NALOGE ZA PRVO SKUPINO OŠ

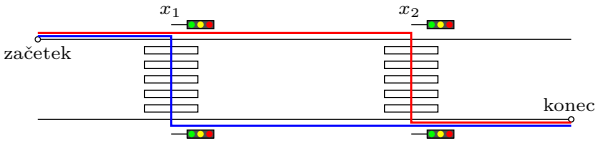
1. Čez cesto

Janko želi čim prej priti do šole, ker skoraj že zamuja na prvo uro pouka. K sreči se Jankova hiša in njegova šola nahajata ob isti cesti, tako da do tja ni zelo daleč, težava je le v tem, da je šola na drugi strani ceste in jo mora Janko zato prečkati. Ker je vzoren državljan, Janko cesto vedno prečka na prehodu za pešce. K sreči sta na poti do šole še dva semaforja.

Janko bi rad prečkal cesto tako, da bo čim manj časa čakal na zeleno luč in bo na cilju kar se da hitro. Ker svojo pot v šolo že zelo dobro pozna, ve, da se bo zelena luč na prvem semaforju prvič prižgala čez T_1 sekund, na drugem pa čez T_2 sekund. Ve tudi, da bo do obeh semaforjev prišel, preden se zelena luč prvič prižge.

Razdalja od Jankovega doma do šole vzdolž ceste je D metrov. Širina ceste je S metrov. Prvi semafor se nahaja X_1 metrov od Jankovega doma, drugi pa X_2 metrov. Janko za en meter poti do šole potrebuje natanko eno sekundo.

Napiši program, ki bo ugotovil, na katerem semaforju se Janku bolj splača čakati, in koliko časa (v sekundah) bo v tem primeru hodil od doma do šole. Program naj s standardnega vhoda prebere števila D, S, X_1 in T_1 ter X_2 in T_2,[1] nato pa naj na standardni izhod izpiše, na katerem semaforju se Janku bolj splača prečkati cesto in koliko časa bo porabil za pot. Če se zgodi, da bi pri obeh semaforjih na prečkanje čakal enako dolgo, lahko izpišeš kateregakoli izmed semaforjev. Vsa vhodna števila bodo med 4 in 1000.



Primer vhoda:

```
20 4
4 7
6 8
```

Pripadajoči izhod:

```
2 26
```

Komentar: v primeru je razdalja vzdolž ceste 20, širina 4, prvi semafor se nahaja na razdalji 4 metre od Jankovega doma in bo prvič zelen 7 sekund po Jankovem odhodu od doma, drugi pa se nahaja na razdalji 6 metrov in bo prvič zelen 8 sekund po Jankovem odhodu od doma. Janko bo na drugem semaforju čakal le 2 sekundi, tako da bo za pot v šolo potreboval $20 + 4$ sekund za hojo in še 2 za čakanje, torej skupno 26 sekund.

2. Zlogi skibidi

V Gruziji se je pojavilo novo ljudstvo inteligentnih WC-školjk, s katerimi pa se je izjemno težko sporazumevati, saj njihova sposobnost govora še vedno peša. Izgovorijo lahko namreč le zloge „`s`“, „`ki`“, „`bi`“ in pa „`di`“. Pomagaj znanstvenikom razvozlati ta skrivnostni jezik.

[1] V besedilu naloge na tekmovanju sta bila tu X_1 in T_1 zamenjana, prav tako pa tudi X_2 in T_2. Za objavo v biltenu smo to spremenili, da je bolj skladno s primerom vhoda kasneje v besedilu.

Podan je niz dolžine n (n je kvečjemu 1000), sestavljen iz teh štirih zlogov. **Napiši program**, ki prebere število n ter podani niz s standardnega vhoda in prešteje, kolikokrat se v nizu pojavi kateri izmed zlogov. Rezultate naj program izpiše enega za drugim na standardni izhod, najprej za „`s`“, potem za „`ki`“, nato za „`bi`“ in na koncu še za „`di`“.

Primer vhoda:

```
8
skibidi
```

Pripadajoči izhod:

```
1 1 1 1
```

Še en primer vhoda:

```
25
skiskiskibidiskibidiskidi
```

Pripadajoči izhod:

```
5 5 2 3
```

Še en primer vhoda:

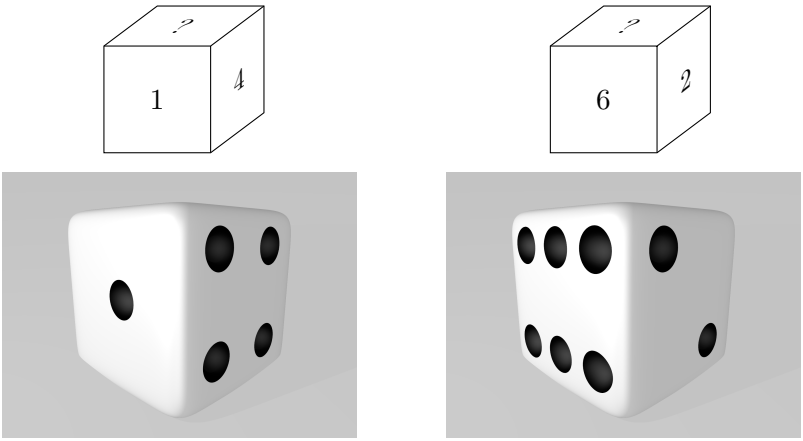
```
5
sssss
```

Pripadajoči izhod:

```
5 0 0 0
```

3. Kocke

Igor meče dve igralni kocki. To nista čisto običajni kocki, saj ne vemo točno, na kateri ploskvi je katero število. Vemo samo, da se števili na nasprotnih ploskvah seštejeta v 7. Da je igra bolj zanimiva, ko Igor vrže kocki, ne pogleda, katero število je na vrhu, ampak pogleda, katera števila vidi s strani kocke (vedno tako, da vidi dve števili na vsaki kocki). Zanima ga, kaj je največje in kaj je najmanjše skupno število pik na zgornjih ploskvah obeh kock.



Napiši program, ki s standardnega vhoda prebere opisa obeh vrženih kock in napove, kaj je najnižja in kaj najvišja možna vsota vrednosti na vrhu kocke. Rezultata naj izpiše na standardni izhod, najprej najnižjo možno vsoto, potem pa še najvišjo.

Primer vhoda:

```
1 4
6 2
```

Pripadajoči izhod:

```
5 9
```

Še en primer vhoda:

```
2 3
3 5
```

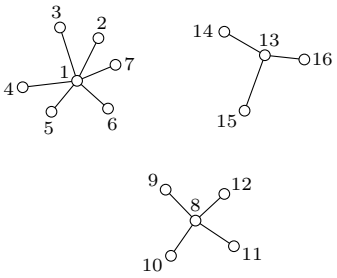
Pripadajoči izhod:

```
2 12
```

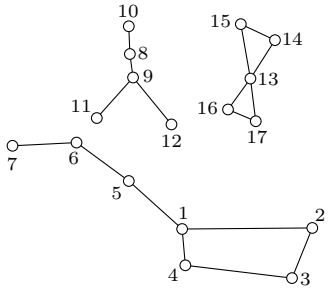
Komentar: oglejmo si primer z zgornje slike. Ker vidimo na prvi kocki stranico s štirimi pikami, vemo, da je tudi stranica s tremi pikami ob strani (nasproti te s tremi pikami). Podobno lahko sklepamo za 1 in 6. Na zgornji in spodnji strani sta torej stranici z dvema in s petimi pikami. Na drugi kocki na podoben način lahko sklepamo, da sta zgoraj in spodaj stranici s po tremi in štirimi pikami, torej je najmanjše število, ki ga lahko pridelamo, 2 s prve kocke in 3 z druge kocke — skupaj 5; največje pa 9.

4. Metkine zvezdice

Tabornica Metka ponoči ob ognju rada opazuje zvezde. Pred davnimi leti so astronomi določili, kako potekajo ozvezdja na nočnem nebu, Metke pa ta ozvezdja nikoli niso zanimala. Zvezde raje povezuje v večje skupine, ki izgledajo kot večje zvezde. V vsaki taki skupini je ena sredinska zvezda, ki je povezana z nekaj sosedi (glej sliko).



(*a*) Metkine zvezdice.



(*b*) Ozvezdja, ki jih ni narisala Metka.

Nekega večera je Metka označila zvezde na nekem delu neba s števili od 1 do n ter si v zvezek zapisala, s katerimi sosedami je vsaka zvezda povezana. Zvezek pa se je pri pospravljanju šotora pomešal z zvezkom Metkine sestre, v katerega je ta zapisala nekaj ozvezdij, ki se ne ravnajo po Metkinih pravilih. Pomagaj tabornikom ugotoviti, kateri zvezek pripada komu.

Napiši program, ki prejme število n, nato pa n seznamov števil, vsakega v svoji vrstici. Seznam v i-ti od teh vrstic opisuje, s katerimi zvezdami je povezana zvezda z oznako i. Program naj izpiše, ali se podana množica zvezd ravna po Metkinih pravilih ali ne.

Primer vhoda:	Pripadajoči izhod:
12 4 6 2 5 3 7 1 1 1 1 1 1 9 11 12 10 8 8 8 8	Metkine zvezdice

Primer vhoda:	Pripadajoči izhod:
12 2 5 4 1 3 2 4 1 3 6 1 5 7 6 9 10 8 11 12 8 9 9	Obicajna ozvezdja

Komentar: prvi od teh dveh primerov opisuje levo in spodnje ozvezdje iz Metkinega zvezka z zgornje slike, drugi primer pa levo in spodnje ozvezdje iz sestrinega zvezka z zgornje slike.

NALOGE ZA DRUGO SKUPINO OŠ

1. Nalepke

Imamo album z nalepkami, oštevilčenimi z $1, 2, \ldots, n$. Poleg albuma smo si zapisali zaporedne številke vseh nalepk, ki smo jih dobili. Zanima nas, katere nalepke nam še manjkajo, da napolnimo album.

Podana je velikost albuma in seznam nalepk, ki smo jih že dobili. Po vrsti izpiši števila nalepk, ki jih še nimamo, vsako v svojo vrstico.

Vhodni podatki. V prvi vrstici se nahajata n (število nalepk v albumu) in m (število nalepk, ki jih že imamo). V naslednjih m vrsticah se nahajajo nalepke, ki jih že imamo.

Omejitve vhodnih podatkov: $1 \leq n \leq 1\,000\,000$; $1 \leq m \leq 2\,000\,000$.

Izhodni podatki. Po vrsti izpiši zaporedna števila vseh nalepk, ki jih še nimamo. Če nam ne manjka nobena nalepka več, izpiši „`Album poln`“ (brez narekovajev).

Primer vhoda:

```
6 10
1
5
4
3
1
3
1
3
4
5
```

Pripadajoči izhod:

```
2
6
```

Še en primer vhoda:

```
2 3
1
2
2
```

Pripadajoči izhod:

```
Album poln
```

2. SMS

Oskar se je odločil, da ne želi več imeti pametnega telefona in da bo odslej uporabljal telefon na tipke, takega, kot so jih uporabljali v starih časih. Žal pa se Oskar ni zavedal vseh težav, ki pridejo s starinskimi telefoni, in šele zdaj, ko je že napisal obširno besedilo, ki ga želi poslati prijateljici, ugotavlja, kako nadležno je pošiljanje daljših sporočil brez aplikacij za klepetanje, po SMS-ih. Obstaja namreč omejitev dolžine posameznega SMS-a, zato Oskar dolgih sporočil ne more poslati v kosu, ampak jih mora razdeliti, to pa mu predstavlja kar nekaj težav.

Pomagaj Oskarju pravilno razdeliti besedilo. Označimo omejitev dolžine SMS-ov s k. Da bi poslali sporočilo, daljše od k znakov, ga moramo razdeliti na več delov dolžine k, in sicer tako, da najprej pošljemo prvih k znakov, potem drugih k znakov in tako naprej, na koncu pa še vse preostale znake. Vsakemu sporočilu moramo prirediti tudi zaporedno številko, da jih Oskar slučajno ne bo poslal v napačnem vrstnem redu.

Vhodni podatki. V prvi vrstici se nahajata število k — največje število znakov, ki jih lahko pošljemo naenkrat. V drugi vrstici se nahaja besedilo, ki ga želimo poslati.

Omejitve vhodnih podatkov: besedilo, ki ga želimo poslati, bo imelo kvečjemu 250 znakov, omejitev dolžine posameznega SMS-a pa bo med 5 in 50. Besedilo bo sestavljeno iz velikih in malih črk angleške abecede, števil, presledkov in ločil „`!`“, „`?`“, „`.`“ ter „`,`“. Zagotovo se ne bo začelo ali končalo s presledkom, prav tako se nikjer ne bosta pojavila dva zaporedna presledka.

Izhodni podatki: izpiši toliko vrstic, kolikor bo moral Oskar poslati SMS-ov , da bo poslal celotno besedilo, ki ga je napisal. Vrstice naj se začnejo z zaporedno številko SMS-a. V i-ti vrstici naj bo ta oblike „i/?“. Izjema je zadnja vrstica, kjer pa naj bo oblike „i/i“. Zaporedni številki naj sledita dvopičje in presledek, za presledkom pa izpiši i-ti SMS.

Primer vhoda:

```
10
Hej Manca, bi mi lahko pomagala pri domaci nalogi iz matemati
     ke? Potem greva pa lahko se v kino, ali pa na sladoled.
```

Opomba: ker je druga vrstica predolga, je tu za potrebe prikaza zapisana tako, da se nadaljuje še v tretji vrstici (kot je nakazano z zamikom).

Pripadajoči izhod:

```
1/?: Hej Manca,
2/?:  bi mi lah
3/?: ko pomagal
4/?: a pri doma
5/?: ci nalogi
6/?: iz matemat
7/?: ike? Potem
8/?:  greva pa
9/?: lahko se v
10/?:  kino, ali
11/?:  pa na sla
12/12: doled.
```

Še en primer vhoda:

```
7
Manca, pomagaj
```

Pripadajoči izhod:

```
1/?: Manca,
2/2: pomagaj
```

3. Grošev pest

Megumin in Yunyun se nahajata v velikem starem gozdu iglavcev. Odpravili sta se iskat antično vrsto bora, ki naj bi kot sad namesto storžev obrodili groše. Želita si namreč kupiti čarovniško knjigo, ki pa seveda ni zastonj. Knjiga ju bo stala k *grošljev*, vsak *grošelj* pa je vreden 13 *grošev*. Zato si čimprej želita pridobiti dovolj *grošljev* in kupiti knjigo, preden se njena cena zaradi inflacije dvigne.

S pomočjo magije smo že ugotovili vse pozicije bližnjih grošev in kdaj bodo dozoreli ter z veje padli na tla. Če se neki groš nahaja na višini h_i in bo dozorel ob času t_i (merjeno v sekundah), bo po trenutku, ko dozori, potreboval še h_i sekund, da doseže tla. Torej se bo prvič dotaknil tal ob času $t_i + h_i$.

Megumin in Yunyun lahko s skupnimi močmi pričarata urok, ki vse groše na tleh v trenutku prestavi v njun mošnjiček, ampak uporabita ga lahko le enkrat. Zanima ju najmanjši možen čas, ob katerem lahko uporabita urok, da bo v njunem mošnjičku vsaj k grošljev.

Vhodni podatki. V prvi vrstici se nahajata števili n in k, število grošev na drevesu in koliko grošljev Megumin in Yunyun potrebujeta.

V naslednji vrstici se nahaja n števil h_i. Vsako število h_i označuje, kako visoko se nahaja i-ti groš.

V tretji vrstici se nahaja n števil t_i. Število t_i označuje, ob katerem času bo i-ti groš padel na tla.

Omejitve: $1 \leq k \leq n \leq 10^6$; $1 \leq h_i \leq 10^9$; $1 \leq t_i \leq 10^9$. *Izhodni podatki.* Če rešitev obstaja, izpiši najmanjši možen čas, pri katerem lahko pridobita vsaj k grošljev. Če rešitev ne obstaja, izpiši „`Morali bi uporabiti eksplozivno magijo`“, brez navednic.

Primer vhoda:

```
4 1
4 2 1 4
4 2 3 1
```

Pripadajoči izhod:

```
Morali bi uporabiti eksplozivno magijo
```

Še en primer vhoda:

```
13 1
1 1 2 1 1 1 1 1 1 1 1 1 1
1 1 3 1 1 1 1 1 1 1 1 1 1
```

Pripadajoči izhod:

```
5
```

Komentar: pri prvem od gornjih dveh primerov imamo samo 4 *groše*, potrebujemo pa 1 *grošelj*, torej nam manjka 9 *grošev*, zato rešitev ne obstaja. Pri drugem primeru dobimo 12 grošev po dveh sekundah, za zadnjega pa moramo počakati do 5-te sekunde. Odgovor je zato 5.

4. Navijači

Bliža se velika tekma in programerski športni klub je ugotovil, da so njegovi igralci toliko sedeli za računalniki, da so že pozabili, kako igrati. Namesto da bi osvojili pokal, so se zato odločili, da bodo osvojili srca svojih navijačev; in to tako, da bodo vrgli ogromno žog na tribuno. Ker pa želijo dati vsem svojim oboževalcem enake priložnosti, da žogo ujamejo, potrebujejo tvojo pomoč.

Gledalci na tribuni sedijo v eni ravni vrsti. Ker je tik pred tekmo pihala divja burja, so stoli razmetani in niso postavljeni enakomerno. Trenerji v klubu so bili že na veliko tekmah, zato vedo, kako bodo gledalci reagirali, ko vidijo žogo na tribuni. Najbližjih nekaj gledalcev bo nemudoma steklo k žogi in jo poskusilo uloviti, nato pa se bodo hitro posedli nazaj na svoja mesta, da jim ne bi sosed spil pijače. Trenerji želijo, da dobi čim več gledalcev priložnost steči proti bližnji žogi. Ker pa sami niso dobri v programiranju, so prosili tebe, da jim pomagaš.

Napiši program, ki sprejme podatke o postavitvi stolov na tribunah ter o lokaciji vrženih žog in izračuna, koliko navijačev je med tekmo dobilo priložnost ujeti žogo.

Vhodni podatki. V prvi vrstici vhoda se nahajajo tri števila: n (število gledalcev), k (število najbližjih gledalcev, ki poskusijo ujeti vsako žogo) in q (število žog, vrženih na tribuno). V drugi vrstici je n števil x_i, ki označujejo pozicije stolov v decimetrih od levega roba tribune. Pozicije stolov niso nujno urejene. Sledi še q vrstic; v i-ti od njih je eno samo število y_i, ki pove lokacijo, kamor je padla i-ta žoga, merjeno v decimetrih od levega roba tribune.

Omejitve vhodnih podatkov: $1 \le k \le n \le 10^5$; $1 \le q \le 10^5$; $1 \le x_i \le 10^5$; $1 \le y_i \le 10^5$. Nobena dva stola si nista bližje kot decimeter. Zagotovljeno bo, da bosta k-ti in $(k+1)$-vi najbližji navijač različno oddaljena. V 90 % testnih primerov velja $k \le 20$, v zadnjih 10 % pa $1 \le k \le 10^5$. V prvih 30 % testnih primerov dodatno velja $n \cdot q \le 10^6$.

Izhodni podatki. Program naj izpiše eno samo število: koliko navijačev je imelo med tekmo priložnost, da bi ujeli žogo.

Primer vhoda:

```
5 2 2
1 5 10 6 20
3
6
```

Pripadajoči izhod:

```
3
```

REŠITVE NALOG ZA PRVO SKUPINO OŠ

1. Čez cesto

Ker se Janko premika s hitrostjo enega metra na sekundo, pride do prvega semaforja X_1 sekund po začetku svoje poti. Naloga zagotavlja, da se na tem semaforju prvič prižge zelena luč šele T_1 sekund po začetku Jankove poti in da je to kasneje od časa, ko bo prišel do semforja; če hoče torej prečkati cesto pri tem semaforju, bo moral čakati $T_1 - X_1$ sekund. Podoben razmislek za drugi semafor pove, da bi moral čakati $T_2 - X_2$ sekund, če bi hotel prečkati cesto pri njem. Čas hoje pa je v vsakem primeru enak: D sekund hodi ob cesti (nekaj časa na eni strani, nekaj na drugi), S sekund pa čez cesto. Najhitreje bo torej do šole prišel, če bo prečkal cesto pri tistem semaforju, kjer je treba čakati manj časa; pogledati moramo le, kateri od obeh časov čakanja je manjši, $T_1 - X_1$ ali $T_2 - X_2$.

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

int main()
{
  // Preberimo vhodne podatke.
  int D, S, X1, T1, X2, T2;
  cin >> D >> S >> X1 >> T1 >> X2 >> T2;

  // Izračunajmo, kje in kako dolgo bo čakal.
  int kjeCakati, casCakanja;
  if (T1 - X1 < T2 - X2) kjeCakati = 1, casCakanja = T1 - X1;
  else kjeCakati = 2, casCakanja = T2 - X2;

  // Izpišimo rezultate.
  cout << kjeCakati << " " << (D + S + casCakanja) << endl;
  return 0;
}
```

2. Zlogi skibidi

Nalogo lahko rešimo tako, da vhodni niz pregledamo v zanki po znakih, pri tem pa vzdržujemo štiri celoštevilske spremenljivke, v katerih štejemo pojavitve vsakega od iskanih podnizov. Ko opazimo v nizu znak `s`, povečajmo števec pojavitev podniza „`s`“; ko pa opazimo znak `i`, poglejmo prejšnji znak in če je bil ta eden od `k`, `b` ali `d`, moramo povečati števec pojavitev ustreznega izmed podnizov „`ki`“, „`bi`“ ali „`di`“. Na koncu vse štiri števce izpišemo.

Ker je niz kratek (dolg največ 1000 znakov), bi lahko celega naenkrat prebrali v pomnilnik; ni pa nujno, da to naredimo, saj ga lahko beremo po znakih in sproti štejemo pojavitve zlogov. Pri tem moramo poleg trenutnega znaka hraniti le še prejšnji znak (spodnji program v ta namen uporablja spremenljivko prej), vse še zgodnejše znake pa lahko sproti pozabljamo.

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

int main()
{
  int s = 0, ki = 0, bi = 0, di = 0; // Števci pojavitev posameznih podnizov.
```

```
  char prej = ' '; // Prejšnji znak.
  int n; cin >> n; // Preberimo dolžino vhodnega niza.
  // Preberimo vhodni niz in pri tem štejmo pojavitve podnizov.
  while (n-- > 0) {
    char c; cin >> c; // Preberimo naslednji znak.
    // Poglejmo, ali prejšnji in trenutni znak tvorita kakšnega od podnizov, ki nas zanimajo.
    if (c == 's') ++s;
    else if (c != 'i') { }
    else if (prej == 'k') ++ki;
    else if (prej == 'b') ++bi;
    else if (prej == 'd') ++di;
    prej = c; }
  // Izpišimo rezultate.
  cout << s << " " << ki << " " << bi << " " << di << endl;
  return 0;
}
```

3. Kocke

Ker se števili na nasprotnih ploskvah kocke seštejeta v 7, tvorita enega od treh parov: $(1, 6)$, $(2, 5)$ in $(3, 4)$. Na ploskvah, ki ju vidimo (in dobimo v vhodnih podatkih), imamo iz dveh od teh treh parov po eno število — recimo tema številoma a in b. Preveriti moramo torej, iz katerih dveh parov sta prebrani števili, potem pa vemo, da tretji (preostali) par predstavlja zgornjo in spodnjo ploskev kocke. To bi se vsekakor dalo preveriti z nekaj pogojnimi stavki; konec koncev je le 24 možnosti glede tega, kateri dve števili lahko dobimo za a in b. Z malo razmisleka pa lahko sestavimo še lažjo in elegantnejšo rešitev. V mislih predstavimo vsak par z manjšim izmed obeh števil v paru; imamo torej pare 1, 2 in 3. Če je kakšno od prebranih števil a in b večje od 3, ga odštejmo od 7, da tako dobimo manjše od obeh števil v njegovem paru; potem vemo, da je vsota vseh treh parov $1 + 2 + 3 = 6$, torej lahko od nje odštejemo oba para, ki ju vidimo, in dobimo tretji par, ki nas zanima: $c := 6 - a - b$. To je torej manjša od obeh vrednosti, ki bi se utegnili pojaviti na zgodnji ploskvi kocke; večja je potem $7 - c$. To dvoje moramo sešteti po obeh kockah, pa dobimo iskana rezultata.

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

// Vrne najmanjše možno število na zgornji ploskvi, če vidimo ploskvi s številoma a in b.
int TretjiPar(int a, int b)
{
  if (a >= 4) a = 7 - a;   // Zdaj je a manjše število iz prvega para.
  if (b >= 4) b = 7 - b;   // Zdaj je b manjše število iz drugega para.
  return 6 - a - b;        // Manjša števila iz vseh treh parov se seštejejo v 6.
}

int main()
{
  // Določimo najmanjšo možno vrednost prve kocke.
  int a, b; cin >> a >> b;
  int c = TretjiPar(a, b);
  // Določimo najmanjšo možno vrednost druge kocke.
```

```
  cin >> a >> b; c += TretjiPar(a, b);

  // Izpišimo rezultate.
  cout << c << " " << (14 - c) << endl;
  return 0;
}
```

4. Metkine zvezdice

Pri Metkinih ozvezdjih je sredinska zvezda edina, ki ima več kot eno sosedo; vse njene sosede pa so preostale (ne-sredinske) zvezde njenega ozvezdja in imajo po eno samo sosedo (namreč sredinsko zvezdo). Dovolj je torej, če za vsako zvezdo z, ki ima več kot eno sosedo, preverimo, če imajo vse njene sosede vsaka le po eno sosedo. Tega, ali je ta edina soseda res ravno z, niti ni treba posebej preverjati, kajti če ne bi bila, bi bila to že napaka v vhodnih podatkih (če je ena zvezda soseda druge, je namreč tudi druga zvezda soseda prve), za takšne napake pa smemo predpostaviti, da jih ni, razen če bi naloga posebej povedala, da se lahko pojavijo.

Če pišemo implementacijo rešitve v C++, je treba pri branju vhodnih podatkov nekaj pazljivosti; koristno je, če beremo sezname sosed po vrsticah, šele nato pa iz posamezne vrstice izluščimo številke zvezd v njej.

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

int main()
{
  // Preberimo število zvezd.
  int n; cin >> n >> ws; // ws poskrbi, da preberemo tudi konec vrstice.

  // Za vsako zvezdo preberimo seznam njenih sosed.
  vector<vector<int>> sosede(n);
  for (auto &v : sosede)
  {
    // Preberimo naslednjo vrstico.
    string vrstica; getline(cin, vrstica);

    // Preberimo vse številke zvezd iz te vrstice.
    for (istringstream iss(vrstica); ; ) {
      int zvezda; iss >> zvezda;
      if (! iss) break;
      v.emplace_back(zvezda); }
  }

  // Za vsako zvezdo z več kot eno sosedo preverimo,
  // če imajo vse te sosede za sosedo le njo.
  bool ok = true;
  for (auto &v : sosede) if (v.size() > 1) {
    for (int zvezda : v) if (sosede[zvezda - 1].size() != 1) { ok = false; break; }
    if (! ok) break; }

  // Izpišimo rezultat.
  cout << << (ok ? "Metkine zvezdice" : "Obicajna ozvezdja") << endl;
  return 0;
}
```

REŠITVE NALOG ZA DRUGO SKUPINO OŠ

1. Nalepke

Lahko si pripravimo tabelo oz. vektor n logičnih vrednosti, v katerem si bomo označevali, katere nalepke že imamo. Na začetku inicializirajmo vse elemente vektorja na 0, nato pa preberimo seznam m prejetih nalepk in v vektorju postavljajmo ustrezne vrednosti na **true**. Na koncu pojdimo v zanki po vseh nalepkah (od 1 do n), pri vsaki preverimo, ali jo imamo, in če je nimamo, to izpišimo. Imejmo tudi logično spremenljivko, v kateri si označimo, ali je album poln ali ne; na začetku predpostavimo, da je poln, če pa se pri izpisu izkaže, da nam kakšna nalepka manjka, potem vemo, da album ni poln. Če je album (po pregledu vseh nalepk od 1 do n) res poln, moramo to na koncu tudi izpisati.

Pri implementaciji v C++ in podobnih jezikih moramo paziti na to, da se indeksi v tabele in vektorje štejejo od 0 naprej, številke nalepk pri naši nalogi pa od 1 naprej. To lahko rešimo z odštevanjem 1 pri branju in prištevanjem 1 pri izpisu, lahko pa tako, da si pač rezerviramo $n + 1$ elementov dolgo tabelo oz. vektor (slednje naredi tudi naša spodnja rešitev).

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

int main()
{
  int n, m; cin >> n >> m;

  // Preberimo seznam prejetih nalepk in si označimo, katere že imamo.
  vector<bool> imamo(n + 1, false);
  while (m-- > 0) {
    int x; cin >> x; imamo[x] = true; }

  // Izpišimo številke manjkajočih nalepk.
  bool poln = true;
  for (int x = 1; x <= n; ++x) if (! imamo[x]) {
    cout << x << endl; poln = false; }

  // Izpišimo, če je album poln.
  if (poln) cout << "Album poln" << endl;
  return 0;
}
```

Namesto tabele ali vektorja bi lahko uporabili tudi razpršeno tabelo, npr. razred `unordered_set` iz C++ove standardne knjižnice. To bi bilo koristno, če je število (različnih) kart, ki jih imamo, majhno v primerjavi z n; če pa imamo veliko kart, morda celo vseh n, je takšna rešitev za neki konstanten faktor počasnejša od tiste z vektorjem in porabi tudi nekajkrat več pomnilnika.

2. SMS

Ena možnost je, da preberemo celo vrstico kot niz in nato v zanki izpisujemo po k znakov naenkrat kot posamezne SMSe. Iz dolžine niza ni težko izračunati, na koliko delov moramo niz razdeliti (deliti moramo dolžino niza s k in zaokrožiti rezultat navzgor, če se deljenje ne izide), in tako preverjati, ali smo trenutno pri zadnjem delu (od česar je odvisno, ali moramo izpisati vprašaj ali številko dela):

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

int main()
{
  // Preberimo vhodne podatke.
  int k; cin >> k >> ws; // ws poskrbi, da preberemo tudi konec vrstice.
  string s; getline(cin, s);

  // Izpišimo SMSe.
  int stDelov = (s.length() + k - 1) / k;
  for (int d = 1; d <= stDelov; ++d)
  {
    cout << d << "/";
    if (d == stDelov) cout << d; else cout << "?";
    cout << ": ";

    // Naslednji SMS sestavlja „dolzina“ znakov od indeksa „zacetek“ naprej.
    int zacetek = (d - 1) * k;
    int dolzina = (d < stDelov) ? k : s.length() - zacetek;
    cout.write(s.c_str() + zacetek, dolzina);
    cout << endl;
  }
  return 0;
}
```

Lahko pa smo še bolj varčni in vhodni niz beremo po delih, k znakov naenkrat, in jih sproti izpisujemo in pozabljamo. Lepo pri tej rešitvi je, da porabi le $O(k)$ pomnilnika ne glede na to, kako dolg je vhodni niz (pri naši nalogi ta izboljšava seveda ni zares pomembna, saj naloga zagotavlja, da bo vhodni niz dolg kvečjemu 250 znakov).

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

int main()
{
  int k; cin >> k >> ws; // ws poskrbi, da preberemo tudi konec vrstice.
  string sms(k, ' ');
  int del = 0;
  do {
    // Če je iz prejšnje iteracije cin-ov failbit prižgan,
    // ga moramo izključiti, preden lahko preberemo še kaj.
    cin.clear();

    // Preberimo naslednjih k znakov (ali manj, če se vrstica prej konča).
    cin.getline(sms.data(), k + 1);

    // Naslednji pogoj poskrbi za možnost, da je vhodno sporočilo prazen niz.
    if (cin.gcount() == 0) break;

    // Izpišimo številko tega SMSa.
    cout << ++del << "/";

    // Če getline ni prišel do konca vrstice, je vključil cin-ov failbit.
    // Iz tega lahko vidimo, če je trenutni del zadnji ali ne.
    if (cin) cout << del; else cout << "?";

    // Če bi z operatorjem << izpisali „sms“ namesto „sms.data()“,
```

```
    // bi se izpisalo k znakov, četudi smo v zadnji vrstici, ki je morda
    // krajša (na koncu bi dobili v izpisu null terminator in zadnji del prejšnje vrstice).
    cout << ": " << sms.data() << endl;

    // Če cin-ov failbit ni prižgan, zanko končamo, kajti to je znak,
    // da je metoda getline pri branju prišla do konca vrstice.
  } while (! cin);
  return 0;
}
```

Gornja rešitev se opira na dejstvo, da cin.getline iz standardne knjižnice prižge cin-ov failbit, če je prebrala k znakov in še ni dosegla konca vrstice. Stanje tega bita lahko preverimo tako, da cin pretvorimo v logično vrednost, npr. v pogoju pri stavku **if** ali **while**; tako bomo vedeli, ali smo dosegli konec vrstice (in je trenutni SMS zadnji) ali še ne. Paziti pa moramo na to, da dokler je failbit prižgan, iz toka ne moremo prebrati ničesar, zato ga pred naslednjim branjem izključimo (z metodo cin.clear).

Opisani pristop bi imel težave v primeru, če bi bilo besedilo, ki ga želimo poslati, prazen niz; v tem primeru bi namreč cin.getline prižgal failbit, četudi bi dosegel konec vrstice. Ta primer obravnavamo posebej (tako, da z cin.gcount pogledamo, koliko znakov je getline prebral), čeprav to v resnici ni zares potrebno, saj si je besedilo naloge bolj smiselno razlagati tako, kot da niz, ki ga želimo poslati, ne bo prazen.

Še ena podrobnost, na katero moramo paziti, pa je pri izpisu: spremenljivko sms smo inicializirali tako, da ima prostora za k znakov in še null terminator na koncu. Če je getline prebral manj kot k znakov, je za njimi primerno postavil tudi null terminator; toda če bi želeli potem spremenljivko sms izpisati z „cout << sms“, bi se izpisalo vseh k znakov v bloku pomnilnika, ki si ga je sms rezerviral ob inicializaciji, torej bi dobili v izpisu za koncem trenutne vrstice (in null terminatorjem) še zadnjih nekaj znakov prejšnje. Da to preprečimo, smo vsebino niza sms izpisali kot **char ***, torej s „cout << sms.data()“.

Oglejmo si še eno implementacijo tovrstne rešitve, tokrat s scanf in printf namesto s cin in cout:

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

int main()
{
  int k; scanf("%d ", &k);          // Preberimo dolžino SMSa in konec vrstice.
  string sms(k, ' ');               // Pripravimo prostor za k znakov.

  // Pripravimo opis formata, s katerim bomo brali po največ k znakov naenkrat.
  char format[50]; sprintf(format, "%%%d[^\n]", k);

  // Preberimo sporočilo po delih in jih sproti izpisujmo.
  int del = 0; bool konec;
  do {
    // Preberimo naslednjih k znakov (ali manj, če se vrstica prej konča).
    scanf(format, sms.data());

    // Poglejmo, ali smo na koncu vrstice.
    int c = getchar(); konec = (c == EOF || c == '\n');

    printf("%d/", ++del);           // Izpišimo številko tega dela.

    // Če je to zadnji del, izpišimo številko dela še enkrat.
```

```
    if (konec) printf("%d", del);
    // Sicer izpišimo vprašaj, pravkar prebrani znak „c“ pa vrnimo.
    else { printf("?"); ungetc(c, stdin); }
    printf(": %s\n", sms.data());    // Izpišimo besedilo trenutnega dela.
  } while (! konec);                 // Če smo na koncu vrstice, končajmo.
  return 0;
}
```

Najprej s scanf preberemo k (dolžino posameznega SMSa); presledek na koncu formatnega niza poskrbi, da scanf nato prebere (in preskoči) tudi znak za konec vrstice (prebral bi tudi morebitne presledke na začetku naslednje vrstice, torej tiste, kjer je besedilo, vendar naloga na srečo zagotavlja, da presledkov tam ne bo). V nadaljevanju bomo v zanki brali (spet s scanf) po največ k znakov besedila naenkrat. Spomnimo se, da če s scanf-om beremo niz s formatom `%s`, bo bral do prvega presledka; mi pa hočemo brati do konca vrstice, zato uporabimo format `%[^\n]`. Toda povedati moramo še, koliko največ znakov naj prebere; na primer, če bi hoteli brati po največ 12 znakov naenkrat, bi morali uporabiti format `%12[^\n]`. Takšen niz (s pravo vrednostjo k) si pripravimo v tabeli format, kasneje pa ga v vsaki iteraciji glavne zanke uporabimo pri klicu funkcije scanf.[7] Vprašanje je še, kako ugotoviti, ali smo že na koncu vrstice; v ta namen preberemo naslednji znak (s funkcijo getchar) in ga, če se izkaže, da to ni znak za konec vrstice, vrnemo nazaj (s funkcijo ungetc), da ga bomo v naslednji iteraciji zanke prebrali še enkrat.

3. Grošev pest

Naloga pravi, da moramo zbrati k grošljev, vsak pa je vreden 13 grošev; če je torej $13k > n$, je problem nerešljiv, saj grošev v vhodnih podatkih preprosto ni dovolj. Sicer pa lahko razmišljamo takole: če uporabimo urok ob času T, bomo pobrali vse tiste groše, za katere velja $t_i + h_i \leq T$. Če T počasi povečujemo, je teh grošev vedno več: število pobranih grošev se poveča vsakič, ko T doseže eno od vrednosti $t_i + h_i$. Ker moramo pobrati $13k$ grošev, naloga torej pravzaprav sprašuje po tem, katera je $(13k)$-ta najmanjša vrednost $t_i + h_i$ (po vseh n groših iz vhodnih podatkov). Poiščemo jo lahko na več načinov.

Ena možnost je, da vrednosti $t_i + h_i$ zložimo v seznam, ga uredimo naraščajoče in v njem preprosto odčitamo $(13k)$-to vrednost; ta rešitev ima zaradi urejanja časovno zahtevnost $O(n \log n)$.

Druga možnost je, da po vrsti pregledujemo vrednosti $t_i + h_i$ in pri tem vzdržujemo kopico, ki hrani najmanjših $13k$ doslej videnih vrednosti; največja med njimi, torej $(13k)$-ta najmanjša vrednost, je v korenu kopice. Ko pridemo do konca, vrednost iz korena kopice izpišemo. Časovna zahtevnost te rešitve je $O(n \log k)$.

Še boljša rešitev pa je algoritem quickselect, s katerim lahko $(13k)$-ti najmanjši element seznama dobimo v $O(n)$ časa.

Za potrebe naloge na našem tekmovanju so sicer vse te rešitve dovolj dobre, ker bo program tako ali tako porabil več časa za branje podatkov kot pa za iskanje $(13k)$-tega najmanjšega elementa. Za reševanje na tekmovanju ima druga rešitev

[7] Pri printf obstaja tudi možnost, da v formatnem nizu za širino namesto številke uporabimo zvezdico, želeno vrednost (v našem primeru bi bila to k) pa podamo kot parameter pri klicu funkcije; žal pa scanf tega ne podpira, zato si moramo formatni niz pripraviti vnaprej.

še to slabost, da je z njeno implementacijo malo več dela, pri prvi in tretji rešitvi pa moramo le poklicati eno od funkcij iz C++ove standardne knjižnice (sort oz. nth_element). Oglejmo si implementacijo tretje rešitve:

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

int main()
{
  ios_base::sync_with_stdio(false); // Za hitrejše branje in pisanje.
  int n, k; cin >> n >> k;
  k *= 13; // Odslej bo k število grošev (in ne grošljev), ki nas zanima.
  // Preverimo, ali je grošev sploh dovolj.
  if (k > n) { cout << "Morali bi uporabiti eksplozivno magijo" << endl; return 0; }
  // Preberimo višine h_i.
  vector<int> casi(n);
  for (auto &t : casi) cin >> t;
  // Preberimo čase t_i in jih prištejmo k višinam, da
  // za vsak groš dobimo čas, ko postane dostopen.
  for (auto &t : casi) { int ti; cin >> ti; t += ti; }
  // Izpišimo k-ti najmanjši čas iz tega seznama.
  nth_element(casi.begin(), casi.begin() + (k - 1), casi.end());
  cout << casi[k - 1] << endl; return 0;
}
```

Še opomba glede klica sync_with_stdio na začetku: z njim izključimo sinhronizacijo med cin in scanf, ki je ne potrebujemo (ker beremo samo s cin); privzeto je ta sinhronizacija vključena in zaradi nje je branje s cin veliko počasnejše kot s stdin, v našem primeru (ko bomo morali pri največjih testnih primerih prebrati dva milijona števil) celo toliko počasnejše, da bi program na tekmovanju prekoračil časovno omejitev. Ko pa sinhronizacijo izključimo, je branje s cin enako hitro kot s scanf; ta prijem pride na računalniških tekmovanjih pogosto prav.

4. Navijači

Za začetek je koristno x-koordinate navijačev urediti, tako da bomo v nadaljevanju predpostavljali, da velja $x_1 \le x_2 \le \cdots \le x_n$. Za dani položaj žoge y bomo skupino njej najbližjih k navijačev imenovali *soseščina* te žoge. Recimo, da sta v soseščini navijača x_i in x_j, pri čemer je $i < j$. Če velja $y \le (x_i + x_j)/2$, so vse x-koordinate z intervala $[x_i, x_j]$ oddaljene od y manj kot x_j; če pa velja $y \ge (x_i + x_j)/2$, so vse te x-koordinate oddaljene od y manj kot x_i; v obeh primerih lahko torej zaključimo, da morajo biti v soseščini točke y vsi navijači med x_i in x_j, torej $x_i, x_{i+1}, \ldots, x_{j-1}, x_j$. Soseščino torej vedno tvori nekaj zaporednih navijačev; vsaka soseščina je oblike $\{x_i, x_{i+1}, \ldots, x_{i+k-1}\}$, vprašanje je le, pri katerem navijaču (katerem i) se začne (za dani y).

Če vzamemo $y < x_1$, leži žoga levo od vseh navijačev in njeno soseščino tvori prvih k navijačev, torej $\{x_1, \ldots, x_k\}$. Recimo zdaj, da y počasi povečujemo (torej žogo premikamo proti desni). Ko y preseže mejo $(x_1 + x_{k+1})/2$, mu postane navijač x_{k+1} bližji kot navijač x_1, zato slednji izpade iz soseščine, vanjo pa pride x_{k+1}; nova

soseščina je $\{x_2, \ldots, x_{k+1}\}$. Do naslednje spremembe potem pride, ko y preseže mejo $(x_2 + x_{k+2})/2$, ko iz soseščine izpade x_2, vanjo pa pride x_{k+2}. V splošnem torej pri $y = (x_i + x_{i+k})/2$ iz soseščine izpade x_i, vanjo pa pride x_{i+k}.[8] Soseščina oblike $\{x_i, \ldots, x_{i+k-1}\}$ torej velja za žoge z območja $(x_{i-1} + x_{i+k-1})/2 \leq y \leq (x_i + x_{i+k})/2$. (Pravzaprav do enakosti v tej neenačbi ne more priti: če bi na primer žoga ležala točno na meji $y = (x_i + x_{i+k})/2$, bi bila njej k-ti in $(k+1)$-vi najbližji navijač (to sta x_i in x_{i+k}) enako oddaljena od žoge, za kar pa naloga zagotavlja, da se ne bo zgodilo.) Da nam ne bo treba delati z ne-celimi števili, je koristno pomnožiti to neenačbo z 2; če žoga pade na y, nas torej zanima tisti i, pri katerem je $M_{i-1} < 2y < M_i$, pri čemer smo meje definirali s formulo $M_i := x_i + x_{i+k}$. Vrednosti $M_1, \ldots, M_{n-k}$ si je koristno pripraviti v tabeli, potem pa bomo lahko za poljuben y poiskali pravi i z bisekcijo.

Takrat bomo torej vedeli, da za žogo y stečejo navijači $\{x_i, \ldots, x_{i+k-1}\}$. Naloga sprašuje, koliko je na koncu navijačev, ki so vsaj enkrat stekli za kakšno žogo. Ker je tako navijačev kot žog lahko do 10^5, si ne moremo privoščiti, da bi šli pri vsaki žogi v zanki po vseh k navijačih, ki so stekli za njo, in v neki tabeli za vsakega od teh navijačev označili, da je stekel za žogo. Pač pa lahko naredimo takole: naj bo Z_t število žog, za katerimi je stekel navijač x_t. Ko soseščina $\{x_i, \ldots, x_{i+k-1}\}$ steče za žogo, se vrednosti $Z_i, \ldots, Z_{i+k-1}$ povečajo za 1. To si lahko predstavljamo kot dve spremembi: najprej so se vse vrednosti od Z_i naprej povečale za 1, nato pa so se vse vrednosti od Z_{i+k} naprej zmanjšale za 1. Definirajmo torej $\triangle_t := Z_t - Z_{t-1}$; ko navijači od x_i do x_{i+k-1} stečejo za žogo, se zaradi tega $\triangle_i$ poveča za 1, vrednost $\triangle_{i+k}$ pa se zmanjša za 1. Vrednosti $\triangle_1, \ldots, \triangle_n$ bomo hranili v tabeli oz. vektorju; pri vsaki žogi moramo spremeniti le dve od teh vrednosti, na koncu pa lahko iz njih izračunamo Z_t-je (po formuli $Z_t = Z_{t-1} + \triangle_t$) in preštejemo, koliko navijačev ima $Z_t > 0$ (to so tisti, ki so stekli za vsaj eno žogo).

Kakšna je časovna zahtevnost te rešitve? Urejanje navijačev vzame $O(n \log n)$ časa; nato pri vsaki od q žog porabimo $O(\log n)$ časa za bisekcijo; izračun Z_t-jev na koncu nam vzame $O(n)$ časa; skupaj torej $O((n+q)\log n)$.

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

int main()
{
  // Preberimo položaje navijačev in jih uredimo.
  int n, k, q; cin >> n >> k >> q;
  vector<int> X(n); for (auto &xi : X) cin >> xi;
  sort(X.begin(), X.end());

  // Vrednost meje[i] nam pove, da pri y = meje[i] / 2 navijač X[i]
  // izpade iz soseščine, vanjo pa pride navijač X[i + k].
  vector<int> meje(n - k);
  for (int i = 0; i + k < n; ++i) meje[i] = X[i] + X[i + k];

  // Naj bo Z[t] število žog, pri katerih se je navijač X[t] znašel v soseščini.
  // Naj bo DZ[t] = Z[t] - Z[t - 1]. Ko pride žoga s soseščino X[i..i + k - 1],
```

[8] S takšnim razmislekom smo se na naših tekmovanjih že srečali; gl. *Bilten* 2014, str. 57, in *Bilten* 2017, str. 125.

```
  // se vrednosti Z[i..i + k − 1] povečajo za 1, zato pa se DZ[i] poveča za 1,
  vector<int> DZ(n + 1, 0);                    // DZ[i + k] pa se zmanjša za 1.
  while (q−− > 0) {
    // Preberimo naslednjo žogo in z bisekcijo določimo njeno soseščino.
    int y; cin >> y;
    int i = lower_bound(meje.begin(), meje.end(), 2 * y) − meje.begin();
    // V soseščini te žoge so navijači X[i..i + k − 1].
    ++DZ[i]; −−DZ[i + k]; }
  // Preštejmo navijače, ki imajo Z[t] > 0, in izpišimo rezultat.
  int Z = 0, rezultat = 0;
  for (int dz : DZ) if ((Z += dz) > 0) ++rezultat;
  cout << rezultat << endl; return 0;
}
```

Nalogo bi lahko rešili tudi tako, da bi koordinate žog uredili naraščajoče in potem to zaporedje zlivali z zaporedjem mej M_i; tako bi lahko za vsako žogo y videli, med katerima dvema mejama leži (torej pri katerem i je izpolnjen pogoj $M_{i-1} < 2y < M_i$). Res pa je, da s tem ničesar posebnega ne pridobimo: namesto $O(q \log n)$ časa za bisekcijo bi zdaj porabili $O(q \log q)$ časa za urejanje žog; če je $q \gg n$, je torej nova rešitev slabša od prejšnje, če pa je $q \ll n$, bo pri obeh rešitvah tako ali tako prevladoval čas urejanja navijačev, torej $O(n \log n)$.

REŠITVE NALOG ŠOLSKEGA TEKMOVANJA OŠ

1. Podaje

Recimo, da je v krogu n otrok in da je bilo vsega skupaj k podaj. Maja poda žogo v prvi podaji, nato spet v $(n+1)$-vi oddaji, nato v $(2n+1)$-vi oddaji in tako naprej. Majina m-ta podaja je torej $((m-1)\cdot n+1)$-va podaja celotne igre; če je to število večje od k (števila vseh podaj), potem Maja m-te povezave sploh ni izvedla. Da dobimo število Majinih podaj, moramo torej poiskati največji tak m, pri katerem je $(m-1)\cdot n+1 \le k$. Ta pogoj lahko predelamo v $(m-1)\cdot n \le k-1$, torej $m \le 1+(k-1)/n$. Največji celoštevilski m, ki ustreza temu pogoju, je torej $m = 1+\lfloor (k-1)/n \rfloor$, pri čemer znaka $\lfloor\cdot\rfloor$ pomenita zaokrožanje navzdol. Oglejmo si implementacijo te rešitve v C++ (spomnimo se, da operator /, ko ga uporabljamo na celih številih, rezultat že sam zaokroži navzdol, torej točno tako, kot si pri tej nalogi želimo):

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

int main()
{
  int n, k; cin >> n >> k;                // Preberimo vhodne podatke.
  cout << 1 + (k - 1) / n << endl;        // Izračunajmo rezultat in ga izpišimo.
  return 0;
}
```

Zapišimo to rešitev še v pythonu. Za celoštevilsko deljenje z zaokrožanjem navzdol moramo tu uporabiti operator //:

```
n, k = map(int, input().split())          # Preberimo vhodne podatke.
print(1 + (k - 1) // n)                   # Izračunajmo rezultat in ga izpišimo.
```

Ob zgornjem razmisleku o zaokroževanju se morda postavlja vprašanje, zakaj ne bi rezultata izračunali tako, da vhodni števili delimo v tipu **float** ali **double** in dobljeni rezultat potem zaokrožimo navzgor. Žal pri računanju s plavajočo vejico v računalniku pride do zaokrožitvenih napak, ki bi lahko povzročile, da s tem postopkom ne bi dobili pravega rezultata.[9]

Do enakih rezultatov lahko pridemo tudi z naslednjim razmislekom. Maja žogo poda enkrat na začetku vsakega kroga. Število zaključenih krogov dobimo s celoštevilskim deljenjem k/n. Vendar pa je iz drugega primera v nalogi razvidno, da to ni vedno pravi odgovor; lahko se namreč zgodi, da po vseh zaključenih krogih ostane še kakšna podaja. Tedaj bo Maja žogo podala še enkrat in količniku moramo prišteti ena. Za izračun končnega odgovora moramo torej preveriti, ali je število podaj deljivo s številom otrok v krogu. V naši gornji rešitvi bi torej lahko vrstico za izračun in izpis rezultata spremenili takole:

[9] Natančneje povedano, pri računanju s plavajočo vejico se rezultat deljenja zaokroži na najbližje število, ki je v izbranem tipu še predstavljivo; lahko se zgodi, da je to najbližje predstavljivo število célo in da je večje od pravega rezultata deljenja; takrat bi se torej pravi rezulat najprej zaokrožil navzgor na najbližje celo število in ko bi ga mi kasneje poskušali zaokrožiti navzdol, se ne bi več spremenil. Na primer: če imamo $k = n = 2^{24}+1$ in če delimo $k-1$ z n v tipu **float**, že dobimo napačen rezultat 1 (namesto nečesa, kar bi bilo malo pod 1 in kar bi lahko kasneje zaokrožili navzdol na 0). Pri tipu **double** nastopi enaka težava pri $k = n = 2^{53}+1$; in v splošnem, če imamo tip z b-bitno mantiso, nastopi težava pri $k = n = 2^{b+1}+1$.

```
cout << k / n + (k % n > 0 ? 1 : 0) endl;    // v C++
print(k / n + (1 if k % n > 0 else 0))       # v pythonu
```

2. Pribor

Podatke o žlicah lahko beremo in obdelujemo v zanki. Nov kup je treba začeti, ko je trenutna žlica večja od prejšnje; da bomo lahko preverjali ta pogoj, si je torej koristno velikost prejšnje žlice zapomniti v neki spremenljivki. Ko začnemo nov kup, je tudi primeren trenutek za izpis prejšnjega kupa. V ta namen potrebujemo podatek o tem, koliko žlic smo dodali nanj; imejmo torej še eno spremenljivko, ki šteje žlice na trenutnem kupu. Ko dodamo žlico na kup, povečamo ta števec za 1, ko pa začnemo nov kup, postavimo ta števec na 0. Pazimo še na to, da moramo na koncu izpisati tudi zadnji kup (tistega, na katerga smo malo prej dodali zadnjo, n-to prebrano žlico).

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

int main()
{
  int n; cin >> n;                    // Preberimo število žlic.
  // Vzdrževali bomo velikost prejšnje žlice in število žlic na trenutnem kupu.
  int prejsnja, stNaKupu = 0;
  while (n-- > 0)                     // V zanki obdelajmo vse žlice.
  {
    int zlica; cin >> zlica;          // Preberimo velikost naslednje žlice.
    // Če je večja od prejšnje, bo treba začeti nov kup.
    if (stNaKupu > 0 && zlica > prejsnja) {
      cout << stNaKupu << endl;       // Dosedanji kup izpišimo.
      stNaKupu = 0; }                 // Začnimo nov kup.
    // Trenutno žlico dodajmo na trenutni kup in si jo zapomnimo.
    ++stNaKupu; prejsnja = zlica;
  }
  // Izpišimo še zadnji kup.
  if (stNaKupu > 0) cout << stNaKupu << endl;
  return 0;
}
```

Preden preverjamo, ali je trenutna žlica večja od prejšnje, smo s pogojem stNaKupu > 0 preverili, ali nismo morda šele pri prvi žlici; tako se nam ni treba ukvarjati z vprašanjem, na kakšno vrednost naj inicializiramo spremenljivko prejsnja, da prva prebrana žlica gotovo ne bo večja od nje (drugače bi morali na primer predpostaviti, da so velikosti žlic manjše od neke znane maksimalne vrednosti ali kaj podobnega).

Zgornji program pazi še na eno malenkost, ki sicer ni najbolj nujna, ga pa naredi malo robustnejšega: pred izpisom zadnjega kupa preverimo, če je na njem sploh kakšna žlica; tako program v primeru, če je $n = 0$ (torej če je vhodni seznam žlic čisto prazen), ne bo izpisal ničesar, drugače pa bi izpisal število 0 (kot da bi bil nastal en kup z 0 žlicami).

Zapišimo to rešitev še v pythonu:

```
n = int(input())          # Preberimo število žlic.
stNaKupu = 0              # Število žlic na kupu.
for i in range(n):        # V zanki obdelajmo vse žlice.
  zlica = int(input())    # Preberimo velikost naslednje žlice.
  # Če je večja od prejšnje, bo treba začeti nov kup.
  if stNaKupu > 0 and zlica > prejsnja:
    print(stNaKupu)       # Dosedanji kup izpišimo.
    stNaKupu = 0          # Začnimo nov kup.
  # Trenutno žlico dodajmo na trenutni kup in si jo zapomnimo.
  stNaKupu += 1; prejsnja = zlica
# Izpišimo še zadnji kup.
if stNaKupu > 0: print(stNaKupu)
```

3. Palindromski oklepaji

V zanki bomo primerjali znake z začetka in s konca niza: najprej prvi in zadnji znak, nato drugi in predzadnji znak in tako naprej. Če se pri vsaki taki primerjavi izkaže, da sta znaka enaka, je niz palindrom; če se pri vsaki primerjavi izkaže, da sta si znaka ravno zrcalna, je niz zrcalen izraz; sicer pa ni nič od tega dvojega. (Oboje hkrati, palindrom in zrcalen, je lahko le, če je prazen; drugače sta namreč v vsakem palindromu prvi in zadnji znak enaka, torej si ne moreta biti zrcalna, torej tak niz ne more biti zrcalen izraz. Naša spodnja rešitev pri praznem nizu izpiše, da je zrcalen.)

Oglejmo si implementacijo te rešitve v C++. Po nizu se bomo hkrati premikali z dvema iteratorjema: iL gre od leve proti desni, iD pa od desne proti levi; zanka se konča, ko se oba iteratorja srečata (na sredini niza), lahko pa jo prekinemo tudi prej, če se izkaže, da ni niz niti palindrom niti zrcalni izraz.

Preverjanje, ali sta si trenutna znaka zrcalna, si lahko malo poenostavimo tako, da najprej preverimo, ali sta enaka; če sta enaka, potem tako ali tako vemo, da si nista zrcalna; če pa nista enaka, potem sta si zrcalna natanko tedaj, če oba pripadata istemu tipu oklepajev, torej če sta oba okrogla ali pa oba oglata. (O tem se lahko prepričamo takole: če znaka pripadata istemu tipu oklepajev in si nista enaka, potem mora biti eden od njiju predklepaj, drugi pa zaklepaj, to pa pomeni, da sta si zrcalna; če pa znaka ne pripadata istemu tipu oklepajev — torej če je eden okrogel, eden pa oglat —, potem si gotovo nista zrcalna, saj se pri zrcaljenju tip oklepaja ohranja.)

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

int main()
{
  string s; cin >> s; // Preberimo vhodni niz.
  // V zanki primerjajmo znake z obeh koncev niza.
  bool palindrom = true, zrcalen = true;
  for (auto iL = s.begin(), iD = s.end(); iL < iD && (palindrom || zrcalen); )
  {
    char cL = *iL++, cD = *--iD;
```

```
    // Če sta levi in desni znak enaka, niz gotovo ni zrcalen, lahko pa je palindrom.
    if (cL == cD) zrcalen = false;

    // Če sta si levi in desni znak zrcalna, niz gotovo ni palindrom, lahko pa je zrcalen.
    // Ker tu že vemo, da si levi in desni znak nista enaka, je za njuno zrcalnost dovolj že,
    // če oba pripadata istemu tipu oklepajev, torej če sta oba okrogla ali oba oglata.
    else if ((cL == '(' || cL == ')') == (cD == '(' || cD == ')')) palindrom = false;

    // Sicer ni niz nič od obojega.
    else zrcalen = false, palindrom = false;
  }
  // Izpišimo rezultat.
  cout << (zrcalen ? "zrcalen izraz" : palindrom ? "palindrom" : "navaden") << endl;
  return 0;
}
```

Zapišimo to rešitev še v pythonu:

```
s = input() # Preberimo vhodni niz.

# V zanki primerjajmo znake z obeh koncev niza.
i = 0; j = len(s) - 1; palindrom = True; zrcalen = True
while i <= j and (palindrom or zrcalen):
  ci = s[i]; cj = s[j]; i += 1; j -= 1

  # Če sta levi in desni znak enaka, niz gotovo ni zrcalen.
  if ci == cj: zrcalen = False

  # Če sta si levi in desni znak zrcalna, niz gotovo ni palindrom.
  elif (ci == '(' || ci == ')') == (cj == '(' || cj == ')'): palindrom = False

  # Sicer ni niz niti palindrom niti zrcalen.
  else: zrcalen = False; palindrom = False

# Izpišimo rezultat.
print("zrcalen izraz" if zrcalen else "palindrom" if palindrom else "navaden")
```

Nalogo lahko rešimo tudi tako, da si pripravimo obrnjeno različico niza; če je enaka prvotnemu nizu, lahko zaključimo, da gre za palindrom. Če ni palindrom, pa zamenjajmo vsak znak v obrnjenem nizu z njegovo prezrcaljeno različico (torej predklepaj z zaklepajem enakega tipa in obratno); če je niz zdaj enak prvotnemu, je šlo za zrcalni izraz.

```
s = input()              # Preberimo vhodni niz.
r = s[::-1]              # Obrnimo niz z desne proti levi.

if r == s:               # Če sta prvotni in obrnjeni niz enaka,
  print("palindrom")     # gre za palindrom.

else:
  # Zamenjajmo v obrnjenem nizu vsak znak z njegovo
  # zrcalno sliko; tako dobimo zrcalno sliko prvotnega niza.
  r = r.replace('(', 'x').replace(')', '(').replace('x', ')')
  r = r.replace('[', 'x').replace(']', '[').replace('x', ']')

  # Če sta prvotni in prezrcaljeni niz enaka, gre za zrcalen izraz.
  print("zrcalen izraz" if r == s else "navaden")
```

(Ta rešitev se od prejšnje sicer razlikuje po tem, da da pri praznem nizu zdaj izpišemo, da je palindrom, prej pa smo izpisali, da je zrcalni izraz.) Pri zamenjavi

znakov smo se oprli na predpostavko, da v vhodnem nizu ni drugih znakov kot oklepajev in zaklepajev; zato lahko uporabimo na primer znak `'x'` kot pomožen znak pri zamenjavi (najprej zamenjamo predklepaje z `'x'`, nato zaklepaje s predklepaji in končno znake `'x'` z zaklepaji).

4. Slika iz kock

Ker naloga zagotavlja, da bo med vhodnimi podatki gotovo neka kocka iz zadnjega stolpca, lahko širino slike določimo preprosto tako, da si zapomnimo največjo številko stolpca x_i, ki nastopa v vhodnih podatkih; podobno nam največja številka vrstice y_i v vhodnih podatkih pove višino slike. Da bomo lako izpisali manjkajoče kocke, pa si je koristno med branjem vhodnih podatkov nekako označiti, katere kocke smo prebrali; ko končamo z branjem vhodnih podatkov, lahko potem z dvema gnezdenima zankama pregledamo vse položaje (x, y) na naši sliki, pri vsakem od njih preverimo, ali smo tisto kocko že prebrali, in če je nismo, jo zdaj izpišemo kot manjkajočo. Naloga zagotavlja, da bodo številke vrstic in stolpcev le od 1 do 5000, zato lahko za označevanje tega, katere kocke smo že prebrali, uporabimo kar tabelo 5000×5000 logičnih vrednosti — ali pa celo 5001×5001 vrednosti, da bomo lahko za indekse uporabljali števila od 1 do 5000 namesto od 0 do 4999. Oglejmo si primer implementacije takšne rešitve v C++:

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

bool znana[5001][5001] = {};              // Tu bomo označevali že prebrane kocke.

int main()
{
  int w = 0, h = 0;                       // Širina in višina slike.
  int n; cin >> n;                        // Preberimo število kock.

  // Preberimo vseh n kock in si jih označimo kot znane.
  while (n-- > 0) {
    int xi, yi, bi; cin >> xi >> yi >> bi;  // Preberimo naslednjo kocko.
    znana[yi][xi] = true;                   // Označimo jo kot znano.

    // V w in h si zapomnimo največjo doslej opaženo številko stolpca oz. vrstice.
    if (xi > w) w = xi;
    if (yi > h) h = yi; }

  cout << w << " " << h << endl;          // Izpišimo velikost slike.

  // Pojdimo po vseh kockah slike in izpišimo tiste, ki jih še ne poznamo.
  for (int y = 1; y <= h; ++y) for (int x = 1; x <= w; ++x)
    if (! znana[y][x]) cout << x << " " << y << endl;

  return 0;
}
```

Namesto tabele **bool**ov bi lahko uporabili tudi razreda bitset ali vector<**bool**> iz C++ove standardne knjižnice, s čimer bi prihranili nekaj pomnilnika (če sta omenjena razreda v knjižnici implementirana tako, da v njiju posamezni element porabi le en bit pomnilnika).

Zapišimo podobno rešitev še v pythonu:

```
znana = [[False] * 5001 for y in range(5001)]   # Tu bomo označevali že prebrane kocke.
w = 0; h = 0                                    # Širina in višina slike.
n = int(input())                                # Preberimo število kock.

# Preberimo vseh n kock in si jih označimo kot znane.
for i in range(n):
    xi, yi, bi = map(int, input().split())      # Preberimo naslednjo kocko.
    znana[yi][xi] = True                        # Označimo jo kot znano.
    # V w in h si zapomnimo največjo doslej opaženo številko stolpca oz. vrstice.
    w = max(w, xi); h = max(h, yi)

print(w, h)                                     # Izpišimo velikost slike.

# Pojdimo po vseh kockah slike in izpišimo tiste, ki jih še ne poznamo.
for y in range(1, h + 1):
    for x in range(1, w + 1):
        if not znana[y][x]: print(x, y)
```

Tudi tu bi lahko prihranili nekaj pomnilnika, če bi za `znana` namesto pythonovega seznama (`list`) uporabili `bytearray`.

Še ena možnost je, da namesto tabele 5000×5000 uporabimo razpršeno tabelo oz. množico, v katero shranjujemo že prebrane kocke. To bi lahko prihranilo nekaj pomnilnika v primerih, ko je n majhen v primerjavi z velikostjo slike. V prejšnji rešitvi bi morali spremeniti le tri vrstice:

```
znana = set()                                   # Tu bomo označevali že prebrane kocke.
  ⋮
  znana.add((xi, yi))                           # Označimo jo kot znano.
  ⋮
    if (x, y) not in znana: print(x, y)
```

Naloge so sestavili: zlogi skibidi, grošev pest, podaje, pribor — Bor Brudar; SMS — Bor Grošelj Simić; kocke — Ella Potisek; nalepke — Jaka Velkaverh; čez cesto, Metkine zvezdice, navijači, palindromski oklepaji, slika iz kock — Patrik Žnidaršič.