začetku: prvi naslednji vpis se mora zgoditi ob najzgodnejšem času, ko se konča kakšno od preostalih predavanj (če bi se zgodil kasneje, bi tisto predavanje zgrešili, če pa bi se zgodil prej, ne bi s tem ničesar pridobili). Ker imamo predavanja urejena po času konca, je dovolj, če poiščemo prvo tako predavanje, ki ima $z_i > k_1$; za ta i potem uporabimo k_i kot čas drugega vpisa. Po tem postopku lahko zdaj nadaljujemo in poiščemo prvo tako predavanje j, ki ima $z_j > k_i$ in za ta j potem uporabimo k_j kot čas tretjega vtisa; in tako naprej, dokler niso pokrita vsa predavanja.

Zapišimo ta postopek še s psevdokodo:

```
uredi tekmovanja naraščajoče po času konca, tako da bo k_1 ≤ k_2 ≤ ... ≤ k_n;
r := 0; v := −∞; (* r je število vpisov, v je čas zadnjega vpisa doslej *)
for i := 1 to n:
  if r = 0 or z_i > v:
    r := r + 1; v := k_i; (* potreben je vpis ob času v *)
return r;
```

Časovna zahtevnost te rešitve je $O(n \log n)$ zaradi urejanja po k_i v prvem koraku. Še ena možnost bi bila, da bi pustili predavanja neurejena in šli po vsakem vpisu z zanko po vseh predavanjih, da bi ugotovili, katero med tistimi, ki se začnejo po tem vpisu, ima najzgodnejši čas konca. Takšna rešitev bi imela časovno zahtevnost $O(n \cdot r)$, kjer je r najmanjše potrebno število vpisov; ta rešitev je lahko boljša od prejšnje, če je r dovolj majhen, v najslabšem primeru pa je seveda r lahko tudi $O(n)$ in bo ta rešitev veliko slabša, reda $O(n^2)$.

REŠITVE NALOG ZA TRETJO SKUPINO

1. Padalski izlet

Vzemimo za začetek na izlet vse padalce, ki imajo svoj avto, torej ki imajo $a_i \geq 0$; recimo, da je med njimi Z začetnikov, da izkušeni med njimi lahko skočijo z vsega skupaj B začetniki in da oboji skupaj lahko prepeljejo P potnikov:

$$Z = |\{i : a_i \geq 0, b_i < 0\}, \quad B = \textstyle\sum_{i:a_i \geq 0, b_i \geq 0} b_i, \quad P = \textstyle\sum_{i:a_i \geq 0} a_i.$$

Zdaj lahko izberemo še največ P potnikov. Pri tem lahko na vsakem koraku razmišljamo takole: če je $Z < B$, lahko skočijo vsi dosedanji začetniki in zmogljivosti izkušenih padalcev še niso izkoriščene, zato je bolje vzeti za naslednjega potnika še kakšnega začetnika; če pa je $Z \geq B$, moramo najprej dodati še kakšnega izkušenega potnika, kajti doslej izbrani izkušeni padalci tako ali tako ne morejo poskrbeti za več kot Z začetnikov. Pri drugem primeru izbirajmo izkušene potnike po padajočih b_i, tako da bomo s čim manj izkušenimi potniki lahko poskrbeli za čim več začetnikov. Tako smo dobili naslednji postopek:

while $P > 0$:
  if $Z < B$:
    če ni nobenega začetnika več, končaj;
    sicer vzemi za naslednjega potnika enega od začetnikov;
    $Z := Z + 1$; $P := P - 1$;
  else:
    če ni nobenega izkušenega več, končaj;
    sicer vzemi za naslednjega potnika enega od izkušenih z največjim b_i;
    $B := B + b_i$; $P := P - 1$;

Vidimo lahko, da ta postopek nikoli ne poveča Z-ja tako, da bi bil ta potem večji od B; na koncu postopka lahko torej $Z > B$ velja le v primeru, če je veljala že na začetku. To se torej zgodi takrat, ko smo vsa potniška mesta zapolnili z izkušenimi padalci (ali pa je izkušenih celo zmanjkalo, še preden smo zapolnili vsa potniška mesta), pa jih še vedno ni dovolj, da bi poskrbeli za vse tiste začetnike, ki imajo svoj avto. Naloga pravi, da se začetniki brez mentorja ne morejo udeležiti izleta; torej bomo morali $Z - B$ od naših začetnikov z avtomobili pustiti doma. Zato se bo število potnikov, ki jih lahko prepeljemo, morda kaj zmanjšalo; ali nam bo to povzročilo kakšne težave? Prepričajmo se, da ni tako. Za začetek opazimo, da če je med potniki (ki so v tem scenariju sami izkušeni) kakšen z $b_i = 0$, ni od njega nobene koristi in ga lahko pustimo doma (pravzaprav bi lahko take padalce — ki imajo $b_i = 0$ in $a_i < 0$ — ignorirali že ob branju vhodnih podatkov). V nadaljevanju torej predpostavimo, da imajo vsi potniki $b_i \geq 1$; število potnikov, recimo mu I, je zato $\leq B$. — Če je torej zdaj $Z > B$, uredimo v mislih začetnike z avtomobili po naraščajočih vrednostih a_i. (1) Če je med njimi morda vsaj $Z - B$ takih, ki imajo $a_i = 0$, jih lahko odslovimo, pri čemer se število potnikov, ki jih lahko prepeljemo, nič ne zmanjša, vrednost Z pa se bo s tem izenačila z B in razpored je veljaven. (2) Drugače pa odslovimo vse začetnike z $a_i = 0$; število potnikov, ki jih lahko prepeljemo, se pri tem ne spremeni; Z se zmanjša za število teh odslovljenih začetnikov, vendar pa še vedno ostane večji od B. Zdaj imamo torej Z začetnikov z $a_i \geq 1$; katerihkoli B od njih bo zmoglo prepeljati vsaj B potnikov, mi pa imamo le $I \leq B$ potnikov, tako da je pravzaprav čisto vseeno, katerih $Z - B$ začetnikov pustimo doma.

Razmislek v prejšnjem odstavku nam je pokazal, da če se naš prej omenjeni postopek konča z $Z > B$, bomo lahko na izlet vzeli največ B začetnikov (tako, da bodo vsi lahko skočili v tandemu s kakšnim izkušenim padalcem); če pa se postopek konča z $Z \leq B$, bomo seveda lahko vzeli vseh tistih Z začetnikov. Tako je torej rezultat, po katerem nas sprašuje naloga, enak $\min\{Z, B\}$ (za vrednosti Z in B ob koncu postopka).

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
```

```
int main()
{
  typedef long long int llint;
  llint prostorZaPotnike = 0; // za toliko potnikov je prostora v avtomobilih
  llint stZac = 0; // toliko začetnikov gre na izlet
  llint maxZac = 0; // s toliko začetniki lahko skočijo tisti izkušeni, ki gredo na izlet
  vector<int> izkuseniBrezAvta; // vrednosti b_i izkušenih padalcev brez avtomobila
  llint zacetnikiBrezAvta = 0; // število začetnikov brez avtomobila

  // Preberimo podatke o padalcih. Vse padalce, ki imajo avto, takoj vzemimo na izlet.
  int n; cin >> n;
  for (int i = 0; i < n; ++i)
  {
    int a, b; cin >> a >> b;

    // Če ima avto, ga vzemimo na izlet.
    if (a >= 0) { prostorZaPotnike += a; if (b < 0) ++stZac; else maxZac += b; }

    // Sicer gre na seznam padalcev brez avta.
    else if (b > 0) izkuseniBrezAvta.push_back(b);
    else if (b < 0) ++zacetnikiBrezAvta;
  }

  // Uredimo izkušene padalce brez avta po b_i.
  sort(izkuseniBrezAvta.begin(), izkuseniBrezAvta.end());

  // Izberimo potnike.
  while (prostorZaPotnike > 0)

    if (stZac >= maxZac) {
      // Trenutno je več začetnikov, kot jih lahko skoči z dosedanjimi
      // izkušenimi, zato pošljimo še enega izkušenega.
      if (izkuseniBrezAvta.empty()) break; // Če izkušenih ni več, končajmo.
      --prostorZaPotnike; maxZac += izkuseniBrezAvta.back();
      izkuseniBrezAvta.pop_back(); }

    else {
      // Trenutno je manj začetnikov, kot jih lahko skoči z dosedanjimi
      // izkušenimi, zato pošljimo še kakšnega začetnika.
      llint d = min(zacetnikiBrezAvta, maxZac - stZac);
      if (d <= 0) break; // Če začetnikov ni več, končajmo.
      stZac += d; zacetnikiBrezAvta -= d; prostorZaPotnike -= d; }

  // Izpišimo rezultat.
  cout << min(maxZac, stZac) << endl; return 0;
}
```

Ta rešitev ima časovno zahtevnost $O(n \log n)$, ker mora urediti izkušene padalce brez avtomobila; za potrebe našega tekmovanja je to čisto dovolj dobro, vseeno pa razmislimo še o izboljšavi, ki se urejanju izogne.

Označimo z Z število začetnikov z avtom, z Z' število začetnikov brez avta, z I' število izkušenih brez avta, s P vsoto a_i po vseh padalcih z avtom in z B vsoto b_i po vseh izkušenih padalcih z avtom. Naj bo še B_k vsota vrednosti b_i po tistih k izkušenih padalcih brez avta, ki imajo največje b_i. Če se odločimo izbrati k izkušenih potnikov (za $0 \le k \le \min\{P, I'\}$), bomo lahko izbrali potem še $z(k) := \min\{P - k, Z'\}$ neizkušenih potnikov, končni rezultat pa bo $f(k) := \min\{Z + z(k), B + B_k\}$. Vrednost k bi radi seveda izbrali tako, da bo $f(k)$ čim večja. Ker je $Z + z(k)$ padajoča funkcija k-ja, $B + B_k$ pa naraščajoča, je pri majhnih k vrednost $f(k)$ enaka $B + B_k$ in je zato f tam tudi naraščajoča, pri velikih k pa je $f(k)$ enaka $Z+z(k)$ in je zato tam tudi ona padajoča. Svoj maksimum zato doseže $f(k)$ tam, kjer preide iz rastočega režima v padajočega, se pravi tam, kjer se funkciji $Z+z(k)$ in $B+B_k$ sekata. Dovolj je torej, če poiščemo največji k, pri katerem je $B + B_k \le Z + z(k)$, in najmanjši k, pri katerem je $B + B_k \ge Z + z(k)$.

To lahko naredimo z bisekcijo po k; pomembna podrobnost pa je naslednja: ko smo pri bisekciji omejeni na neki interval možnih k-jev, recimo od k_1 do k_2, vzdržujmo pri tem vsoto B_{k_1} ter seznam b_i-jev za $k_1 \le i \le k_2$ — natančneje rečeno, to so tisti izkušeni padalci brez avta, ki bi bili na indeksih od k_1 do k_2, če bi b_i-je vseh takih padalcev uredili padajoče. Mi pa ne bomo imeli urejenega celega seznama, pač pa le elemente,

ki bi bili v takem seznamu na indeksih od k_1 do k_2, pri čemer jih ne bomo imeli nujno urejenih v kakšnem posebnem vrstnem redu.

Ko moramo nato pri trenutnem koraku bisekcije preizkusiti naslednjega kandidata za k, to je $k = \lfloor (k_1 + k_2)/2 \rfloor$, lahko poiščemo ustrezni b_i s quickselectom po našem seznamu v $O(k_2 - k_1)$ časa; pri tem tudi razdelimo seznam na levo in desno polovico, od katerih nam bo ena prišla prav v naslednjem koraku bisekcije; in ko vrednosti B_{k_1} prištejemo vse elemente leve polovice, dobimo ravno B_k, ki ga potrebujemo v trenutnem koraku bisekcije. Ker se širina opazovanega intervala, $k_2 - k_1$, na vsakem koraku bisekcije razpolovi, je vsota teh širin po vseh korakih le $O(n)$ in zato je tudi časovna zahtevnost te rešitve le $O(n)$.

2. Ulične luči

Ko se a povečuje, se tudi interval, ki ga osvetljuje posamezna luč, le povečuje; če je ulica v celoti osvetljena pri nekem a, je tudi pri vsakem večjem a; in če pri nekem a ni v celoti osvetljena, potem ni v celoti osvetljena tudi pri nobenem manjšem. To pomeni, da lahko najmanjši primerni a poiščemo z bisekcijo; začnemo lahko na primer s tem, da vemo, da je $a = 0$ premajhen, $a = m$ pa je gotovo dovolj velik.

Ko moramo pri posameznem a preveriti, ali bi bila ulica pri tem a v celoti osvetljena, lahko razmišljamo takole. Za začetek je koristno luči urediti naraščajoče po p_i; odslej bomo torej predpostavili, da je $0 \le p_1 < p_2 < \ldots < p_n \le m$. Luč i osvetljuje interval od $\ell_i(a) := p_i - ac_i$ do $d_i(a) := p_i + ac_i$; najbolj desna točka, ki jo doseže svetloba luči od 1 do i, je potem $D_i(a) := \max_{1 \le j \le i} d_j(a)$; najbolj leva točka, ki jo doseže svetloba luči od i do n, pa je $L_i(a) := \min_{i \le j \le n} \ell_j(a)$. Naša ulica je interval $[0, m]$, ki si ga lahko predstavljamo razbitega na krajše podintervale: $[0, p_1], [p_1, p_2], \ldots, [p_{n-1}, p_n], [p_n, m]$. Prvi od teh je v celoti osvetljen, če je $L_1(a) \le 0$; zadnji je v celoti osvetljen, če je $D_n(a) \ge m$; vmes pa velja, da je interval $[p_{i-1}, p_i]$ v celoti osvetljen, če je $D_{i-1}(a) \ge L_i(a)$. Levo od tega intervala so namreč luči $1, \ldots, i-1$ in svetloba, ki prihaja desno od njih, se mora srečati s svetlobo, ki prihaja levo od luči $i, \ldots n$, ki ležijo desno od našega intervala.

Vrednosti $D_i(a)$ lahko računamo v zanki po naraščajočih in jih shranjujemo v neko tabelo: $D_0(a) = 0$, $D_i(a) = \max\{D_{i-1}(a), d_i(a)\}$. Podobno lahko potem računamo tudi vrednosti $L_i(a)$ po padajočih i, pri vsakem i pa preberemo še prej shranjeno vrednost $D_{i-1}(a)$ in preverimo pogoj $D_{i-1}(a) \ge L_i(a)$.

```
#include <vector>
#include <algorithm>
#include <cstdio>
using namespace std;

int main()
{
  // Preberimo vhodne podatke.
  int n, m; scanf("%d %d", &n, &m);
  struct Luc { int p, c; };
  vector<Luc> luci(n);
  for (auto &L : luci) scanf("%d %d", &L.p, &L.c);

  // Uredimo luči od leve proti desni.
  sort(luci.begin(), luci.end(), [] (const auto x, const auto y) { return x.p < y.p; });

  // Najmanjši a poiščimo z bisekcijo.
  int a1 = 0, a2 = m; vector<long long int> desno(n + 1);
  while (a2 - a1 > 1)
  {
    // a1 je premajhen, a2 je dovolj velik; poskusimo na sredi med njima.
    long long int a = (a1 + a2) / 2;

    // desno[i] = desni rob svetlobe prvih i luči
    desno[0] = 0;
    for (int i = 0; i < n; ++i) desno[i + 1] = max(desno[i], luci[i].p + a * luci[i].c);

    // Če svetloba ne doseže desnega krajišča cele ulice, je a premajhen.
    bool ok = (desno[n] >= m);

    // Poglejmo, kako se širi svetloba proti levi.
```

```
    long long int levo = m;
    for (int i = n - 1; i >= 0 && ok; --i) {
      levo = min(levo, luci[i].p - a * luci[i].c);
      // Zdaj je „levo" = levi rob svetlobe luči od i do n - 1.
      // Interval od luči i - 1 do i je pokrit, če se ta svetloba sreča s tisto,
      // ki prihaja desno od luči 0, ... , i - 1. (Pri i = 0 ta pogoj preveri,
      // če svetloba z desne doseže tudi levo krajišče cele ulice.)
      ok = (desno[i] >= levo); }
    // Pripravimo se na naslednjo iteracijo bisekcije.
    if (ok) a2 = a; else a1 = a;
  }
  printf("%d\n", a2); return 0; // Izpišimo rezultat.
}
```

Pri vsakem koraku bisekcije imamo $O(n)$ dela s preverjanjem, ali je ulica v celoti osvetljena, zato ima ta rešitev časovno zahtevnost $O(n \log m)$.

Osvetljenost bi lahko preverili tudi tako, da bi intervale $[p_i - a \cdot c_i, p_i + a \cdot c_i]$ zložili v seznam in jih uredili v začetnem krajišču; potem ni težko preveriti, ali je osvetljena celotna ulica:

```
vector<pair<int, int>> intervali;
⋮
sort(intervali.begin(), intervali.end());
int svetloDo = 0;
for (auto [L, D]: intervali)
  // Dosedanji intevali osvetljujejo ulico do x-koordinate svetloDo.
  // Če se trenutni interval začne desno od tam, je vmes neosvetljeno območje.
  if (L > svetloDo) break;
  // Sicer nam trenutni interval morda osvetli ulico še dlje v desno.
  else svetloDo = max(svetloDo, D);
// Preverimo, ali je ulica osvetljena do konca.
if (svetloDo < m) a1 = a; else a2 = a;
```

Ta rešitev je sicer slabša od prejšnje, kajti zaradi urejanja seznama intervalov nam vsak korak bisekcije vzame $O(n \log n)$ časa, celotna rešitev pa zato $O(n(\log n)(\log m))$ časa. Pri največjih testnih primerih bo ta rešitev prekoračila časovno omejitev, razen če smo zelo pazljivi pri implementaciji in vključimo v rešitev kakšno hevristiko, ki prihrani nekaj časa, na primer eno od naslednjih dveh:

- za začetno zgornjo mejo bisekcije je koristno namesto $a = m$ vzeti maksimum vrednosti p_1/c_1, $(m - p_n)/c_m$ in $\max_i(p_i - p_{i-1})/(c_{i-1} + c_i)$, kajti to je a, pri katerem bo vsak interval ulice v celoti osvetljen že zgolj zaradi luči, ki stojita na njegovih krajiščih;
- za interval, ki ga osvetljuje luč i, je koristno vzeti $[\max\{0, p_i - ac_i\}, \min\{m, p_i + ac_i\}]$, torej odrezati tisto, kar bi sicer segalo čez rob ulice. Če se nam potem pri več lučeh pojavijo intervali z levim krajiščem 0, obdržimo od njih le tistega, ki sega najdlje na desno; podobno pa tudi med intervali z desnim krajiščem m obdržimo le tistega, ki sega najdlje v levo. Tako lahko zmanjšamo število intervalov, ki jih bo treba urejati.

Od druge hevristike je korist predvsem pri velikih a, kjer bi drugače veliko intervalov štrlelo čez rob ulice; prva hevristika pa se izogne velikim a tako, da že na začetku zmanjša zgornjo mejo bisekcije. Zato ni prave koristi od tega, da bi v rešitvi uporabili obe hevristiki hkrati (čeprav ni narobe, če to naredimo).

3. Špijonaža

Hierarhijo vohunov si lahko predstavljamo kot drevo s korenom 1; *poddrevo* vohuna u tvorijo on in vsi njegovi posredno ali neposredno podrejeni. Označimo s $f(u)$ najmanjše število izvodov, ki morajo hkrati obstajati v u-jevem poddrevesu od trenutka,

ko u prejme svoj izvod, do trenutka, ko so vsi vohuni v u-jevem poddrevesu že videli dokument. Naloga torej sprašuje po $f(1)$. Funkcijo f lahko računamo z rekurzivnim razmislekom. Če u sploh nima neposredno podrejenih (torej če je list drevesa), je $f(u) = 1$, saj lahko u uniči svoj izvod takoj po tem, ko ga je prejel.

Če pa u ima neposredno podrejene (torej če je u notranje vozlišče), razmišljajmo takole: ko izroči u nekemu svojemu neposredno podrejenemu, recimo v-ju, kopijo dokumenta, bo sčasoma moralo biti v v-jevem poddrevesu hkrati prisotnih $f(v)$ izvodov dokumenta, preden bodo vsi vohuni v tem poddrevesu videli dokument; s tem pa bodo ti izvodi seveda prisotni tudi v u-jevem poddrevesu. Medtem bo v u-jevem poddrevesu prisoten vsaj še izvod, ki ga bo imel u sam, razen če je bil v zadnji u-jev neposredno podrejeni, ki je prejel svojo kopijo dokumenta; v tem slednjem primeru lahko u uniči svoj izvod, čim v prejme kopijo.

Tako torej vidimo, da za vsakega u-ju neposredno podrejenega v razen za zadnjega, ki prejme svoj izvod, velja $f(u) \geq 1 + f(v)$. Za zadnjega neposredno podrejenega, ki prejme svoj izvod — recimo mu z_u — pa velja $f(u) \geq f(z_u)$, vendar tudi $f(u) \geq 2$, kajti v trenutku, ko je z_u že dobil svoj izvod dokumenta, u pa svojega še ni uničil, sta obstajala vsaj tadva izvoda hkrati (ta robni primer je pomemben le, če ima u enega samega neposredno podrejenega, ta pa je list in ima $f(z_u) = 1$; takrat moramo paziti, da za $f(u)$ dobimo 2 in ne 1).

Najmanjša možna vrednost za $f(u)$ je torej $\max\{2, f(z_u), 1 + \max_v f(v)\}$. Pri tem gre $\max_v$ po vseh u-jevih neposredno podrejenih razen po z_u. Slednji je torej edini, pri katerem se lahko izognemo potrebi po tem, da bi njegovi $f(\cdot)$ prišteli 1; zato je smiselno za z vzeti tistega neposredno podrejenega, ki ima največjo vrednost $f(z)$.

Zdaj imamo vse, kar potrebujemo za rekurzivno računanje funkcije f. Ko smo pri vohunu u, najprej z vgnezdenimi rekurzivnimi klici obdelamo vse njegove neposredno podrejene; zapomnimo si, pri katerem je nastopila največja vrednost $f(\cdot)$ — to bo naš z_u; poleg tega pa si zapomnimo še največjo vrednost $f(\cdot)$ po ostalih neposredno podrejenih — to bo naš $\max_v f(v)$. Ko imamo to dvoje, lahko izračunamo $f(u)$ po prej omenjeni formuli, poleg tega pa si zapomnimo tudi z_u, ki bo prišel prav pri izpisu zaporedja korakov.

Naloga namreč zahteva, da izpišemo ne le $f(1)$, pač pa tudi zaporedje korakov, s katerim lahko vsi vohuni vidijo dokument, ne da bi kdaj obstajalo več kot $f(1)$ izvodov naenkrat. Tudi za ta izpis lahko poskrbimo z rekurzijo. Ko smo pri vohunu u, najprej obdelamo vse njegove neposredno podrejene razen z_u; vsak neposredno podrejeni dobi svoj izvod dokumenta, nato pa z rekurzivnim klicem obdelamo njegovo poddrevo (v okviru tega vsi vohuni v tem poddrevesu svoje izvode prej ali slej tudi uničijo). Nazadnje dobi svoj izvod še z_u, potem u svoj izvod uniči in z rekurzivnim klicem obdelamo še z_u-jevo poddrevo. (Pazimo seveda tudi na robni primer: če u nima podrejenih, vohun z_u sploh ne obstaja.)

```
#include <cstdio>
#include <vector>
#include <algorithm>
using namespace std;

struct Vohun
{
  vector<int> podrejeni; // seznam neposredno podrejenih
  int f = −1; // najboljši rezultat za poddrevo, ki ga tvorijo u in njegovi podrejeni
  int z = −1; // neposredno podrejeni z največjim f
};
vector<Vohun> vohuni;

// Izračuna f po celem u-jevem poddrevesu in vrne f(u).
int Izracunaj(int u)
{
  // Rekurzivno obdelajmo vse u-jeve neposredno podrejene
  // in si zapomnimo dva z največjim f.
  auto &U = vohuni[u]; int f1 = −1, f2 = −1;
  for (int v : U.podrejeni)
    if (int fv = Izracunaj(v); fv > f1) f2 = f1, f1 = fv, U.z = v;
```

```
    else if (fv > f2) f2 = fv;
  // Izračunajmo rezultat tudi za u.
  return U.f = (U.z < 0) ? 1 : max(max(f1, 2), 1 + f2);
}

// Izpiše vse korake v u-jevem poddrevesu od trenutka, ko u prejme svoj izvod.
void Izpisi(int u)
{
  // Vohun u je prejel svoj izvod dokumenta. Najprej rekurzivno obdelajmo
  // vse podrejene razen z.
  auto &U = vohuni[u];
  for (int v : U.podrejeni) if (v != U.z) {
    printf("1 %d\n", v + 1); // Podrejeni v prejme svoj izvod.
    Izpisi(v); } // Rekurzivno obdelajmo v-jevo poddrevo.
  // Končno tudi podrejeni z prejme svoj izvod.
  if (U.z >= 0) printf("1 %d\n", U.z + 1);
  // Zdaj lahko u svoj izvod uniči.
  printf("2 %d\n", u + 1);
  // Rekurzivno obdelajmo poddrevo vohuna z.
  if (U.z >= 0) Izpisi(U.z);
}

int main()
{
  // Preberimo vhodne podatke.
  int n; scanf("%d", &n); vohuni.resize(n);
  for (int u = 1; u < n; ++u) {
    int p; scanf("%d", &p);
    vohuni[--p].podrejeni.emplace_back(u); }
  // Izračunajmo rezultate in jih izpišimo.
  printf("%d %d\n", 2 * n - 1, Izracunaj(0));
  Izpisi(0); return 0;
}
```

4. Valj

Nalogo lahko rešujemo z iskanjem v širino. Začnimo v poljubnem polju (x_0, y_0) in pregledujmo mrežo tako, da se vedno premaknemo iz trenutnega polja na sosednja polja le, če so iste barve. Tako bomo sčasoma pregledali celotno povezano komponento (strnjeno zaplato polj iste barve), ki ji pripada (x_0, y_0). Pri tem veljata dve polji za sosednji, če imata skupno stranico, poleg tega pa sta za si v vsaki vrstici sosednji tudi skrajno levo in skrajno desno polje — s tem upoštevamo dejstvo, da naša mreža ne leži na ravnini, ampak tvori plašč valja.

Definicija obhoda valja v besedilu naloge je koristen namig: za obhod desno mora biti vsaka navpičnica prečkana enkrat več v desno kot v levo. Število teh prečkanj pri meji med stolpcema x in $x+1$ označimo z d_x oz. ℓ_x. Pri obhodu v desno mora torej za vsak x veljati $d_x - \ell_x = 1$; in če to seštejemo po vseh m navpičnicah, mora za obhod kot celoto veljati $d - \ell = m$, kjer je d skupno število korakov v desno, ℓ pa skupno število korakov v levo. Podobno mora pri obhodu v levo veljati $d - \ell = -m$.

Zato je ob pregledovanju mreže z iskanjem v širino koristno, če si za vsako polje zapomnimo tudi razliko med številom korakov v desno in korakov v levo na naši poti od (x_0, y_0) do tistega polja; recimo temu $d(x, y)$. Če potem med pregledovanjem sosedov nekega polja (x', y') vidimo, da smo nekega takega soseda (x, y) obiskali že prej in da se je takratni $d(x, y)$ razlikoval ravno za $\pm m$ od vrednosti, ki bi jo dobil, če bi na tisto polje (ponovno) stopili zdaj — recimo tej vrednosti $d'(x, y)$ —, potem vemo, da imamo obhod okrog valja: sestavlja ga najprej pot, po kateri smo prišli od (x_0, y_0) do (x', y'), nato korak od tam v njegovega soseda (x, y), nato pa pot od tam do (x_0, y_0), ravno obratna od poti, po kateri smo prvič dosegli polje (x, y). Skupaj je namreč razlika med številom desnih in levih korakov na tem obhodu enaka $d'(x, y) - d(x, y)$, to pa je ravno

$\pm m$.[1]

Ko na ta način pregledamo celotno povezano komponento, dosegljivo iz (x_0, y_0), jo v mislih pobrišemo in nadaljujemo pri kakšnem drugem še nepregledanem polju, dokler ni pregledana celotna mreža. V spodnji rešitvi označujemo polja, ki smo jih že odkrili (in dodali v vrsto pri iskanju v širino), tako, da postavimo njihovo barvo na -1. Za polja, ki ne pripadajo trenutni komponenti, imamo v $d(x, y)$ vrednost $n(m + 1)$, ki je gotovo za več kot m različna od katerekoli vrednosti, ki jo lahko $d(x, y)$ dobi pri poljih, ki pripadajo trenutni komponenti.

```
#include <cstdio>
#include <vector>
#include <algorithm>
using namespace std;

int main()
{
  // Preberimo vhodne podatke.
  int w, h; scanf("%d %d", &h, &w);
  vector<int> A(w * h); for (auto &a : A) scanf("%d", &a);

  vector<int> rezultati; // Seznam barv, po katerih je obhod mogoč.
  vector<int> D(w * h, w * (h + 1)); // Zamik (v smeri x) glede na začetno polje.
  vector<int> vrsta; // Vrsta za iskanje v širino.
  const int DX[] = { 1, -1, 0, 0 }, DY[] = { 0, 0, 1, -1 };

  for (int y0 = 0; y0 < h; ++y0) for (int x0 = 0; x0 < w; ++x0)
  {
    int barva = A[y0 * w + x0]; if (barva < 0) continue;

    // Pojdimo iz polja (x0, y0) po poljih iste barve.
    D[y0 * w + x0] = 0; A[y0 * w + x0] = -1;
    vrsta.clear(); int glava = 0; vrsta.emplace_back(y0 * w + x0);
    bool obstajaObhod = false;
    while (glava < vrsta.size())
    {
      int u = vrsta[glava++]; int ux = u % w, uy = u / w;
      // Dosegli smo polje (ux, uy). Preglejmo njegove sosede.
      for (int smer = 0; smer < 4; ++smer)
      {
        int vx = (ux + DX[smer] + w) % w, vy = uy + DY[smer];
        if (vy < 0 || vy >= h) continue;
        int v = vx + w * vy;

        // Če smo tega soseda dosegli že prej pri trenutnem (x0, y0) in je
        // razlika med takratnim in sedanjim zamikom ravno ± w, potem obstaja obhod.
        if (abs(D[u] + DX[smer] - D[v]) == w) obstajaObhod = true;
        if (A[v] != barva) continue;

        // Če je sosed enake barve in ga prej še nismo dosegli, ga dodajmo v vrsto.
        A[v] = -1; D[v] = D[u] + DX[smer];
        vrsta.emplace_back(v);
      }
    }

    if (obstajaObhod) rezultati.emplace_back(barva);

    // Počistimo za sabo v tabeli D.
    for (int u : vrsta) D[u] = w * (h + 1);
  }
```

[1] Tu se morda utegne pojaviti pomislek, ali bi se lahko zgodilo, da bi obstajal le obhod, ki gre večkrat okrog valja, ne pa tudi obhod, ki gre okrog natanko enkrat; tedaj bi morali preverjati, ali je razlika $d'(x, y) - d(x, y)$ večkratnik m, ne pa le $\pm m$. Pa recimo, da se to res zgodi. Poskusimo mrežo poplaviti (*flood fill*) od spodnjega roba proti zgornjemu, pri čemer se lahko poplava širi s trenutnega polja na vse njegove sosede, ki imajo z njim skupno vsaj eno oglišče, ne sme pa se širiti na polja, ki pripadajo našemu obhodu. Ker gre obhod vsenaokrog valja, bo naši poplavi preprečil, da bi kjerkoli dosegla zgornji rob mreže. Poplavljeno območje torej na vrhu vsepovsod meji na celice, ki pripadajo našemu obhodu, in iz teh mejnih celic lahko zdaj sestavimo krajši obhod, ki gre okrog valja le enkrat. Torej se res ne more zgoditi, da bi obstajal le obhod, ki gre okrog valja večkrat, ne pa tudi tak, ki gre okrog njega le enkrat.

```
  // Izpišimo rezultate; pazimo, da izpišemo vsako barvo največ enkrat.
  sort(rezultati.begin(), rezultati.end());
  int prej = −1; for (int barva : rezultati) if (barva != prej) {
    printf("%s%d", prej >= 0 ? "  " : "", barva); prej = barva; }
  printf("%s\n", rezultati.empty() ? "mavrica" : ""); return 0;
}
```

Časovna zahtevnost te rešitve je $O(nm)$ za samo iskanje v širino in $O(n \log n)$ za urejanje seznama rezultatov. Vsak obhod je namreč sestavljen iz vsaj m polj, zato lahko obstajajo obhodi za največ $(nm)/m = n$ različnih barv (npr. če je vsaka vrstica v celoti svoje barve).

5. Urejanje z medianami

V nalogi so podstavki oštevilčeni od 1 do n, vendar je za potrebe pisanja rešitve v C++ bolj prikladno, če jih oštevilčimo od 0 do $n - 1$ (pri sporazumevanju z ocenjevalnim sistemom pa tem številkam prištejemo 1). Poleg tega bomo oštevilčili tudi zaboje: vsak zaboj naj dobi številko tistega podstavka, na katerem je stal na začetku urejanja, preden smo zaboje začeli premikati. Operacijam prvega tipa bomo rekli na kratko „zamenjave“, operacijam drugega pa „izračuni median“ ali krajše kar „mediane“, če ne bo tveganja za zmedo.

Znanih je veliko algoritmov za urejanje, ki večinoma temeljijo na tem, da je mogoče dva elementa primerjati in povedati, kateri mora v urejenem vrstnem redu priti pred drugim. Pri naši nalogi ne moremo primerjati dveh zabojev, pač pa lahko z izračunom mediane primerjamo tri zaboje; izziv je torej predvsem v tem, kako kakšnega od znanih algoritmov za urejanje prilagoditi tako, da bo namesto primerjave uporabljal izračune median.

Uporabimo lahko na primer iskanje z vstavljanjem (*insertion sort*): tu postopoma pripravljamo urejen seznam zabojev, pri čemer začnemo s praznim seznamom, nato pa na vsakem koraku vzamemo neki zaboj ter ga vstavimo v naš seznam na tako mesto, da seznam ostane urejen. Toda če bomo pri tem vrivanju zaboje dejansko premikali po skladišču, bomo izvedli $O(n^2)$ zamenjav, kar je že preveč (spomnimo se, da gre n do 1000, omejeni pa smo na 40 000 operacij). Namesto tega si bomo urejeni seznam zabojev pripravljali v pomnilniku, ko pa bo nared, bomo zaboje zares prerazporedili z največ $O(n)$ zamenjavami (za to v spodnji rešitvi poskrbi podprogram ZakljuciUrejanje).

Razmisliti moramo še o tem, kako ugotoviti, kam v urejeni seznam je treba vriniti novi zaboj, da bo ostal urejen. Če bi imeli možnost primerjati po dva zaboja, bi lahko uporabili bisekcijo; ker pa imamo na voljo le izračun mediane, ki nekako primerja tri zaboje, lahko namesto bisekcije uporabimo „trisekcijo“: naj bo m_1 zaboj približno na eni tretjini našega seznama, m_2 zaboj približno na dveh tretjinah seznama in x naš novi zaboj; tedaj vemo, da če je mediana zabojev m_1, m_2 in x ravno zaboj x, to pomeni, da je x po teži med m_1 in m_2, torej ga moramo vriniti v srednjo tretjino seznama; če je mediana omenjenih treh zaboj m_1, to pomeni, da m_1 leži po teži med x in m_2, torej moramo x vriniti v prvo tretjino seznama, levo od m_1; in podobno, če je mediana zaboj m_2, leži m_2 po teži med m_1 in x, torej moramo x vrniti v zadnjo tretjino seznama, desno od m_2. Tako smo z enim izračunom mediane skrajšali območje, ki še pride v poštev za vrivanje zaboja x, na tretjino prvotne dolžine. V naslednjem koraku trisekcije razdelimo to tretjino spet na tretjine in tako naprej.

Če smo imeli seznam k zabojev, bomo tako v približno $\log_3 k$ korakih našli pravi položaj, kamor je treba vriniti novi zaboj. Tako imamo vsega skupaj približno $\sum_{k=3}^{n} \log_3 k \approx n \log_3 n - O(n)$ računanj mediane. S tem smo že zelo blizu teoretični spodnji meji: ker moramo znati prerazporediti zaboje na $n!$ možnih načinov in ker se lahko obnašanje naše rešitve po vsakem izračunu mediane razveji v tri možna nadaljevanja, moramo gotovo izvesti vsaj $\log_3 n!$ izračunov mediane, kar je tudi približno $n \log_3 n - O(n)$.

Zapišimo dosedanjo rešitev v C++:

```
#include <cstdio>
#include <vector>
#include <algorithm>
#include <utility>
```

```
#include <random>
using namespace std;

// Definirajmo tipa Zaboj in Podstavek, da se bo v izvorni kodi
// bolje videlo, katere vrednosti so številke zabojev in katere podstavkov.
typedef int Zaboj, Podstavek;

int n; // število podstavkov in tudi število zabojev
vector<Zaboj> zaboji; // zaboji[p] = zaboj na podstavku p
vector<Podstavek> kjeJe; // kjeJe[z] = podstavek, kjer stoji zaboj z

// Zamenja zaboja na podstavkih p1 in p2.
void ZamenjajP(Podstavek p1, Podstavek p2) {
  printf("%d %d -1\n", p1 + 1, p2 + 1); fflush(stdout);
  Zaboj z1 = zaboji[p1], z2 = zaboji[p2];
  swap(zaboji[p1], zaboji[p2]); swap(kjeJe[z1], kjeJe[z2]); }

// Zamenja zaboja s številkama z1 in z2.
void ZamenjajZ(Zaboj z1, Zaboj z2) { ZamenjajP(kjeJe[z1], kjeJe[z2]); }

// Vrne številko podstavka, na katerem je srednji zaboj po teži.
Podstavek MedianaP(Podstavek p1, Podstavek p2, Podstavek p3) {
  printf("%d %d %d\n", p1 + 1, p2 + 1, p3 + 1); fflush(stdout);
  int m; scanf("%d", &m); return m - 1; }

// Kot MedianaP, le da dela s številkami zabojev namesto podstavkov.
Zaboj MedianaZ(Zaboj z1, Zaboj z2, Zaboj z3) {
  return zaboji[MedianaP(kjeJe[z1], kjeJe[z2], kjeJe[z3])]; }

// Razporedi zaboje v vrstni red, kot ga določa „v“.
void ZakljuciUrejanje(const vector<Zaboj> &v) {
  for (Podstavek p = 0; p < n; ++p)
    if (int q = kjeJe[v[p]]; q != p) ZamenjajP(p, q);
  printf("-1\n"); fflush(stdout); }

// Ugotovi, na kateri indeks v „v“ je treba vriniti novi zaboj,
// da bo zaporedje ostalo urejeno po teži.
int Trisekcija(const vector<Zaboj> &v, Zaboj novi)
{
  int L = 0, D = v.size();
  while (D > L)
  {
    // Novi zaboj bo treba vriniti v „v“ na enega od indeksov L, ... , D.
    if (D - L == 1) { if (L > 0) --L; else ++D; }
    // Zdaj območje L, ... , D obsega vsaj tri indekse. Razdelimo ga na tretjine.
    int M1 = (2 * L + D) / 3, M2 = (L + 2 * D) / 3;
    // V katero tretjino sodi novi zaboj?
    Zaboj med = MedianaZ(v[M1], v[M2], novi);
    if (med == v[M1]) D = M1;
    else if (med == v[M2]) L = M2 + 1;
    else L = M1 + 1, D = M2;
  }
  return L;
}

void UrediZVstavljanjem(vector<Zaboj> &v)
{
  // Zaboje bomo pregledovali v naključnem vrstnem redu.
  vector<int> vrstniRed(n); mt19937_64 r;
  for (int i = 0; i < n; ++i) {
    int j = uniform_int_distribution<int>(0, i)(r);
    vrstniRed[i] = vrstniRed[j]; vrstniRed[j] = i; }
  // V vektorju „v“ bo nastajal vrstni red, kjer bodo zaboji urejeni po teži.
  v.clear(); v.reserve(n);
  for (Zaboj z : vrstniRed)
```

```
    if (v.size() < 2) v.emplace_back(z);
    else v.insert(v.begin() + Trisekcija(v, z), z);
}

int main()
{
  scanf("%d", &n); zaboji.resize(n); kjeJe.resize(n);
  for (int i = 0; i < n; ++i) zaboji[i] = i, kjeJe[i] = i;
  vector<Zaboj> vrstniRed; UrediZVstavljanjem(vrstniRed);
  ZakljuciUrejanje(vrstniRed); return 0;
}
```

Slabost te rešitve je, da obraba ni dovolj enakomerno porazdeljena po podstavkih. Recimo, da smo v urejeno zaporedje že dodali k zabojev; ko dodajamo naslednji zaboj, nastopata v prvi iteraciji trisekcije zaboja na mestih (približno) $k/3$ in $2k/3$ v tem urejenem zaporedu. Zanju torej velja, da je približno tretjina izmed dosedanjih k zabojev lažjih oz. težjih od njiju; zato pa lahko pričakujemo, da bo tudi približno tretjina izmed vseh n zabojev lažjih oz. težjih od njiju, saj je bilo dosedanjih k zabojev naključno izbranih izmed vseh n zabojev in bi morale biti njihove teže podobno porazdeljene. V prvi iteraciji vsakega izvajanja trisekcije torej praviloma nastopata dva taka zaboja, ki bosta v končnem vrstnem redu vseh n zabojev v bližini mest $n/3$ in $2n/3$; podstavki, na katerih stojijo takšni zaboji, se zato zelo nadpovprečno obrabijo, saj zabojev medtem nič ne premikamo (to naredimo šele na koncu, v podprogramu ZakljuciUrejanje). Pri naših poskusih z $n = 1000$ je bila maksimalna obraba (po vseh podstavkih) v povprečju okrog 134 in na testnih primerih s tekmovanja bi ta rešitev dobila 86 točk od 100.

Skupno število računanj mediane je, kot smo videli, približno $n \log_3 n$ in pri vsakem se obrabijo trije podstavki; vseh podstavkov pa je n, torej bo povprečna obraba približno $3 \log_3 n$, kar pri $n = 1000$ nanese približno 18,9. Če torej hočemo, da bo maksimalna obraba ≤ 20 (kar naloga zahteva za vse točke), bomo morali poskrbeti, da bodo podstavki obrabljeni zelo enakomerno. (V resnici so sicer naše ocene tukaj nekoliko pesimistične; dejansko število računanj mediane je bilo pri $n = 1000$ pri naših poskusih v povprečju okrog 5579, kar pomeni povprečno obrabo 16,7.)

Pri računanju mediane nas v resnici zanima mediana treh *zabojev*; omejitev, ki nam jo postavlja naloga, pa se nanaša na to, pri koliko računanjih mediane sodeluje posamezni *podstavek*, ne pa zaboj. Če bi se torej slučajno izkazalo, da bi neki zaboj potrebovali pri veliko računanjih mediane, nas nič ne sili, da ga imamo ves čas na istem podstavku in tako povečujemo obrabo tega podstavka; zaboj lahko vsake toliko časa premaknemo na kak drug podstavek, ki je zaenkrat manj obrabljen. Tako lahko z zaboji „kolobarimo“ med podstavki in skrbimo, da so vsi podstavki čim bolj enakomerno obrabljeni (torej da vsi nastopijo v približno enako veliko računanjih mediane).

Dogovorimo se na primer, da bomo poskušali vzdrževati naslednjo lastnost: razlika med maksimalno in minimalno obrabo (po vseh odstavkih) sme biti največ R, razen če je maksimalna obraba $\leq B$. Pri tem sta B in R konstanti, ki si ju bomo izbrali pred začetkom urejanja. Če bi neki zaboj x, ki ga nameravamo uporabiti pri naslednjem izračunu mediane, to pravilo prekršil (ker bi se zaradi povečanja njegove obrabe povečala tudi maksimalna obraba in bila po novem večja od B ter tudi za več kot R večja od minimalne obrabe), ga bomo zamenjali z enim od zabojev, ki so trenutno na podstavkih z minimalno obrabo.[2]

Namen tega pravila je, da dokler je obraba pri vseh podstavkih nizka (do B), se nam ni treba truditi z uravnoteževanjem obrabe in prerazporejati zabojev po podstavkih; kasneje pa bo posamezni zaboj treba prerazporediti le na vsakih R računanj mediane, v katerih je udeležen. Najmanjšo maksimalno obrabo, vendar tudi največ zamenjav, bomo dobili pri $B = 0$, $R = 1$ (pri naših poskusih je bilo tu pribl. 2,6-krat toliko zamenjav kot računanj mediane); precej zamenjav pa lahko prihranimo, če postavimo B višje. Pri našem postopku s trisekcijo smo na primer videli, da bo povprečna obraba približno $3 \log_3 n$, maksimalna obraba po vseh podstavkih pa tudi ne more biti manjša

[2]Natančneje rečeno: med zaboji, ki ne nastopajo v naslednjem izračunu mediane, vzamemo tistega, ki trenutno stoji na najmanj obrabljenem podstavku, in ga zamenjamo z zabojem x. Lahko se tudi zgodi, da primernega kandidata za zamenjavo sploh ni (oz. so sami taki, katerih podstavki so enako obrabljeni kot x-ov, takrat pa od zamenjave ne bi bilo nobene koristi).

od povprečne. Postavimo torej B malo pod to mejo, na primer na $3(\lfloor \log_3 n \rfloor - 1)$, parameter R pa naj ostane 1: dobili bomo enako dobro maksimalno obrabo kot pri $B = 0$, $R = 1$, število zamenjav pa bo zdaj manjše kot število računanj mediane.

Oglejmo si implementacijo takšnega uravnoteževanja obrabe. Za vsako možno obrabo si načeloma želimo vzdrževati seznam vseh podstavkov s to obrabo; da bomo lahko podstavek poceni premaknili z enega seznama na naslednjega, ko se mu obraba poveča, bomo za sezname uporabili verige, povezane s kazalci (*linked lists*). Te sezname hranimo v vektorju postavkiPoObrabi, ki ga uporabljamo kot krožno tabelo: postavki z obrabo x so v postavkiPoObrabi[x % M], pri čemer mora biti M dovolj velik, da gotovo ne bomo hkrati potrebovali več kot M takih seznamov. Primerna vrednost je $M = \max\{B, R\} + 2$.

```
struct Uravnotezevalnik
{
  int B, R, M;

  // Obraba vsakega podstavka ter minimum in maksimum po vseh podstavkih.
  vector<int> obrabaPodstavka; int minObraba, maxObraba;

  // Podstavki z obrabo x tvorijo seznam postavkiPoObrabi[x % M].
  vector<list<int>> podstavkiPoObrabi;

  // kjePodstavek[p] kaže na element z vrednostjo p v seznamu
  // postavkiPoObrabi[obrabaPodstavka[x] % M].
  vector<list<int>::iterator> kjePodstavek;

  void Init(int n)
  {
    int log3_n = 0, N = n; while (N > 1) N /= 3, ++log3_n;
    B = 3 * max(log3_n - 1, 0); R = 1; M = max(B, R) + 2;

    // Na začetku je obraba vseh podstavkov 0.
    minObraba = 0; maxObraba = 0; obrabaPodstavka.clear(); obrabaPodstavka.resize(n, 0);

    // Dodajmo vse podstavke v seznam podstavkiPoObrabi[0].
    podstavkiPoObrabi.clear(); podstavkiPoObrabi.resize(M);
    kjePodstavek.clear(); kjePodstavek.resize(n);
    auto &L = podstavkiPoObrabi[0];
    for (Podstavek p = 0; p < n; ++p) kjePodstavek[p] = L.emplace(L.begin(), p);
  }

  // Po potrebi skuša premakniti zaboj z na kak drug podstavek, da se
  // maksimalna obraba ne bi povečala. Pri tem ne uporabi podstavkov,
  // na katerih sta trenutno zaboja z1 in z2 (ker bosta tadva zaboja tudi
  // sodelovala v naslednjem izračunu mediane in zato od takšne zamenjave
  // ne bi bilo nobene koristi).
  void Uravnotezi(Zaboj z, Zaboj z1 = -1, Zaboj z2 = -1)
  {
    Podstavek p = kjeJe[z];
    if (obrabaPodstavka[p] == maxObraba && maxObraba >= B &&
                                                  maxObraba - minObraba >= R)
      // Premaknimo ta zaboj na kak manj obrabljen podstavek. Vzeli bomo
      // najmanj obrabljen tak podstavek, ki ne vsebuje zabojev z1 ali z2.
      for (int o = minObraba; o < maxObraba; ++o)
        for (Podstavek q : podstavkiPoObrabi[o % M]) {
          if (Zaboj z = zaboji[q]; z == z1 || z == z2) continue;
          ZamenjajP(p, q); p = q;
          o = maxObraba; break; }

    // Zapomnimo si, da bo obraba podstavka p zdaj narasla.
    int &obraba = obrabaPodstavka[p];
    auto &prej = podstavkiPoObrabi[obraba % M],
         &potem = podstavkiPoObrabi[(obraba + 1) % M];
    potem.splice(potem.begin(), prej, kjePodstavek[p]); // Preselimo ga v naslednji seznam.
    if (prej.empty() && obraba == minObraba) ++minObraba;
    if (++obraba > maxObraba) maxObraba = obraba;
  }
} U;
```

V funkcijo main dodajmo klic U.Init(n), lahko takoj po tistem, ko preberemo n. Namesto dosedanje funkcije MedianaZ pa bomo morali (v funkciji Trisekcija) za izračun mediane treh zabojev uporabljati naslednjo:

```
// Kot MedianaZ, vendar z uravnoteževanjem obrabe.
Zaboj MedianaZU(Zaboj z1, Zaboj z2, Zaboj z3) {
  U.Uravnotezi(z1, z2, z3); U.Uravnotezi(z2, z1, z3); U.Uravnotezi(z3, z1, z2);
  return MedianaZ(z1, z2, z3); }
```

Na testnih primerih s tekmovanja dobi ta rešitev vse točke; pri $n = 1000$ je maksimalna obraba 17.

Nalogo lahko rešimo še na veliko drugih načinov; na kratko si jih oglejmo nekaj. V oglatih oklepajih je pri posameznih rešitvah napisano, koliko točk bi dobile na testnih primerih z našega tekmovanja.

• Naša dosedanja rešitev porabi načeloma $O(n^2)$ časa za vrivanje zabojev v urejeni seznam. To nas sicer ni motilo, ker je n majhen in ker zaboje premikamo le v pomnilniku, ne delamo pa res zamenjav v skladišču. Če pa bi hoteli časovno zahtevnost vendarle zmanjšati, bi morali za predstavitev urejenega seznama namesto tabele oz. vektorja uporabiti kakšno primerno uravnoteženo drevesasto strukturo (npr. rdeče-črno drevo, B-drevo, treap). Tam bi vrivanje v drevo vzelo $O(\log n)$ časa, enako pa tudi dostop do elementa na določenem indeksu; ker potrebujemo tak dostop na vsakem koraku trisekcije, teh pa je skupno $O(n \log n)$; bi bila časovna zathevnost te rešitve $O(n(\log n)^2)$.

Še ena možnost pa je, da našo trisekcijo prilagodimo strukturi drevesa. Recimo, da imamo binarno drevo, v katerem ima koren otroka a in b; če bi radi v drevo vrinili nov zaboj x, izračunajmo mediano zabojev a, b in x. Če je mediana a, moramo x dodati v levo a-jevo poddrevo; če je mediana b, moramo x dodati v desno b-jevo poddrevo; sicer pa leži x med a in b, torej ga moramo dodati v desno a-jevo ali v levo b-jevo poddrevo (ta primer lahko obravnavamo tako, da se za hip pretvarjamo, kot da bi bili omenjeni poddrevesi pritrjeni neposredno na koren, ne pa šele en nivo nižje). Tako se lahko postopoma spuščamo po drevesu in v $O(\log n)$ korakih najdemo pravo mesto, kamor je treba vriniti novi element x; celotna rešitev ima zato časovno zahtevnost le $O(n \log n)$. Slabost te rešitve je, da je število iteracij trisekcije (in s tem število izračunov mediane) malo večje kot pri prvotni rešitvi, ker je globina drevesa sicer reda $O(\log n)$, vendar je večja od $\log_3 n$ (ker je binarno in ker je uravnoteženo le približno, ne pa popolnoma).

• Uporabimo lahko quicksort oz. hitro urejanje. Ta postopek običajno temelji na tem, da si izberemo neki delilni element (*pivot*) in razdelimo zaporedje, ki ga urejamo, na dva dela: tiste, ki so manjši od delilnega elementa, in tiste, ki so večji; potem uredimo vsak del posebej z rekurzivnim klicem. Za naše potrebe lahko ta postopek prilagodimo tako, da izberemo dva delilna elementa in razdelimo zaporedje na tri dele: zaboje, ki so lažji od obeh delilnih; zaboje, ki so težji od obeh delilnih; in zaboje, ki so po teži med delilnima. Pri vsakem zaboju lahko z enim izračunom mediane ugotovimo, v katerega od teh treh delov sodi. Ko so vsi zaboji razdeljeni, uredimo vsak del posabej z rekurzivnim klicem.

Tudi tu je obraba načeloma precej neenakomerna: delilna zaboja na glavnem klicu rekurzije sodelujeta v kar $O(n)$ računanjih mediane. Malo bolje je, če zabojev ne premeščamo šele na koncu (kot to počne naša prvotna rešitev s podprogramom ZakljuciUrejanje), pač pa že sproti [87 točk]; ali pa pred vsakim računanjem mediane izvedemo dve zamenjavi, s katerima premaknemo oba delilna elementa na dva naključno izbrana podstavka [93 točk]; za res enakomerno obrabo vseh podstavkov pa moramo tudi tu uporabljati prej opisani uravnoteževalnik obrabe, tako kot pri prvotni rešitvi [97 točk].

• Poiščimo najprej najlažji in najtežji zaboj; to lahko storimo takole: izračunajmo mediano poljubnih treh zabojev in zanjo vemo, da ta zaboj ni niti najlažji od vseh niti najtežji od vseh. Ta zaboj torej v mislih pobrišimo, spet izračunajmo mediano poljubnih treh (preostalih) zabojev, pobrišimo tudi njo in tako naprej; ko nam ostaneta le dva zaboja, vemo, da sta to najlažji in najtežji zaboj. Recimo enemu od njiju m; predpostavili bomo, da je najlažji (če pa je v resnici največji, bo učinek le ta, da bomo na koncu dobili padajoče namesto naraščajoče urejeno zaporedje zabojev, kar je tudi povsem sprejemljivo). Zdaj lahko poljubna dva druga zaboja, recimo a in b, primerjamo po teži tako, da izračunamo mediano zabojev m, a in b; mediana bo v tem primeru

tisti izmed a in b, ki je lažji izmed teh dveh zabojev. Zdaj znamo torej primerjati dva zaboja po teži, to pa lahko podamo kot primerjalno funkcijo podprogramu sort iz C++ove standardne knjižnice, pa nam bo on uredil zaboje.

Ta rešitev ima sicer eno ali dve slabosti: zaboj m sodeluje v $O(n \log n)$ izračunih mediane, torej je obraba zelo neenakomerna (kar sicer lahko tudi tu rešujemo z uravnoteževalnikom); in konstanta, ki se skriva v tem $O(\cdot)$, je večja kot prej, kajti z vsakim računanjem mediane v resnici primerjamo le dva elementa, zato si lahko predstavljamo, da bo treba gotovo vsaj $n \log_2 n$ takšnih primerjav, ne pa le $n \log_3 n$. [60 točk brez uravnoteževanja obrave, 90 z njim.]

• Tudi urejanje z zlivanjem (*merge sort*) lahko prilagodimo za našo nalogo. Tu se med drugim soočimo s težavo, da pri urejanju z medianami ne moremo ločiti med primerom, ko je neko zaporedje urejeno naraščajoče, in primerom, ko je urejeno padajoče. Ko zato zlivamo dve krajši urejeni zaporedji v eno daljše, je mogoče, da je eno od njiju naraščajoče, drugo pa padajoče. Ta primer lahko odkrijemo z največ tremi računanji mediane in po potrebi eno od zaporedij obrnemo, tako da sta potem obe urejeni na enak način: naj bo z_1 zaboj na začetku prvega zaporedja, k_1 pa tisti na koncu; predpostavimo, da je z_1 lažji od k_1; v mislih si lahko predstavljamo, da nam tadva razdelita vse druge zaboje na tri skupine: (1) tisti, ki so lažji od z_1; (2) tiste, ki so med z_1 in k_1; (3) tiste, ki so težji od k_2. Za vsakega od z_2 in k_2 poglejmo (s po enim izračunom mediane), v katero od teh treh skupin spada. Če z_2 v zgodnejšo skupino kot k_2, je vse v redu; če v kasnejšo, je treba drugo zaporedje obrniti; če pa padeta oba v isto skupino, lahko izračunamo še mediano zabojev z_1, z_2 in k_2 in iz rezultata vidimo, ali bo treba drugo zaporedje obrniti ali ne.

Recimo torej zdaj, da imamo dve zaporedji, obe urejeni naraščajoče (ali obe padajoče), in da ju hočemo zliti. Označimo i-ti zaboj v prvem zaporedju z a_i, v drugem pa z b_i. Izračunajmo mediano zabojev a_1, a_2 in b_1; če je to b_1, lahko v izhodno zaporedje premaknemo a_1 in b_1; če je mediana a_1, lahko v izhodno zaporedje premaknemo b_1; če pa je mediana a_2, lahko v izhodno zaporedje premaknemo a_1 in a_2. Tako nadaljujemo, dokler ne pridemo do konca obeh vhodnih zaporedij. Če ostane v prvem le še en element, v drugem pa vsaj dva, lahko razmišljamo enako, le njuni vlogi se zamenjata. Če pa ostane v vsakem po en element, lahko izračunamo mediano teh dveh ter zadnjega elementa, ki smo ga pred njima premaknili v izhodno zaporedje; ta mediana nam bo povedala, kateri od obeh preostalih elementov je manjši in ga moramo preseliti v izhodno zaporedje najprej, potem pa še drugega.

Zdaj torej znamo zlivati krajša urejena zaporedja v daljša. Postopek poženemo tako, da razdelimo naše zaporedje n zabojev na skupine po 2 (vsaka od njih je že sama po sebi urejena), nato pa jih zlivamo po dve in dve v vse daljše in daljše skupine, dokler ne ostane ena sama skupina z vsemi n zaboji. Tudi ta postopek izvede $O(n \log n)$ računanj mediane. [90 točk brez uravnoteževanja obrabe, 100 z njim.]

Naloge so sestavili: ulične luči — Benjamin Bajd; stoli — Bor Grošelj Simić; oviratlon, prisotnost — Tomaž Hočevar; lučka — Gregor Kikelj; padalski izlet, valj — Vid Kocijan; kibi, mebi, konkordanca, nedeljiva hramba — Mark Martinec; tehtnica — Polona Novak in Mark Martinec; neurejene besede — Jasna Urbančič; videostena — Borut Žnidar; špijonaža, urejanje z medianami — Janez Brank.

Vprašanja, pripombe, komentarji, popravki ipd. v zvezi z nalogami in rešitvami so dobrodošli: ⟨*janez@brank.org*⟩.