15. tekmovanje ACM v znanju računalništva za srednješolce

28. marca 2020

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 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("%d + %d = %d" % (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("%d. vrstica: \"%s\"" % (i, s))
print("%d vrstic, %d znakov." % (i, d))
```

```
# 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("Skupaj %d znakov." % i)
```

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.");
   }
}
```

15. tekmovanje ACM v znanju računalništva za srednješolce

28. marca 2020

NALOGE ZA PRVO SKUPINO

Naloge rešuj samostojno; ne sprašuj drugih ljudi za nasvete ali pomoč pri reševanju (niti v živo niti prek interneta ali kako drugače), ne kopiraj v svoje odgovore tuje izvorne kode in podobno. Tekmovalna komisija si pridržuje pravico, da tekmovalce diskvalificira, če bi se kasneje izkazalo, da nalog niso reševali sami. Internet lahko uporabljaš, če ni v nasprotju s prejšnjimi omejitvami (npr. za branje dokumentacije), vendar za reševanje nalog ni nujno potreben. Tvoje odgovore bomo pregledali in ocenili ročno, 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 (bo)ste dobili po elektronski pošti. Pri oddaji preko računalnika rešitev natipkaš neposredno v brskalniku. 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“. Gumb „Shrani in zapri“ uporabiš, ko si bodisi zadovoljen z rešitvijo ter si zaključil nalogo, ali ko želiš začasno prekiniti pisanje rešitve naloge ter se lotiti druge naloge. Po pritisku na ta gumb se vpisana rešitev shrani in te vrne v glavni menu. (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), izjemoma pa tudi po elektronski pošti na `rtk-info@ijs.si`.

Če imaš pri oddaji odgovorov prek spletnega strežnika kakšne težave, lahko izjemoma pošlješ svoje odgovore po elektronski pošti na `rtk-info@ijs.si`, vendar nas morajo doseči pred koncem tekmovanja; odgovorov, prejetih po koncu tekmovanja, ne bomo upoštevali.

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.

Rešitve bodo objavljene na `http://rtk.ijs.si/`. Predvidoma nekaj dni po tekmovanju bodo tam objavljeni tudi rezultati.

1. Vesoljske vsote

Preučevalec vesoljske matematike, Tim, je na eni izmed svojih odprav v vesolje odkril skrivnosten tuj zapis vsot. Niz „`*---**-*-----*`“ predstavlja vesoljski zapis vsote $1+4+4+5+10$. Po podrobnem pregledu nekaj zapisov je ugotovil, da za zapisovanje vsot v vesolju obstajajo naslednja pravila:

- Na začetku postavimo trenutno število na 1.
- Znak „`*`“ doda trenutno število v zapis vsote (na konec).
- Znak „`-`“ poveča trenutno število za 1.

Tim ni najbolj spreten pri programiranju, zato te prosi, da mu **napišeš program** (ali podprogram oz. funkcijo), ki vesoljski zapis vsote pretvori v človeku berljiv račun. Na standardni vhod bo tvoj program dobil niz znakov „`*`“ in „`-`“, ki naj ga pretvori v človeku berljiv račun (na koncu naj doda tudi enačaj in končno vrednost vsote) in izpiše na standardni izhod (ali pa v datoteko `vsota.txt`, karkoli ti je lažje). Predpostaviš lahko, da vsebuje vhodni niz vsaj eno zvezdico „`*`“. Zgornji niz naj tvoj program izpiše kot „`1 + 4 + 4 + 5 + 10 = 24`“.

Tim je pripravil tudi dva primera, da mu boš lažje pomagal.

Primer 1:

Vhod: `*---**-*-----*`
Izhod: `1 + 4 + 4 + 5 + 10 = 24`

Naslednja tabela kaže vrednost trenutnega števila po vsakem prebranem znaku:

Znak		*	-	-	-	*	*	-	*	-	-	-	-	-	*
Število	1	1	2	3	4	4	4	5	5	6	7	8	9	10	10

Primer 2:

Vhod: `-----*-*`
Izhod: `6 + 7 = 13`

Naslednja tabela kaže vrednost trenutnega števila po vsakem prebranem znaku:

Znak		-	-	-	-	-	*	-	*
Število	1	2	3	4	5	6	6	7	7

2. Ključ

Ključ za običajno cilindrično ključavnico ima šest zarez, globina vsake se mora ujemati z dolžino istoležnega zatiča v ključavnici, da se ključavnica lahko odklene. Zareze niso poljubno globoke, ampak proizvajalec predpisuje določene pogoje, ki jih morajo globine zarez izpolnjevati, da lahko nek tip ključavnice deluje pravilno in zanesljivo ter da se lahko ključ vanjo vstavi in izvleče brez zatikanja. Možnih je deset različnih globin, označenih s številko med 0 in 9.

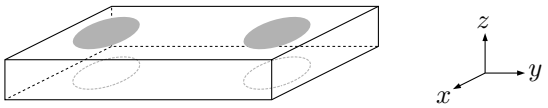
1. dovoljene so le globine med 1 in 8;
2. globini sosednjih dveh zarez se lahko razlikujeta za največ 5 (globina 2 na primer ne sme biti sosednja globini 8; to pravilo zagotavlja, da se ključ ne more zatakniti pri vstavljanju ali odstranjevanju);
3. dve sosednji zarezi ne smeta imeti enake globine;
4. nobena globina zareze se ne sme pojaviti več kot dvakrat.

Napiši podprogram (funkcijo), ki bo kot argument prejel celoštevilčno kodo ključa (na primer: 597969). Desetiške števke tega števila predstavljajo šest globin zarez. Lahko je prvih nekaj števk enakih 0 (na primer: število 123 predstavlja kodo 000123). Za vsako od naštetih štirih pravil naj podprogram izpiše, ali je pravilo izpolnjeno ali ne.

Primer: 597969 ustreza drugemu in tretjemu pogoju, ne pa prvemu (ker v njem niso samo števke od 1 do 8) in četrtemu (ker se števka 9 pojavlja trikrat).

3. Obračanje jogija

Imamo jogi v obliki kvadra. Odvisno od tega, kako ga obrnemo, se lahko naša glava znajde na enem od štirih možnih mest, kot kaže leva slika spodaj:



Sivo pobarvani elipsi kažeta dva možna položaja glave na eni strani jogija, črtkani elipsi pa še dva možna položaja glave na drugi strani jogija.

Spati želimo tako, da imamo glavo na najmanj obrabljenem delu, zato smo pripravljeni jogi občasno obrniti — zavrtimo ga za 180 stopinj okrog ene od osi x, y ali z (desna slika zgoraj kaže, kam je usmerjena katera os), tako da bo potem naša glava na kakšnem od ostalih treh možnih mest na jogiju.

Napiši funkcijo ObrniJogi(n), ki jo bo uporabnik poklical, ko bo pripravljen obrniti jogi, ona pa mu mora primerno svetovati, po kateri osi naj obrne jogi za 180 stopinj, da bo imel glavo na najmanj obrabljenem delu (torej na tistem, na katerem je doslej spal najmanj dni). Funkcija naj vrne enega od znakov `'x'`, `'y'`, `'z'`; če pa je bolje, da uporabnik trenutno jogija sploh ne obrača, naj tvoja funkcija vrne znak `'n'`. Uporabnik potem obrne jogi v skladu z rezultatom, ki ga vrne tvoja funkcija, in vse odtlej do naslednjega klica vsak dan spi na njem v njegovem sedanjem položaju (jogija torej nikoli ne obrača na lastno pest).

Kot parameter `n` bo tvoja funkcija dobila število dni od zadnjega klica (toliko dni je uporabnik spal na dosedanjem mestu).

Predpostavi, da uporabnik tvojo funkcijo prvič pokliče z `n = 0` in da dotlej na jogiju ni še nikoli spal. Poleg funkcije ObrniJogi lahko deklariraš tudi poljubne globalne spremenljivke (oz. spremenljivke zunaj svoje funkcije) in jih po želji inicializiraš.

4. Zobna ščetka

Električno zobno ščetko poganja elektromotor, kot uporabniški vmesnik pa služi tipka; ta je lahko pritisnjena ali spuščena. Delovanje motorja upravlja preprost računalnik, ki lahko odčitava trenutno stanje tipke in lahko vklaplja ali izklaplja motor. Izklopljeno ščetko spravimo v pogon s pritiskom na tipko. Trajanje pritiska na tipko ne vpliva na delovanje, važen je le trenutek začetka pritiska tipke. Da uporabnik ne pretirava s čiščenjem zob, se mora ščetka samodejno izklopiti po 120 sekundah od zadnjega vklopa, lahko pa jo uporabnik izklopi že pred iztekom tega časa s (ponovnim) pritiskom na tipko.

Napiši program, ki bo upravljal z motorjem zobne ščetke tako, kot je zgoraj predpisano. Na razpolago so naslednje funkcije:

- za odčitavanje stanja tipke:

  Tipka() — funkcija vrne **true**, če je tipka pritisnjena, sicer **false**. (Funkcija ne čaka na pritisk tipke, ampak se vrne takoj in sporoči trenutno stanje tipke. Če uporabnik dlje časa drži tipko pritisnjeno, funkcija v tem času ob vsakem klicu vrne **true**.)

- za vklop ali izklop motorja:

  Motor(vklop) — če ima parameter vklop vrednost **true**, bo funkcija vklopila motor, če ima vrednost **false**, pa ga bo izklopila.

- štoparica, ki meri čas v sekundah:

  PozeniUro() — postavi čas na 0 in požene štoparico;
  UstaviUro() — ustavi štoparico;
  OdcitajUro() — vrne čas štoparice v sekundah kot celo število (tipa **int** oz. integer).

(Kdor piše v pythonu, naj si namesto **true** in **false** misli True in False.)

5. Plonkanje

Zaradi pojava virusa so morali organizatorji nekega tekmovanja iz znanja izvesti le-to prek interneta, kjer pa tekmovalna komisija ni mogla nadzirati, če so si tekmovalci med seboj kaj pomagali („plonkali“).

Po natančni analizi vseh oddanih nalog več sto tekmovalcev je tekmovalni komisiji uspelo (seveda s pomočjo računalnika) natančno rekonstruirati, kdo je plonkal od koga.

Podatki o plonkanju so (zaradi varstva osebnih podatkov) anonimizirani in predstavljeni v tabeli z dvema stolpcema, kjer prva številka pomeni številko tekmovalca, ki je bil *pomočnik*, druga pa številko *prepisovalca*, torej tekmovalca, ki je prepisoval (plonkal). Tisti, ki je plonkal, je seveda lahko bil ob neki drugi priložnosti pomočnik in je pomagal novemu prepisovalcu in tako naprej. Pomočnik je lahko pomagal več prepisovalcem (v prvem stolpcu bodo lahko tudi enake številke), medtem ko je prepisovalec vedno lahko plonkal samo od enega pomočnika (v drugem stolpcu bodo same različne številke). Primer:

pomočnik	prepisovalec (plonkar)
15	18
41	62
15	29
47	50
29	47
15	41
33	21
91	55
41	37
21	12
12	72
12	33

Iz zgornje tabele ugotovimo na primer, da je tekmovalec 41 prepisoval od tekmovalca 15 in da je tekmovalec 41 pomagal tekmovalcu 62 in tekmovalcu 37. Tekmovalec 15 pa je pomagal tudi tekmovalcu 29 in tekmovalcu 18.

Opiši postopek, ki bo iz tako podanih podatkov ugotovil:

(a) kateri so bili tisti tekmovalci, ki niso plonkali od nikogar, ampak so bili izključno pomočniki — imenujmo jih *izvirni tekmovalci*;

(b) kateri so bili *plonkarji prvega reda*, torej tekmovalci, ki so plonkali od izvirnih tekmovalcev;

(c) kateri so bili *plonkarji drugega reda*, torej tekmovalci, ki so plonkali od nekoga, ki je sam plonkal od izvirnega tekmovalca.

V zgornjem primeru sta izvirna tekmovalca 15 in 91, plonkarji prvega reda so 18, 41, 29 in 55, plonkarji drugega reda pa so 37, 62 in 47.

15. tekmovanje ACM v znanju računalništva za srednješolce

28. marca 2020

NALOGE ZA DRUGO SKUPINO

Naloge rešuj samostojno; ne sprašuj drugih ljudi za nasvete ali pomoč pri reševanju (niti v živo niti prek interneta ali kako drugače), ne kopiraj v svoje odgovore tuje izvorne kode in podobno. Tekmovalna komisija si pridržuje pravico, da tekmovalce diskvalificira, če bi se kasneje izkazalo, da nalog niso reševali sami. Internet lahko uporabljaš, če ni v nasprotju s prejšnjimi omejitvami (npr. za branje dokumentacije), vendar za reševanje nalog ni nujno potreben. Tvoje odgovore bomo pregledali in ocenili ročno, 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 (bo)ste dobili po elektronski pošti. Pri oddaji preko računalnika rešitev natipkaš neposredno v brskalniku. 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“. Gumb „Shrani in zapri“ uporabiš, ko si bodisi zadovoljen z rešitvijo ter si zaključil nalogo, ali ko želiš začasno prekiniti pisanje rešitve naloge ter se lotiti druge naloge. Po pritisku na ta gumb se vpisana rešitev shrani in te vrne v glavni menu. (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), izjemoma pa tudi po elektronski pošti na `rtk-info@ijs.si`.

Če imaš pri oddaji odgovorov prek spletnega strežnika kakšne težave, lahko izjemoma pošlješ svoje odgovore po elektronski pošti na `rtk-info@ijs.si`, vendar nas morajo doseči pred koncem tekmovanja; odgovorov, prejetih po koncu tekmovanja, ne bomo upoštevali.

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.

Rešitve bodo objavljene na `http://rtk.ijs.si/`. Predvidoma nekaj dni po tekmovanju bodo tam objavljeni tudi rezultati.

1. Metanje na koš

Profesionalni košarkarji se morajo med treningom vaditi tudi v metanju prostih metov na koš, saj so prosti meti pomemben del košarkaških tekem in je važno, da so pri tem čim bolj zanesljivi.

Organiziramo torej tekmovanje v metanju prostih metov na koš, kjer se tekmovalci preizkušajo, kdo ima najboljše živce in bo največkrat vrgel na koš, ne da bi enkrat samkrat zgrešil koš.

Pravila so takšna: Vsak tekmovalec bo žogo na koš vrgel velikokrat, recimo tisočkrat (seveda lahko vmes tudi počiva, spije vodo, ali kaj prigrizne `;-)`). Zmagovalec tekmovanja bo tisti, ki bo naredil najdaljši neprekinjeni niz zadetkov v koš.

Če pa bo imelo več tekmovalcev enak najdaljši neprekinjeni niz zadetkov, potem bo zmagovalec tisti od njih, ki ima najdaljši niz zadetkov, v katerem je en sam met mimo koša.

In če bo tudi pri tem pogoju še vedno več tekmovalcev imelo enako dolg niz zadetkov z enim zgrešenim metom, potem bo zmagovalec tisti, ki bo imel najdaljši niz zadetkov z dvema zgrešenima metoma. In tako dalje, s tremi, štirimi, petimi zgrešenimi meti v nizu, dokler ne bo število zgrešenih metov v zaporednem nizu enako številu vseh zgrešenih metov tekmovalca. Če bo tudi takrat več tekmovalcev imelo enak rezultat (pri tisoč metih je sicer zelo malo verjetno, da bi do tega prišlo), bo zmagovalca pač določila tekmovalna komisija z žrebom.

Podatki za enega tekmovalca so predstavljeni z nizom enic in ničel (enica za zadetek v koš, ničla za met mimo koša), na primer takole (35 metov, 26 zadetkov, 9 mimo koša):

```
11101111001101111101111001011111110
```

Opiši postopek ali napiši podprogram (funkcijo), ki za dani niz in celo število k izračuna dolžino najdaljšega takega niza zadetkov, med katerimi je največ k metov mimo koša. (Če tega ne znaš rešiti v splošnem, reši nalogo vsaj za primere, ko je $k = 0$ ali $k = 1$, in boš dobil delne točke.)

Primer: za gornji niz 35 metov bi pri $k = 0$ dobili rezultat 7, pri $k = 1$ rezultat 9 (zaporedje petih zadetkov, nato en met mimo koša in nato še zaporedje štirih zadetkov), pri $k = 3$ pa rezultat 12.

2. Ne odlašaj na jutri, kar lahko storiš pojutrišnjem

Učenci v šoli imajo n obveznosti, ki jih morajo izpolniti. Pri tem jim i-ta obveznost vzame d_i dni in jo morajo opraviti najkasneje na dan k_i (mislimo si, da so dnevi oštevilčeni z naravnimi števili od začetka šolskega leta). V posameznem dnevu se lahko ukvarjajo samo z eno obveznostjo, lahko pa počivajo in se ne ukvarjajo z nobeno. Ko se enkrat lotijo neke obveznosti, jo morajo potem opraviti v neprekinjenem sklopu d_i zaporednih dni (torej jim ta obveznost vzame dan, ko so začeli z njo, in še naslednjih $d_i - 1$ dni). Če je na primer $d_i = 3$, to pomeni, da se morajo začeti z i-to obveznostjo ukvarjati najkasneje na dan $k_i - 2$.

Kot tipični učenci si seveda želijo vse te obveznosti opraviti čim kasneje. Torej, če lahko nek dan počivajo in se obveznosti lotijo jutri in še zmeraj opravijo vse obveznosti pravočasno, potem danes seveda raje počivajo.

Natančneje povedano, dva možna razporeda tega, kdaj opravijo kakšno obveznost, primerjamo takole: poiščemo najzgodnejši dan, na katerega pri enem razporedu počivajo, pri drugem pa delajo; če takega dne ni, štejemo razporeda za enakovredna in sta nam oba enako dobra; sicer pa nam je boljši tisti razpored, pri katerem na tisti dan počivajo.

Napiši program ali podprogram (funkcijo), ki kot vhodne podatke dobi podatke o obveznostih (števila $n, k_1, d_1, k_2, d_2, \ldots, k_n, d_n$) in izračuna najboljši možni razpored (za vsako obveznost i naj torej ugotovi, na kateri dan z_i se morajo začeti ukvarjati z njo), ali pa ugotovi, da za te vhodne podatke sploh ni nobenega veljavnega razporeda. Če je možnih več enako dobrih razporedov, je vseeno, katerega poiščeš. Podrobnosti glede oblike vhodnih in izhodnih podatkov si izberi sam in jih tudi opiši. Tvoja rešitev naj bo čim učinkovitejša, tako da bo uporabna tudi za večje vhodne primere (recimo, da gre lahko n do 10^6, datum k_i pa do 10^9).

3. Lenoba

V službi želimo preživeti čim manj časa, ne da bi to kdorkoli opazil. Za vse sodelavce natanko vemo čas prihoda in odhoda (vsakdo pride in odide zgolj enkrat v dnevu; vsi podatki so znotraj enega dneva, nihče ne ostane v službi čez polnoč). **Opiši postopek**, ki izračuna, kdaj moramo priti v službo in koliko časa moramo tam preživeti, da nobena oseba ne bo prisotna takrat, ko pridemo, in še vedno prisotna takrat, ko odidemo. Edini dodatni pogoj je, da želimo priti v službo pred dvanajsto in oditi po dvanajsti (ker je točno ob dvanajstih kosilo).

Kot vhodne podatke tvoj postopek dobi število sodelavcev in za vsakega sodelavca čas njegovega prihoda in odhoda. Vsi časi se merijo v nanosekundah od polnoči, tako da so to sicer nenegativna cela števila, vendar so lahko precej velika. (Ena sekunda ima 1 000 000 000 nanosekund.) Sodelavec, ki pride ali odide ob istem času kot mi, nas vidi priti ali oditi — če se hočemo temu izogniti, moramo priti vsaj eno nanosekundo pred njim ali oditi vsaj eno nanosekundo za njim. Podrobnosti glede predstavitve vhodnih podatkov si izberi sam in jih v svoji rešitvi tudi opiši.

4. Semafor

Semafor na prehodu za pešce ima dvomestni prikazovalnik sekund do spremembe luči. Prikazuje lahko torej poljubno celo število od 0 do 99, lahko pa je tudi ugasnjen. **Napiši podprogram** oz. funkcijo VsakoSekundo(n), ki jo bo sistem poklical enkrat na sekundo, da mu bo pomagala upravljati s prikazovalnikom na semaforju. Prek parametra n ti sistem pove, kaj se dogaja z lučjo semaforja: če se je v zadnji sekundi stanje luči spremenilo (iz zelene v rdečo ali obratno), ti parameter n pove, koliko sekund bo luč semaforja v svojem novem stanju; če pa se v zadnji sekundi stanje luči ni spremenilo, boš dobil $n = -1$.

Na voljo imaš funkcijo Prikazi(n), ki jo lahko pokličeš, da na prikazovalniku prikažeš število n. Število n mora biti od 0 do 99, lahko pa je -1, če hočeš, da naj bo prikazovalnik prazen (ne prikazuje nobenega števila, niti ničle).

Deklariraš lahko tudi globalne spremenljivke in jih po svoji želji inicializiraš. Predpostavi, da bo vrednost parametra n pri prvem klicu funkcije VsakoSekundo gotovo večja od 0.

Če čas, ko bi se morala luč spremeniti, mine, sistem pa te še ni obvestil o spremembi stanja luči (in trajanju novega stanja), moraš poskrbeti, da bo prikazovalnik prazen, dokler te sistem ne obvesti o spremembi stanja luči.

Prikazovalnik na semaforju je le dvomesten, stanje luči pa včasih traja več kot 99 sekund. Zato naj tvoj podprogram poskrbi, da če je do spremembe stanja luči več kot 99 sekund, naj se prikazuje število 99, vendar naj utripa (eno sekundo gori, eno sekundo je ugasnjena).

5. Prelom besedila

Dano imamo besedilo, dolgo z znakov, ki ga želimo prikazati v okencu, ki ima prostora za w znakov na vrstico. Ko besedilo pišemo v okence, lahko vrstico prelomimo le na presledku pred začetkom nove besede. Presledki ostanejo na prejšnji vrstici in lahko gledajo preko roba okna, nova vrstica pa se začne s prvim naslednjim znakom, ki ni presledek.

Za vsako širino w od 1 do z izračunaj, koliko vrstic bo besedilo imelo, če ga želimo napisati v okence širine w. Besedilo bo vsebovalo le črke, številke, ločila in presledke. Lahko predpostaviš, da drugih znakov za prazen prostor, kot na primer tabulatorjev ali znakov za novo vrstico, ne bo. Besedilo se tudi ne bo začelo s presledkom.

Napiši podprogram (funkcijo), ki sprejme besedilo kot niz znakov dolžine z in vrne ali izpiše seznam z števil, kjer števila po vrsti predstavljajo, koliko vrstic bo besedilo imelo, če ga želimo napisati v okence širine $1, 2, 3, 4, \ldots, z$. Če besedila v okence neke širine ni mogoče spraviti, ne da kakšna beseda gledala prek meja, na tisto mesto napiši -1.

Primer. Recimo, da dobimo takšen vhodni niz (spodaj je napisan v dveh vrsticah, vendar si moramo obe skupaj predstavljati kot en sam niz, brez kakšnih vmesnih znakov za konec vrstice ali česa podobnega; presledki so predstavljeni s simbolom ␣, da se jih bolje vidi):

```
Na␣začetku␣je␣bilo␣ustvarjeno␣vesolje.␣␣To␣je␣povzročilo␣
mnogo␣hude␣krvi␣in␣na␣splošno␣velja␣za␣zelo␣slabo␣potezo.
```

Pravilen izhodni seznam za ta niz je:

$-1, -1, -1, -1, -1, -1, -1, -1, -1, 12, 12, 12, 10, 10, 8, 8, 8, 7, 6, 6, 6,$
$6, 6, 5, 5, 5, 5, 5, 4, 4, 4, 4, 4, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,$
$3, 3, 3, 3, 3, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,$
$2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,$
$2, 2, 1.$

Na primer, če je $w = 19$, spravimo besedilo v 6 vrstic (zato je 19. element v gornjem seznamu enak 6):

```
Na␣začetku␣je␣bilo␣
ustvarjeno␣vesolje.␣␣
To␣je␣povzročilo␣
mnogo␣hude␣krvi␣in␣
na␣splošno␣velja␣za␣
zelo␣slabo␣potezo.
```

(Primer očitno povzet po: Douglas Adams, *Restavracija ob koncu Vesolja*, *Štoparski vodnik po galaksiji*, prevod: Alojz Kodre.)

15. tekmovanje ACM v znanju računalništva za srednješolce

28. marca 2020

PRAVILA TEKMOVANJA ZA TRETJO SKUPINO

Naloge rešuj samostojno; ne sprašuj drugih ljudi za nasvete ali pomoč pri reševanju (niti v živo niti prek interneta ali kako drugače), ne kopiraj v svoje odgovore tuje izvorne kode in podobno. Tekmovalna komisija si pridržuje pravico, da tekmovalce diskvalificira, če bi se kasneje izkazalo, da nalog niso reševali sami. Internet lahko uporabljaš, če ni v nasprotju s prejšnjimi omejitvami (npr. za branje dokumentacije). V rešitvah lahko uporabljaš manjše fragmente izvorne kode, ki si jih napisal sam že pred tekmovanjem.

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 ali python, mi pa jih bomo preverili s prevajalniki FreePascal, GNUjevima `gcc` in `g++` 7.4.0 (ta verzija podpira C++17), prevajalnikom za javo iz JDK 8, s prevajalnikom Mono 4.6 za C# in z interpreterjema za python 2.7 in 3.6.

Na spletni strani `https://putka-rtk.acm.si/contests/rtk-2020-3/` 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/test-sistema/`. Uporabniško ime in geslo za Putko boš dobil po elektronski pošti. Med tekmovanjem lahko vprašanja za tekmovalno komisijo postavljaš prek foruma na Putki (povezava „Diskusija“ na dnu besedila posamezne naloge), izjemoma pa tudi po elektronski pošti na `rtk-info@ijs.si`.

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/help/programming/`. Tvoji programi naj uporabljajo le standardne knjižnice svojega programskega jezika in naj ne delajo z datotekami na disku.

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 vsakem od njih dobi vse točke, če je izpisal pravilen odgovor, sicer pa 0 točk (izjema je 3. naloga, kjer je možno tudi delno točkovanje). Pri prvi in tretji nalogi je testnih primerov po 20 in vsak je vreden po 5 točk, pri četrti in peti nalogi je testnih primerov po 10 in vsak je vreden po 10 točk, pri drugi nalogi pa je testnih primerov 6, število točk za posamezni primer pa je navedeno v besedilu naloge.

Nato se točke po vseh testnih primerih seštejejo v skupno število točk tega programa. Če si oddal N programov za to nalogo in je najboljši med njimi dobil M (od 100) točk,

dobiš pri tej nalogi $\max\{0, M - 3(N-1)\}$ točk. Z drugimi besedami: za vsako oddajo (razen prve) pri tej nalogi se ti odbijejo tri točke. Pri tem pa ti nobena naloga ne more prinesti negativnega števila točk. Če nisi pri nalogi oddal nobenega programa, ti ne prinese nobenih točk. Če se poslana izvorna koda ne prevede uspešno, to ne šteje kot oddaja.

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

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

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));
  }
}
```

15. tekmovanje ACM v znanju računalništva za srednješolce

28. marca 2020

NALOGE ZA TRETJO SKUPINO

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

1. Vaje v slogu

Avantgardni pisatelj Polde ne uporablja presledkov, ločil, velikih začetnic in podobnih stvari. Njegovi spisi zato niso nič drugega kot dolgi nizi samih malih črk. Iznašel je tudi novo retorično figuro, ki jo je z veliko skromnostjo poimenoval *Poldetova trojica*. Tvorijo jo tri pojavitve enakega podniza v dolgem nizu, ki predstavlja njegov spis, pri čemer se te tri pojavitve med seboj ne smejo prekrivati, smejo pa biti med njimi tudi kakšne črke, ki ne pripadajo nobeni od teh treh pojavitev.

Poldetovo trojico lahko na kratko opišemo s četverico števil (i, j, k, d), pri čemer d pove dolžino uporabljenega podniza, števila i, j in k pa so indeksi prvega znaka vsake od njegovih treh pojavitev, ki tvorijo to trojico (znake niza, ki predstavlja celoten spis, si mislimo oštevilčene od 1 do n, pri čemer je n dolžina niza). Pojavitve naštejmo vedno od leve proti desni, tako da bo $i < j < k$.

Oglejmo si nekaj primerov. V nizu `ababaccabab` se med drugim pojavljajo naslednje Poldetove trojice:

Podniz	Pojavitve	Opis (i, j, k, d)
`ab`	**ab**a**ab**acca**b**ab	$(1, 3, 8, 2)$
`ab`	**ab**a**ab**acca**b**ab	$(1, 3, 10, 2)$
`ba`	a**ba**a**ba**cca**b**ab	$(2, 4, 9, 2)$
`a`	**a**b**a**b**a**ccabab	$(1, 3, 5, 1)$
`a`	ab**a**b**a**cc**a**bab	$(3, 5, 8, 1)$
`a`	abab**a**cc**a**b**a**b	$(5, 8, 10, 1)$
`a`	ab**a**bacc**a**b**a**b	$(3, 7, 10, 1)$
`a`	**a**baba**a**ccab**a**b	$(1, 5, 10, 1)$
`b`	a**b**a**b**acca**b**ab	$(2, 4, 9, 1)$

Poleg zgornjih je v istem nizu še več drugih Poldetovih trojic. V tem nizu se trikrat pojavlja tudi podniz `aba`, vendar pa njegove tri pojavitve ne tvorijo Poldetove trojice, ker se prvi dve pojavitvi tega podniza prekrivata.

Polde ima rad trojice, ki jih tvorijo čim daljši podnizi. **Napiši program**, ki mu jih bo pomagal odkrivati. Med vsemi Poldetovimi trojicami poišči tisto z največjo dolžino podniza d oz. preštej, koliko je trojic s tem d.

Vhodni podatki: v prvi vrstici je celo število n (veljalo bo $1 \leq n \leq 10^5$). V drugi vrstici je niz s; dolg je n znakov in vsi ti znaki so male črke angleške abecede.

Pri 70 % testnih primerov bo $n \leq 1000$.

Izhodni podatki: v prvo vrstico izpiši števila i, j, k in d, ločena s po enim presledkom. Zanje mora veljati $1 \leq i$, $i + d \leq j$, $j + d \leq k$, $k + d \leq n + 1$ in ta števila morajo predstavljati opis tiste Poldetove trojice v vhodnem nizu, ki ima največji d. Če obstaja več trojic s tem d, je vseeno, katero izpišeš. Pri naših testnih primerih bo vedno obstajala vsaj ena Poldetova trojica.

V drugo vrstico izpiši eno samo celo število, namreč število Poldetovih trojic z največjim d. Pravzaprav, ker zna biti to število precej veliko, izpiši ostanek po deljenju tega števila z 1 000 037.

(*Nadaljevanje na naslednji strani.*)

Točkovanje: če števila v prvi vrstici tvojega izpisa ustrezajo zahtevam naloge, število v drugi vrstici pa je napačno, dobiš pri tistem testnem primeru 60 % vseh možnih točk. Če sta pravilni obe vrstici, dobiš vse točke.

Primer vhoda:	Eden od možnih pripadajočih izhodov:
`11` `abababaccabab`	`2 4 9 2` `5`

Komentar: v tem vhodnem nizu je pet Poldetovih trojic s podnizi dolžine 2 (štiri trojice iz nizov `ab` in ena iz nizov `ba`).

2. Zamik

Podano je naravno število n. Tvoj program naj izpiše poljubno zaporedje n ničel in enic. Ocenjevalni računalnik ga prebere in na skrivaj krožno zamakne za k mest v desno, nato pa ga mogoče tudi prezrcali z desne na levo (mogoče pa ne). Tvoja naloga je ugotoviti, za koliko mest je ocenjevalni računalnik zamaknil tvoj niz in ali ga je nato tudi prezrcalil. Uporabiš lahko poizvedbe tipa „kakšen je po tem zamiku in morebitnem zrcaljenju i-ti znak niza?“ za različne i. Ugotovi pravi zamik (in to, ali je bil niz po zamiku tudi prezrcaljen ali ne) s čim manj poizvedbami.

V okviru vsakega testnega primera bo moral tvoj program rešiti več takih ugank (vendar ne več kot 100), lahko za različne n. To je interaktivna naloga — tvoj program se bo „pogovarjal“ z ocenjevalnim računalnikom tako, da bo bral s standardnega vhoda in pisal na standardni izhod. Ta pogovor naj poteka po naslednjih korakih:

1. Preberi vrstico, v kateri je celo število n za naslednjo uganko. Če je $n < 0$, je to znak, da je ugank konec in da naj se tvoj program neha izvajati. Sicer bo $n \geq 6$ in moraš nadaljevati z naslednjim korakom.

2. Izpiši vrstico, v kateri je niz n ničel in enic, ki bi ga rad uporabljal v nadaljevanju te uganke.

3. Nato lahko izvedeš 0 ali več (največ 20) poizvedb. Poizvedbo izvedeš tako, da izpišeš vrstico, v kateri je eno samo celo število i z območja $1 \leq i \leq n$, in nato prebereš odgovor ocenjevalnega računalnika — vrstico, v kateri je eno samo celo število, `0` ali `1`, ki ti pove, kakšen je i-ti znak v nizu po tistem, ko ga je sistem zamaknil in mogoče prezrcalil.

4. Ko ugotoviš, za koliko mest je sistem zamaknil tvoj niz v desno (recimo k) in ali ga je nato tudi prezrcalil ali ne, izpiši vrstico z dvema številoma, ločenima s presledkom: prvo število naj bo k, drugo pa naj bo 0, če sistem tvojega niza ni prezrcalil, oz. 1, če ga je prezrcalil.

5. Nato se vrni na korak 1.

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

Omejitve. Pri tej nalogi je šest testnih primerov, ki imajo različne omejitve in so vredni različno število točk:

(1) $n \leq 10$, sistem nikoli ne prezrcali niza (10 točk);

(2) $n \leq 10$, sistem lahko tudi prezrcali niz (10 točk);

(3) $n \leq 100$, sistem nikoli ne prezrcali niza (15 točk);

(4) $n \leq 100$, sistem lahko tudi prezrcali niz (15 točk);

(5) $n \leq 1024$, sistem nikoli ne prezrcali niza (25 točk);

(6) $n \leq 1024$, sistem lahko tudi prezrcali niz (25 točk).

Še enkrat poudarimo, da bo tvoj program v okviru vsakega od teh šestih testnih primerov moral rešiti po več ugank. Točke za tisti testni primer dobi le, če ustrezno reši vse uganke v njem.

Če tvoj program pri kakšni uganki izpiše napačen odgovor, postavi več kot 20 poizvedb ali pa se v kakšnem drugem pogledu ne drži zgoraj opisanega protokola, ga bo sistem takoj ustavil in pri tem testnem primeru ne bo dobil nobenih točk.

Sicer je število točk pri tistem testnem primeru odvisno od maksimalnega števila poizvedb, ki jih je tvoj program postavil (pri katerikoli od ugank v tem testnem primeru); recimo temu maksimumu m. Če je $m \leq 11$, dobiš pri tem testnem primeru vse točke, sicer pa $m - 11$ točk manj kot vse.

(*Nadaljevanje na naslednji strani.*)

Primer:

Tvoj program izpiše	Sistem izpiše	Komentar
	`7`	rešiti bomo morali uganko za $n = 7$
`1010001`		sistem ta niz pri sebi zamakne in mogoče prezrcali
`1`	`1`	naš program deluje precej naivno:
`2`	`0`	sistematično pregleduje zamaknjeni niz
`3`	`1`	znak za znakom
`4`	`1`	
`5`	`0`	
`6`	`0`	
`7`	`0`	
`4 1`		pravilno smo ugotovili, da je sistem zamaknil naš niz za 4 znake desno in da ga je nato tudi prezrcalil
	`-1`	sistem pravi, da je testnega primera konec

V gornjem primeru je sistem najprej zamaknil naš niz `1010001` za štiri mesta v desno (nastalo je `0001101`) in ga nato še prezrcalil (in dobil `1011000`).

3. Zlaganje slik

V nekem muzeju moderne umetnosti želijo natisniti plakat, ki bo v pomanjšani obliki prikazoval vse slike v njihovi zbirki. Pripravili so torej zaporedje n pravokotnih slik, pri čemer je i-ta od njih široka w_i enot in visoka h_i enot.

Te slike bi zdaj radi zložili v vrstice; širina vrstice je definirana kot vsota širin slik v njej, višina vrstice pa kot maksimum njihovih višin. Zlagati jih moramo po vrsti (v takem vrstnem redu, v kakršnem so podane v vhodnem zaporedju), torej prvih nekaj slik v prvo vrstico, naslednjih nekaj v drugo vrstico in tako naprej. Da plakat ne bo preširok, ne sme biti nobena vrstica širša od s enot.

Ker so slike zelo abstraktne, so videti enako dobro tudi, če jih zavrtimo za 90 stopinj, zato se smemo pri vsaki sliki posebej odločiti, ali bi jo mogoče tako zavrteli ali ne (če sliko i zavrtimo za 90 stopinj, bo potem široka h_i enot namesto w_i, visoka pa bo w_i enot namesto h_i).

Znotraj teh omejitev si želimo, da bi bila vsota višin vseh vrstic čim manjša. **Napiši program**, ki izračuna najmanjšo možno vsoto višin vseh vrstic, ki jo je mogoče doseči na ta način.

Vhodni podatki: v prvi vrstici sta dve celi števili, n in s, ločeni s presledkom. Sledi n vrstic, od katerih i-ta vsebuje celi števili w_i in h_i, ločeni s presledkom. Veljalo bo $1 \leq n \leq 10^4$, $1 \leq w_i \leq 10^9$, $1 \leq h_i \leq 10^9$, $s \leq 10^{13}$. Poleg tega bo s zagotovo večji ali enak dolžini krajše stranice vsakega pravokotnika, tako da bo slike zagotovo mogoče razporediti v vrstice, široke največ s enot.

- Pri prvih 20 % testnih primerov bo $n \leq 20$.
- Pri naslednjih 30 % testnih primerov bo $n \leq 1000$.
- Pri preostalih 50 % testnih primerov bo $n \leq 10^4$.

V vsaki od zgornjih treh skupin bodo pri polovici testnih primerov vse slike kvadratne ($w_i = h_i$).

Izhodni podatki: izpiši najmanjšo možno vsoto višin vrstic, ki jo je mogoče doseči, če slike razporedimo v vrstice v skladu s pravili iz besedila naloge.

Primer vhoda:

```
6 24
14 4
8 3
6 11
4 13
11 9
3 14
```

Pripadajoči izhod:

```
18
```

4. Janko in Metka

Janko in Metka sta v hiši zlobne čarovnice našla n sladkarij. Vrednost i-te sladkarije sta ocenila s c_i. Čarovnici bosta ukradla k sladkarij. Težavo pa imata, ker se ne strinjata vedno, ali je posamezna sladkarija dobra ali ne. Sklenila sta kompromis, da mora biti v izbrani množici k sladkarij *vsaj* x dobrih sladkarij po mnenju Janka in *vsaj* x dobrih po mnenju Metke ($x \leq k$).

Napiši program, ki bo izračunal, kakšna je največja možna vsota vrednosti sladkarij, ki jih lahko odneseta Janko in Metka ob upoštevanju svojega kompromisa.

Vhodni podatki: v prvi vrstici so s presledkom ločena števila n, k in x. V drugi vrstici so podane vrednosti sladkarij $c_1, c_2, \ldots, c_n$, ki so prav tako ločene s presledki. Tretja vrstica opisuje sladkarije, ki so dobre po Jankovem mnenju, četrta pa po Metkinem mnenju. Seznama sladkarij se začneta s številom sladkarij v seznamu, ki mu sledijo s presledki ločene številke sladkarij. Vsi vhodni podatki so pozitivna cela števila.

Omejitve: veljalo bo $1 \leq k \leq n \leq 10^6$ in (za vsak i) $1 \leq c_i \leq 10^9$.

V prvih 20 % testnih primerov bo $n \leq 20$. V naslednjih 40 % testnih primerov bo $n \leq 10^4$.

Pozor, pazite na hitrost branja, ker ima naloga velike vhodne podatke!

Izhodni podatki: izpiši iskano vsoto vrednosti sladkarij, ki jih bosta odnesla Janko in Metka. Zagotovljeno je, da bo rešitev obstajala.

Primer vhoda:

```
11 5 2
15 6 14 1 16 7 90 14 3 4 88
5 5 3 1 4 7
4 9 10 7 4
```

Pripadajoči izhod:

```
213
```

Komentar: v danem primeru se jima najbolj splača vzeti sladkarije 1, 5, 7, 10 in 11. Janku so všeč 1, 5 in 7, Metki pa 7 in 10.

5. Ključavničarstvo

Načrtujemo računalniško igrico, v kateri mora igralec hoditi po stavbi in obiskati vse njene sobe. Stavba obsega n sob in $n-1$ hodnikov med njimi. Vsak hodnik neposredno povezuje natanko dve sobi. Prek teh hodnikov je vsaka soba (v enem ali več korakih) povezana z vsako drugo, vendar natanko na en način (z drugimi besedami, hodniki ne tvorijo ciklov).

Sobe so oštevilčene od 1 do n. Na začetku igre je v vsaki sobi nekaj ključev, vsak hodnik med dvema sobama pa je zaklenjen z 0, 1 ali več ključavnicami. Ko igralec prvič vstopi v neko sobo, pri tem avtomatsko pobere vse ključe v njej. Igralec začne svoj sprehod v sobi 1; pred tem ni imel ključev, vendar pa takoj na začetku pobere ključe, ki so bili v sobi 1.

Igralec lahko z vsakim ključem odpre katero koli ključavnico, ki nato ostane odklenjena, uporabljeni ključ pa se pri tem zlomi in ni več uporaben. Po hodniku lahko gre igralec iz ene sobe v drugo šele, če odklene vse ključavnice v njem (ni pa nujno, da odklene vse; dokler je v njem kakšna ključavnica zaklenjena, je hodnik neprehoden). Ko je hodnik odklenjen, se sme igralec po njem sprehoditi tudi po večkrat in to v obe smeri; tako se lahko torej tudi vrača v sobe, ki jih je prej že obiskal.

Želimo se izogniti scenariju, kjer igralec zaradi nespametnega zaporedja obiska sob in odklepanja ključavnic ne more končati igre, ker bi, še preden je obiskal vse sobe, porabil vse ključe in obstal pred zaklenjenim hodnikom. **Napiši program**, ki ugotovi, ali je tak scenarij mogoč.

Vhodni podatki: v okviru enega testnega primera bo moral tvoj program obdelati več stavb. V prvi vrstici je število stavb T. Sledijo opisi stavb, pred vsakim pa je prazna vrstica.

Vsaka stavba je opisana takole: v prvi vrstici je celo število n (število sob). Sledi vrstica z n celimi števili, $k_1, k_2, \ldots, k_n$, ločenimi s po enim presledkom; pri tem število k_i pove, koliko ključev je na začetku igre v i-ti sobi. Sledi še $n-1$ vrstic, ki opisujejo hodnike; j-ta od teh vrstic vsebuje tri cela števila, u_j, v_j in ℓ_j (ločena s po enim presledkom), ki povedo, da j-ti hodnik neposredno povezuje sobi u_j in v_j in da je ob začetku igre zaklenjen s ℓ_j ključavnicami.

Omejitve: veljalo bo $1 \le n \le 10^5$, $0 \le k_i \le 10^4$, $1 \le u_j < v_j \le n$, $0 \le \ell_j \le 10^4$ in $1 \le T \le 10$.

- Pri prvih 20 % testnih primerov bo $n \le 10$.
- Pri naslednjih 20 % testnih primerov bo $n \le 1000$, sobe bodo s hodniki povezane v en samo dolgo zaporedje (seznam) brez razvejitev, skupno število ključev pa ne bo enako skupnemu številu ključavnic.
- Pri naslednjih 20 % testnih primerov bo $n \le 1000$.
- Pri naslednjih 20 % testnih primerov bodo vsi $\ell_j = 1$ in skupno število ključev ne bo enako skupnemu številu ključavnic.
- Pri zadnjih 20 % testnih primerov ni posebnih dodatnih omejitev.

Izhodni podatki: po vrsti izpiši po eno vrstico za vsako stavbo; ta vrstica naj vsebuje le niz „`da`“ ali „`ne`“ (brez narekovajev), ki pove, ali je pri tisti stavbi mogoč tak scenarij, po kakršnem sprašuje naloga.

Primer vhoda:

```
2

4
3 0 1 2
1 2 1
1 3 2
2 4 2

4
3 1 1 1
1 2 4
1 3 1
1 4 0
```

Pripadajoči izhod:

```
da
ne
```

Komentar: v prvem primeru lahko igralec iz sobe 1 obišče sobo 2, nato pa odklene ena izmed vrat na hodniku $(1, 3)$ in ena izmed vrat na hodniku $(2, 4)$. Ostane brez ključev, ne da bi obiskal sobi 3 in 4. V drugem primeru bo igralec vedno lahko obiskal vse sobe.

Omenimo še, da je prvi tloris veljaven primer zaporedja (seznama) brez razvejitev, karkšni se pojavljajo v drugi izmed podnalog.

15. tekmovanje ACM v znanju računalništva za srednješolce

28. marca 2020

REŠITVE NALOG ZA PRVO SKUPINO

1. Vesoljske vsote

Vhodni niz bomo brali znak po znak in pri tem v dveh spremenljivkah hranili trenutno število in dosedanjo vsoto. Za potrebe izpisa pa je koristno hraniti še podatek o tem, ali smo že izpisali kak seštevanec ali ne — z drugimi besedami, ali smo v vhodnem nizu doslej že naleteli na kakšno zvezdico „*“ ali ne. To pride prav zato, ker moramo med seštevanci napisati znak + in presledke; to lahko preprosto izvedemo tako, da izpišemo + pred vsakim seštevancem razen pred prvim. Ko pridemo do konca niza, moramo izpisati le še enačaj in vsoto na koncu izraza.

Oglejmo si takšno rešitev v C/C++:

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

void VesoljskeVsote(const char *s)
{
  int trenutno = 1, vsota = 0;
  bool prvi = true;
  while (*s) // Sprehodimo se po znakih vhodnega niza.
    if (*s++ == '-') // Pri minusu povečamo trenutno število.
      trenutno++;
    else // Pri zvezdici prištejemo trenutno število k vsoti.
    {
      // Pred vsakim seštevancem, razen pred prvim, izpišimo +.
      if (prvi) prvi = false; else printf(" + ");
      printf("%d", trenutno); // Izpišimo trenutni seštevanec.
      vsota += trenutno; // Prištejmo ga k vsoti.
    }
  // Na koncu izpišimo še enačaj in vsoto.
  printf(" = %d\n", vsota);
}
```

Še prav taka rešitev v pythonu:

```
import sys

def VesoljskeVsote(s):
  trenutno = 1; vsota = 0; prvi = True
  for c in s: # Sprehodimo se po znakih vhodnega niza.
    # Pri minusu povečamo trenutno število.
    if c == "-": trenutno += 1
    else: # Pri zvezdici prištejemo trenutno število k vsoti.
      # Pred vsakim seštevancem, razen pred prvim, izpišimo +.
      if prvi: prvi = False
      else: sys.stdout.write(" + ")
      sys.stdout.write(str(trenutno))   # Izpišimo trenutni seštevanec.
      vsota += trenutno                 # Prištejmo ga k vsoti.
  # Na koncu izpišimo še enačaj in vsoto.
  sys.stdout.write(" = %d\n" % vsota)
```

Nalogo lahko rešimo tudi malo drugače. Spodnja rešitev v pythonu med branjem vhodnega niza ničesar ne izpisuje, pač pa dodaja seštevance v seznam cleni. Ko pride do konca niza, s pomočjo tega seznama izpiše najprej vse člene vsote (ločene s plusi) in na koncu še vrednost vsote (ki jo izračuna s pythonovo funkcijo sum).

```
def VesoljskeVsote2(s):
  trenutno = 1; cleni = []
  for c in s:
    # Pri minusu povečamo trenutno število.
    if c == "-": trenutno += 1
    # Pri zvezdici dodamo trenutno število v seznam členov.
    else: cleni.append(trenutno)
  # Izpišimo člene, ločene s plusi, in nato še enačaj in vsoto.
  print("%s = %d" % (" + ".join(str(clen) for clen in cleni), sum(cleni)))
```

2. Ključ

Naloga pravi, da dobimo opis ključa kot celo število (od 0 do 999 999). Da pridemo do posameznih števk, si lahko pomagamo z dejstvom, da je ostanek po deljenju števila z 10 ravno najbolj desna števka tega števila (enice), celi del količnika pri tem deljenju pa je tisto, kar od števila ostane, če mu zadnjo števko pobrišemo. Tako lahko v zanki pregledujemo zareze ključa od desne proti levi.

Pri vsaki zarezi preverimo, če je od 1 do 8 (prvi pogoj iz besedil analoge). Pri vsaki zarezi razen prve tudi preverimo, če je različna od prejšnje (tretji pogoj), vendar ne za več kot pet (drugi pogoj); pri tem si moramo torej prejšnjo zarezo zapomniti (v spodnji rešitvi jo shranimo v spremenljivko prejsnja).

Za četrti pogoj je koristno imeti tabelo, v kateri štejemo, kolikokrat smo kakšno števko že videli; na začetku postavimo vse elemente na 0, nato pa pri vsaki prebrani zarezi ustrezni element tabele povečamo za 1 in pogledamo, če je zdaj večji od 2.

Za vsak pogoj imamo še eno logično spremenljivko (ok1, ..., ok4), ki nam pove, ali je ključ ta pogoj prekršil ali ne; vrednosti teh spremenljivk na koncu izpišemo.

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

enum { Dolzina = 6, MaxRazlika = 5 };

void Preveri(int kljuc)
{
  bool ok1 = true, ok2 = true, ok3 = true, ok4 = true;
  int stPojavitev[10] = { }, zareza = -1;
  for (int i = 0; i < Dolzina; i++)
  {
    // Izluščimo naslednjo zarezo iz spremenljivke 'kljuc'.
    int prejsnja = zareza; zareza = kljuc % 10; kljuc /= 10;

    // Pogoj 1: ali so vse zareze od 1 do 8?
    if (zareza < 1 || zareza > 8) ok1 = false;

    if (i > 0) { // Ker to ni prva zareza, jo lahko primerjamo s prejšnjo.
      int razlika = zareza - prejsnja;

      // Pogoj 2: sosednji zarezi se ne smeta preveč razlikovati.
      if (razlika < -MaxRazlika || razlika > MaxRazlika) ok2 = false;

      // Pogoj 3: sosednji razrezi ne smeta biti enaki.
      else if (razlika == 0) ok3 = false; }

    // Pogoj 4: ne smemo imeti treh ali več enakih zarez.
    if (++stPojavitev[zareza] > 2) ok4 = false;
  }
  // Izpišimo rezultate.
  printf("Ključ ustreza naslednjim pravilom: 1 %s, 2 %s, 3 %s, 4 %s.\n",
    ok1 ? "da" : "ne", ok2 ? "da" : "ne", ok3 ? "da" : "ne", ok4 ? "da" : "ne");
}
```

Zapišimo to rešitev še v pythonu. Za spremembo bomo namesto štirih logičnih spremenljivk uporabili tabelo (list v pythonu) s štirimi elementi:

```
Dolzina = 6; MaxRazlika = 5

def Preveri(kljuc):
```

```
ok = [True] * 4; stPojavitev = [0] * 10; zareza = 1
for i in range(Dolzina):
    # Izluščimo naslednjo zarezo iz spremenljivke „ključ“.
    prejsnja = zareza; zareza = kljuc % 10; kljuc //= 10
    # Pogoj 1: ali so vse zareze od 1 do 8?
    if zareza < 1 or zareza > 8: ok[0] = False
    if i > 0: # Ker to ni prva zareza, jo lahko primerjamo s prejšnjo.
        razlika = zareza - prejsnja
        # Pogoj 2: sosednji zarezi se ne smeta preveč razlikovati.
        if abs(razlika) > MaxRazlika: ok[1] = False
        # Pogoj 3: sosednji razrezi ne smeta biti enaki.
        elif razlika == 0: ok[2] = False
    # Pogoj 4: ne smemo imeti treh ali več enakih zarez.
    stPojavitev[zareza] += 1
    if stPojavitev[zareza] > 2: ok[3] = False
# Izpišimo rezultate.
print("Ključ ustreza naslednjim pravilom: 1 %s, 2 %s, 3 %s, 4 %s." %
    tuple("da" if b else "ne" for b in ok))
```

3. Obračanje jogija

Jogi ima dve strani (zgornjo in spodnjo — to sta tisti dve ploskvi, ki sta pravokotni na os z), na vsaki strani pa imamo lahko glavo na enem od dveh koncev (pri eni od krajših stranic jogija, torej tistih, ki so vzporedne z osjo x). Tako lahko vsako od štirih mest, kjer imamo lahko glavo, opišemo s parom bitov (s, k), ki povesta stran in konec. Drug pogled na tak par bitov pa je seveda ta, da si ga predstavljamo kot število $2s + k$, torej eno od števil 0, 1, 2 ali 3.

S pomočjo tega številčenja mest lahko razmislimo, kaj se zgodi pri vrtenju jogija (pri tem je koristno gledati na sliko, ki kaže, kam so usmerjene koordinatne osi). Če smo doslej spali na mestu (s, k) in jogi nato zavrtimo za 180 stopinj okrog osi z, bo pod našo glavo prišlo mesto $(s, 1 - k)$, torej na isti strani jogija, vendar na drugem koncu. Podobno, če ga zavrtimo okrog osi y, bo pod našo glavo prišlo mesto $(1 - s, k)$, torej na istem koncu jogija, vendar na drugi strani. Če pa ga zavrtimo okrog osi x, pride pod našo glavo mesto $(1 - s, 1 - k)$.

Naša rešitev bo morala v globalnih spremenljivkah za vsa štiri mesta hraniti podatek o tem, kolikokrat je uporabnik doslej spal na njih. Ko uporabnik pokliče našo funkcijo, moramo najprej ustrezno povečati števec obrabljenosti za tisto mesto, na katerem je spal doslej, nato pa pogledati, katero mesto je zdaj najmanj obrabljeno (če je tu več enako dobrih možnosti, je vseeno, katero uporabimo), in s pomočjo razmisleka iz prejšnjega odstavka svetovati, kako je treba zasukati jogi, da bo prišlo pod uporabnikovo glavo najmanj obrabljeno mesto: če je uporabnik doslej spal na (s, k), najmanj obrabljeno pa je $(1 - s, k)$, mu moramo svetovati vrtenje okrog osi y in podobno. Če imamo mesta predstavljena z dvobitnimi celimi števili od 0 do 3, je dovolj že, če pogledamo, v katerih bitih se razlikujeta stara in nova številka mesta (pri tem lahko uporabimo operator xor oz. ^): če v nižjem (konec), je treba obrniti jogi okrog osi z; če v višjem (stran), okrog y; če v obeh, okrog x; če v nobenem, pa jogija ni treba obračati.

Na začetku sicer ne vemo, kako je jogi obrnjen, vendar to za nas tudi ni pomembno, saj so vsa mesta popolnoma neobrabljena. Tisto mesto, na katerem je uporabnikova glava na začetku izvajanja programa, lahko preprosto razglasimo za $(0, 0)$, ostale kombinacije dveh bitov pa potem pač predstavljajo ostala tri mesta odvisno od tega, ali ležijo na istem ali na drugem koncu/strani kot začetno mesto.

```
int obrabljenost[4] = { }, trenutno = 0;

char ObrniJogi(int n)
{
    // Popravimo števec obrabljenosti trenutnega mesta.
    obrabljenost[trenutno] += n;
    // Poglejmo, katero mesto je zdaj najmanj obrabljeno.
```

```
  int novo = 0; for (int i = 1; i < 4; i++)
    if (obrabljenost[i] < obrabljenost[novo]) novo = i;

  // Poglejmo, kako obrniti jogi, da pride mesto „novo“ tja,
  // kjer je bilo doslej mesto „trenutno“.
  char kakoObrniti = "nzyx"[novo ^ trenutno];

  // Zapomnimo si, da bo uporabnik odslej spal na novem mestu.
  trenutno = novo; return kakoObrniti;
}
```

Zapišimo to rešitev še v pythonu:

```
obrabljenost = [0] * 4; trenutno = 0

def ObrniJogi(n):
  global trenutno

  # Popravimo števec obrabljenosti trenutnega mesta.
  obrabljenost[trenutno] += n;

  # Poglejmo, katero mesto je zdaj najmanj obrabljeno.
  novo = 0
  for i in range(1, 4):
    if obrabljenost[i] < obrabljenost[novo]: novo = i

  # Poglejmo, kako obrniti jogi, da pride mesto „novo“ tja,
  # kjer je bilo doslej mesto „trenutno“.
  kakoObrniti = "nzyx"[novo ^ trenutno];

  # Zapomnimo si, da bo uporabnik odslej spal na novem mestu.
  trenutno = novo; return kakoObrniti
```

4. Zobna ščetka

Naš program bo tekel v neskončni zanki. Vsakič preverimo stanje tipke; pritisk na tipko prepoznamo po tem, da je trenutno pritisnjena, ob prejšnjem preverjanju pa še ni bila. Poleg tega si zapomnimo tudi, ali je motor trenutno prižgan ali ne (v spodnji rešitvi je to spremenljivka prizgana).

Če je uporabnik pritisnil na tipko in je bil motor doslej ugasnjen, ga moramo prižgati; ugasniti pa ga moramo, če je bil doslej prižgan in če je uporabnik zdaj pritisnil na tipko ali pa če motor teče že 120 sekund. Če ni izpolnjen noben od teh pogojev, moramo motor pustiti v dosedanjem stanju (stavek **continue** v spodnji rešitvi), sicer pa mu stanje spremenimo (ga vklopimo, če je bil prej izklopljen, oz. izklopimo, če je bil vklopljen). Ko motor poženemo, moramo tudi prižgati štoparico, ko pa ga ugasnemo, moramo štoparico ustaviti.

```
int main()
{
  bool prizgana = false, tipka = false;
  while (true)
  {
    // Zapomnimo si prejšnje stanje tipke in poglejmo sedanje.
    bool prejTipka = tipka; tipka = Tipka();

    // Če uporabnik pritisne tipko in je ščetka ugasnjena,
    // jo bo treba prižgati. Pri tem tudi poženimo štoparico.
    if (tipka && ! prejTipka && ! prizgana) PozeniUro();

    // Če je ščetka prižgana in uporabnik pritisne tipko ali pa je prižgana že
    // več kot 120 sekund, jo bo treba ugasniti. Pri tem tudi ustavimo štoparico.
    else if (tipka && ! prejTipka || prizgana && OdcitajUro() > 120) UstaviUro();

    // Sicer lahko ščetka ostane v sedanjem stanju.
    else continue;

    // Postavimo motor v novo stanje in si ga zapomnimo.
    prizgana = ! prizgana; Motor(prizgana);
  }
}
```

Zapišimo to rešitev še v pythonu:

```
prizgana = False; tipka = False
while True:
  # Zapomnimo si prejšnje stanje tipke in poglejmo sedanje.
  prejTipka = tipka; tipka = Tipka()

  # Če uporabnik pritisne tipko in je ščetka ugasnjena,
  # jo bo treba prižgati. Pri tem tudi poženimo štoparico.
  if tipka and not prejTipka and not prizgana: PozeniUro()

  # Če je ščetka prižgana in uporabnik pritisne tipko ali pa je prižgana že
  # več kot 120 sekund, jo bo treba ugasniti. Pri tem tudi ustavimo štoparico.
  elif tipka and not prejTipka or prizgana and OdcitajUro() > 120: UstaviUro()

  # Sicer lahko ščetka ostane v sedanjem stanju.
  else: continue

  # Postavimo motor v novo stanje in si ga zapomnimo.
  prizgana = not prizgana; Motor(prizgana)
```

5. Prepisovanje

Vhodne podatke si lahko predstavljamo kot zaporedje parov $\langle pomočnik, plonkar \rangle$. Vsako od treh podnalog lahko rešimo s po enim prehodom po tem seznamu.

V prvem prehodu si pripravimo množico vseh pomočnikov in množico vseh plonkarjev; izvirni tekmovalci so potem preprosto tisti, ki so v množici pomočnikov, ne pa tudi v množici plonkarjev. (Namesto tega lahko naredimo tudi dva prehoda: v prvem sestavimo množico vseh pomočnikov, v drugem pa iz nje pomečemo vse plonkarje, pa nam ostanejo ravno vsi izvirni tekmovalci; ali pa v prvem prehodu sestavimo množico vseh plonkarjev, v drugem pa dodamo v množico izvirnih tekmovalcev le tiste pomočnike, ki niso v množici plonkarjev iz prvega prehoda.)

V drugem prehodu pripravimo množico plonkarjev prvega reda tako, da pri vsakem paru z vhodnega seznama pogledamo, če je pomočnik eden od izvirnih tekmovalcev, in če je, dodamo plonkarja v množico plonkarjev prvega reda.

V tretjem prehodu pripravimo množico plonkarjev drugega reda tako, da pri vsakem paru pogledamo, če je pomočnik eden od plonkarjev prvega reda, in če je, dodamo plonkar med plonkarje drugega reda.

Množice lahko predstavimo z razpršenimi tabelami ali pa tudi z navadno tabelo logičnih vrednosti (za vsakega tekmovalca po ena, ki pove, če tekmovalec pripada tej množici ali ne), če so številke tekmovalcev dovolj majhne, da jih lahko uporabljamo kot indekse v tabelo.

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

struct Par { int pomocnik, plonkar; };
typedef unordered_set<int> Mnozica;

void Prepisovanje(const vector<Par> &pari,
                  Mnozica &izvirni, Mnozica &plonkarji1, Mnozica &plonkarji2)
{
  // Pripravimo množico vseh plonkarjev.
  Mnozica plonkarji; for (auto &P : pari) plonkarji.insert(P.plonkar);

  // Izvirni tekmovalci so pomočniki, ki niso plonkarji.
  izvirni.clear(); for (auto &P : pari)
    if (! plonkarji.count(P.pomocnik)) izvirni.insert(P.pomocnik);

  // Pripravimo množico plonkarjev prvega reda.
  plonkarji1.clear(); for (auto &P : pari)
    if (izvirni.count(P.pomocnik)) plonkarji1.insert(P.plonkar);

  // Pripravimo množico plonkarjev drugega reda.
  plonkarji2.clear(); for (auto &P : pari)
    if (plonkarji1.count(P.pomocnik)) plonkarji2.insert(P.plonkar);
}
```

Še bolj jedrnati smo lahko v pythonu:

```
def Prepisovanje(pari):
  plonkarji = set(plonkar for (pomocnik, plonkar) in pari)
  izvirni = set(pomocnik for (pomocnik, plonkar) in pari if pomocnik not in plonkarji)
  plonkarji1 = set(plonkar for (pomocnik, plonkar) in pari if pomocnik in izvirni)
  plonkarji2 = set(plonkar for (pomocnik, plonkar) in pari if pomocnik in plonkarji1)
  return (izvirni, plonkarji1, plonkarji2)
```

Ni si težko predstavljati, da bi lahko definicije iz naše naloge še posplošili in govorili o plonkarjih tretjega reda, pa četrtega in tako naprej. Našo dosedanjo rešitev bi lahko nadaljevali na podoben način kot doslej in računali še plonkarje višjih redov, vendar pa bi se ob tem pokazala slabost te rešitve: za vsak naslednji red potrebuje po en prehod čez celoten seznam vhodnih parov. Če imamo n tekmovalcev in s tem tudi $O(n)$ parov v vhodnem seznamu (ker je le toliko parov, kolikor je vseh plonkarjev), porabimo tako $O(n)$ časa za vsak red, ki ga hočemo računati; in ker bi šli lahko redovi v najslabšem primeru do $n-1$, bi ta postopek porabil takrat $O(n^2)$ časa.

Boljšo rešitev dobimo, če v prvem prehodu čez vhodni seznam predelamo podatke v drugačno obliko: za vsakega tekmovalca pripravimo seznam tistih, ki so plonkali od njega. Spotoma lahko za vsakega označimo še, ali je sam od koga plonkal ali ne. To je dovolj, da s še enim prehodom čez vse tekmovalce pripravimo seznam izvirnih (to so tisti, ki niso od nikogar plonkali). Nato lahko seznam plonkarjev prvega reda pripravimo tako, da gremo po vseh izvirnih tekmovalcih in staknemo skupaj sezname vseh tistih, ki so prepisovali od njih. Pomembna razlika v primerjavi s prvotno rešitvijo je torej ta, da nam ni treba iti še enkrat po vseh podatkih, ampak le po izvirnih tekmovalcih. V nadaljevanju gremo lahko na enak način po seznamu tekmovalcev prvega reda in staknemo skupaj sezname tistih, ki so prepisovali od njih, pa dobimo seznam tekmovalcev drugega reda. Tako bi lahko nadaljevali še za vse višje redove; vsakega tekmovalca največ enkrat dodamo na en tak seznam (pri njegovem redu) in se nato enkrat sprehodimo po seznamu tistih, ki so prepisovali od njega (ko računamo za eno višji red); časovna zahtevnost celotnega postopka, za vse redove skupaj, je tako le še $O(n)$. Lepo je tudi to, da nam ni več treba delati z množicami (ki so implementirane npr. z razpršenimi tabelami; pri prejšnji rešitvi smo jih potrebovali, da smo lahko učinkovito preverjali, ali pomočnik v nekem paru pripada prejšnjemu redu, ko računamo naslednji red), ampak so dovolj že navadni seznami (npr. vektorji).

Oglejmo si implementacijo takšne rešitve v C++:

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

void Prepisovanje(const vector<Par> &pari,
                  vector<int> &izvirni, vector<int> &plonkarji1, vector<int> &plonkarji2)
{
  struct Tekmovalec
  {
    bool jePlonkar = false; // ali je on plonkal
    vector<int> plonkarji; // kdo je plonkal od njega
  };
  // Pripravimo za vsakega tekmovalca seznam, kdo je plonkal od njega.
  unordered_map<int, Tekmovalec> T;
  for (auto &P : pari) {
    T[P.pomocnik].plonkarji.push_back(P.plonkar);
    T[P.plonkar].jePlonkar = true; }
  // Začeli bomo s seznamom izvirnih tekmovalcev.
  izvirni.clear(); for (auto &[u, U] : T) if (! U.jePlonkar) izvirni.push_back(u);
  // Kdor je plonkal od njih, je plonkar prvega reda.
  plonkarji1.clear(); for (auto u : izvirni) for (int v : T[u].plonkarji) plonkarji1.push_back(v);
  // Kdor je plonkal od teh, pa je plonkar drugega reda.
  plonkarji2.clear(); for (auto u : plonkarji1) for (int v : T[u].plonkarji) plonkarji2.push_back(v);
}
```

Zapišimo to rešitev še v pythonu:

```
def Prepisovanje2(pari):
    class Tekmovalec:
        def __init__(self): self.jePlonkar = False; self.plonkarji = []
    # Pripravimo za vsakega tekmovalca seznam, kdo je plonkal od njega.
    T = {}
    for (pomocnik, plonkar) in pari:
        if pomocnik not in T: T[pomocnik] = Tekmovalec()
        if plonkar not in T: T[plonkar] = Tekmovalec()
        T[pomocnik].plonkarji.append(plonkar)
        T[plonkar].jePlonkar = True
    # Začeli bomo s seznamom izvirnih tekmovalcev.
    izvirni = [u for (u, U) in T.items() if not U.jePlonkar]
    # Kdor je plonkal od njih, je plonkar prvega reda.
    plonkarji1 = [v for u in izvirni for v in T[u].plonkarji]
    # Kdor je plonkal od teh, pa je plonkar drugega reda.
    plonkarji2 = [v for u in plonkarji1 for v in T[u].plonkarji]
    return (izvirni, plonkarji1, plonkarji2)
```

REŠITVE NALOG ZA DRUGO SKUPINO

1. Metanje na koš

Vhodni niz lahko v mislih razdelimo na strnjene skupine enic, med katerimi so ničle. Kjer stojita dve ničli skupaj, si mislimo med njima prazno skupino enic. Za niz 35 metov iz besedila naloge dobimo na primer:

$$1\,1\,1\,0\,1\,1\,1\,1\,0\;\cdot\;0\,1\,1\,0\,1\,1\,1\,1\,1\,0\,1\,1\,1\,1\,0\;\cdot\;0\,1\,0\,1\,1\,1\,1\,1\,1\,1\,1\,0\,\cdot,$$

pri čemer smo prazne skupine enic označili s pikami „·“. Zapišimo dolžine tako dobljenih enic kot seznam:

$$3, 4, 0, 2, 5, 4, 0, 1, 7, 0.$$

Vsaka skupina $k+1$ zaporednih členov v tem seznamu predstavlja zaporedje metov na koš, v katerem je k ničel (metov mimo koša), ostalo pa so enice (zadetki); to pa je prav táko zaporedje, po kakršnem sprašuje naša naloga. Skupno število zadetkov v takem zaporedju dobimo tako, da seštejemo tistih $k+1$ zaporednih členov našega seznama. To lahko počnemo v zanki: gremo po zaporedju, na vsakem mestu izračunamo vsoto zadnjih $k+1$ členov in si zapomnimo največjo od tako dobljenih vsot. Pri tem je koristno upoštevati še to, da nam vsote ni treba računati vsakič od začetka; ko se premaknemo za en člen naprej, pridobi vsota na desni en člen, na levi pa najstarejšega izgubi, tako da lahko staro vsoto preprosto in poceni popravimo v novo.

```
int NajdaljsiNizZadetkov(const char *s, int k)
{
  // Pripravimo seznam z dolžinami strnjenih skupin enic.
  // Kjer sta dve ničli skupaj, si mislimo med njima skupino 0 enic.
  // Prvi element seznama predstavlja enice pred prvo ničlo.
  vector<int> skupine; skupine.push_back(0);
  for (int i = 0; s[i]; i++) {
    // Ko pridemo do ničle, začnemo novo skupino enic.
    if (s[i] == '0') skupine.push_back(0);
    // Ko smo pri enici, se trenutna skupina enic podaljša.
    else ++skupine.back(); }

  // Računajmo vsote po k + 1 zaporednih elementov seznama.
  int vsota = 0, naj = 0;
  for (int i = 0; i < skupine.size(); i++)
  {
    // Prištejmo trenutni element.
    vsota += skupine[i];
    // Če ima vsota zdaj že k + 2 členov, najstarejšega odštejmo.
    if (i > k) vsota -= skupine[i - (k + 1)];
    // Največjo vsoto si zapomnimo.
    naj = max(naj, vsota);
  }
  return naj; // Vrnimo najboljšo rešitev.
}
```

Ta rešitev v najslabšem primeru porabi $O(n)$ pomnilnika za seznam z dolžinami skupin enic, če je n dolžina vhodnega niza. Lahko bi jo še malo izboljšali, če bi seznam gradili sproti, medtem ko se premikamo po njem, in iz njega tudi sproti brisali stare elemente, ki jih ne bomo več potrebovali (tiste, ki so več kot k mest levo od trenutnega); poraba pomnilnika bi se s tem zmanjšala na $O(k)$.

Na podobni ideji temelji tudi naslednja rešitev, ki namesto dolžin skupin enic gleda položaje ničel. Ko se premikamo po nizu znak po znak od leve proti desni, vzdržujmo v tabeli oz. vektorju nicle položaje zadnjih $k+1$ ničel, ki smo jih doslej videli. Podniz, ki se začne takoj za najbolj levo izmed teh ničel in traja vse do našega trenutnega položaja, vsebuje torej k ničel, ostalo pa so enice; zato je to podniz take oblike, po kakršni sprašuje

naša naloga, število enic v njem pa je ravno za k manjše od dolžine podniza. Med vsemi tako dobljenimi možnostmi si spet zapomnimo najdaljšo.

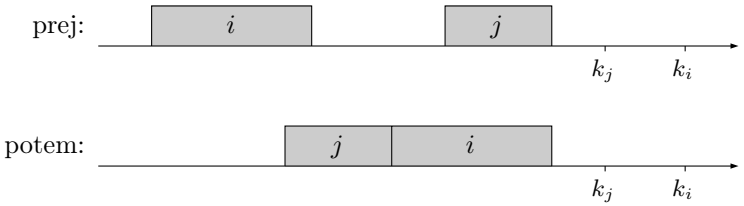
Za inicializacijo tega postopka se je koristno pretvarjati, da levo od začetka našega vhodnega niza stoji še $k+1$ ničel (če indekse v vhodne nizu štejemo od 0 do $n-1$, kjer je n dolžina niza, si mislimo, da stojijo ničle tudi na indeksih $-(k+1), -k, \ldots, -2, -1$).

Vektor zadnjih $k+1$ ničel je koristno uporabljati kot krožno tabelo (*ring buffer*): zapomnimo si, na katerem indeksu v njej je najbolj leva izmed teh $k+1$ ničel (spremenljivka prva); in ko naletimo na naslednjo ničlo, vpišemo njen položaj čez tisto najbolj levo, indeks prva pa pomaknemo za eno mesto naprej, ker je tam zdaj najbolj leva izmed preostalih ničel. Tako imamo pri vsaki ničli le $O(1)$ dela, da ustrezno popravimo vektor nicle.

```
int NajdaljsiNizZadetkov2(const char *s, int k)
{
  // V vektorju „nicle“ bomo hranili indekse zadnjih k + 1 ničel; uporabljali jo
  // bomo kot krožno tabelo, najbolj leva od teh ničel je nicle[prva].
  vector<int> nicle(k + 1); int prva = 0;

  // Za začetek se pretvarjajmo, da je levo od niza še k + 1 ničel.
  for (int i = 0; i <= k; i++) nicle[k − i] = −i − 1;

  // Sprehodimo se po nizu.
  int naj = 0;
  for (int i = 0; s[i]; i++)

    if (s[i] == '1')
      // Znaki na indeksih od nicle[prva] + 1 do k tvorijo zaporedje
      // s k ničlami, ostalo pa so enice.
      naj = max(naj, i − nicle[prva] − k);

    else { // Najbolj levo ničlo v seznamu „nicle“ povozimo s trenutno ničlo.
      nicle[prva] = i; prva = (prva + 1) % (k + 1); }

  return naj; // Vrnimo najboljšo rešitev.
}
```

2. Ne odlašaj na jutri, kar lahko storiš pojutrišnjem

Hitro se dá videti, da je smiselno obveznosti opravljati v takem vrstnem redu, v kakršnem morajo biti končane (torej po naraščajočih vrednostih k_i). O tem se lahko prepričamo takole: recimo, da opravimo obveznost i in nato (mogoče z nekaj dnevi premora) še obveznost j, pri čemer pa je $k_j < k_i$. Če je med njima premor, lahko i zamaknemo toliko naprej, da se konča takrat, ko se j začne, pa bo razpored še vedno veljaven (i se konča pred j, njegov rok pa je za j-jevim; če se j tu konča pravočasno, se i tudi), odlašanja pa je še več. Zdaj smo torej v stanju, ko se i konča takrat, ko se j začne; lahko ju enostavno zamenjamo, pa se bo j končal bolj zgodaj kot prej (in torej še vedno do roka), i pa se bo končal takrat, ko se je prej j (kar je do roka za j, zato pa tudi do roka za i, saj ima slednji kasnejši rok).



Tako lahko torej razpored, v katerem opravila niso urejena po naraščajočem roku k_i, predelamo v takega, kjer so tako urejena, ne da bi se kaj poslabšal.

Za začetek lahko torej obveznosti uredimo po k_i in jih nato obdelujemo od konca proti začetku, torej po padajočih k_i. Zadnjo obveznost (tisto z največjim k_i) je smiselno začeti ob času $z_i := k_i - d_i$; kasneje je ne smemo (ker potem ne bi bila končana do roka), bolj zgodaj pa tudi ne (ker bi v tem primeru razpored lahko še izboljšali, če bi z začetkom te obveznosti še malo počakali). Razmislimo zdaj o predzadnji obveznosti — recimo ji j: ne smemo je začeti kasneje kot ob $k_j - d_j$ (ker potem ne bi bila končana do roka), pa tudi ne kasneje kot ob $z_i - d_j$ (ker potem ne bi bila končana do takrat,

ko moramo začeti zadnjo obveznost (katere z_i smo določili malo prej). Obveznost j moramo torej začeti najkasneje ob $z_j := \min\{k_j, z_i\} - d_j$; bolj zgodaj kot to pa je nima smisla začenjati, saj bi se v tem primeru dalo razpored še izboljšati, če bi njen začetek malo odložili.

Enako lahko razmišljamo tudi pri vseh zgodnejših obveznostih; vsaka se mora torej končati do svojega roka in tudi do začetka naslednje obveznosti. Tako lahko načeloma za vsako obveznost i določimo njen začetni čas z_i. Naloga pravi, da so dnevi oštevilčeni z naravnimi števili od začetka leta, tako da, če na koncu dobimo pri najzgodnejši obveznosti $z_i \leq 0$, je to znak, da je problem nerešljiv (saj dni s številkami, manjšimi od 1, ni), sicer pa smo dobili primeren razpored, po kakršnem sprašuje naloga.

Oglejmo si še implementacijo tega postopka v jeziku C++. Vhodne podatke bomo predstavili z vektorjem majhnih struktur, ki naj ob klicu za vsako obveznost povedo njeno dolžino d_i in rok k_i, do katerega mora biti končana, naša funkcija pa bo v vsako zapisala primeren dan začetka z_i. Funkcija vrne logično vrednost, ki pove, ali obstaja veljaven razpored (tak, v katerem so vsi časi pozitivni). Začetek naslednje obveznosti hranimo v spremenljivki zacNaslednje; ob koncu nam ta pove začetek prve (najzgodnejše) obveznosti, za katero moramo le še preveriti, če je večja od 0.

```
struct Obveznost { int rok, dolzina, zacetek; };

bool PoisciRazpored(vector<Obveznost>& O)
{
  int n = O.size();
  // Pripravimo si vrstni red, v katerem so obveznosti urejene padajoče po roku.
  vector<int> vrstniRed(n);
  for (int i = 0; i < n; i++) vrstniRed[i] = i;
  sort(vrstniRed.begin(), vrstniRed.end(), [&] (int i, int j) {
    return O[i].rok > O[j].rok; });
  // V tem vrstnem redu jih pregledujmo in določajmo čas začetka.
  int zacNaslednje = 1;
  for (int i = 0; i < n; i++) {
    auto &o = O[vrstniRed[i]];
    int konec = (i == 0) ? o.rok : min(o.rok, zacNaslednje - 1);
    // Ta obveznost se mora končati najkasneje na dan „konec“.
    o.zacetek = zacNaslednje = konec - O.dolzina + 1; }
  return zacNaslednje > 0;
}
```

3. Lenoba

Naloga pravi, da moramo priti pred poldnem in oditi po njem. Za začetek lahko torej v mislih pobrišemo sodelavce, ki pridejo šele opoldne ali kasneje (kajti oni nas gotovo ne bodo videli priti na delo), in tudi tiste, ki odidejo najkasneje opoldne (kajti oni nas gotovo ne bodo videli oditi).

Recimo, da nam ostane n sodelavcev in da i-ti od njih pride ob času z_i in odide ob času k_i. Kot smo videli v prejšnjem odstavku, so vsi z_i pred poldnevom, vsi k_i pa po poldnevu. Uredimo sodelavce naraščajoče po času prihoda, tako da bo $z_1 \leq z_2 \leq \ldots \leq z_n$.

Recimo našemu času prihoda p. Opazimo lahko, da če p leži na območju $z_i \leq p < z_{i+1}$, nas bo videlo priti prvih i sodelavcev, ostali pa ne; takrat torej ne smemo oditi prej kot ob času $\max\{k_1, \ldots, k_i\} + 1$ (ni pa tudi nobenega razloga, da bi odšli kasneje). Ker je ta čas odhoda enak za vse prihode na območju $z_i \leq p < z_{i+1}$, je najbolje, če pridemo čisto na koncu tega intervala, ob $p = z_{i+1} - 1$ — tako bomo v službi najkrajši čas.

Poseben primer je še možnost, da pridemo pred prvim sodelavcem, torej ob $p < z_1$. Takrat nas nihče ne vidi priti, zato je vseeno, kdo nas vidi oditi; torej lahko odidemo že eno nanosekundo po poldnevu. Ker je ta čas odhoda enak za vse $p < z_1$, je med njimi spet smiselno vzeti najkasnejšega, torej $p = z_1 - 1$.

Zdaj torej poznamo vse možne čase prihoda in za vsakega od njih tudi čas odhoda; med temi možnostmi bomo na koncu seveda vrnili tisto, pri kateri smo v službi najmanj

časa. Zapišimo našo rešitev še v C++:

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

typedef long long int llint;
struct Par { llint prihod, odhod; };

Par Lenoba(vector<Par>& sodelavci)
{
  const llint M = 12'000'000'000LL; // poldne

  // Pripravimo seznam sodelavcev, ki nas lahko ovirajo.
  vector<Par> v; for (auto &P : sodelavci)
    if (P.prihod < M && M < P.odhod) v.push_back(P);

  // Če takih sodelavcev ni, je problem trivialen.
  if (v.empty()) return {M - 1, M + 1};

  // Uredimo jih naraščajoče po času prihoda.
  sort(v.begin(), v.end(), [] (auto x, auto y) { return x.prihod < y.prihod; });
  v.push_back({M, M}); // stražar na koncu zaporedja

  // Preglejmo možnost, da pridemo pred vsemi sodelavci.
  llint odhod = M + 1;
  Par naj = {v[0].prihod - 1, odhod}; // najboljša rešitev doslej

  // Preglejmo še možnosti kasnejšega prihoda.
  for (int i = 0; i + 1 < v.size(); i++)
  {
    // Na naslednjem intervalu nas vidi priti tudi sodelavec i,
    // kar nas dodatno omejuje pri odhodu.
    odhod = max(v[i].odhod + 1, odhod);

    // Če več sodelavcev pride ob istem času, moramo upoštevati vse,
    // da dobimo pravi čas odhoda.
    if (v[i + 1].prihod == v[i].prihod) continue;

    // Sicer je pametno priti tik pred sodelavcem i + 1.
    llint prihod = v[i + 1].prihod - 1;

    // Če je to najboljša rešitev doslej, si jo zapomnimo.
    if (odhod - prihod < naj.odhod - naj.prihod) naj = {prihod, odhod};
  }
  return naj;
}
```

4. Semafor

Imeli bomo dve globalni spremenljivki: stevec, ki odšteva sekunde do spremembe luči, in prizgan, ki jo uporabljamo za izvedbo utripanja in nam pove, ali je bil prikazovalnik v prejšnji sekundi prižgan.

Naš podprogram VsakoSekundo ob vsakem klicu zmanjša števec za 1 (pri tem pazimo, da ne pade pod -1; vrednost -1 bomo uporabljali kot znak, da je čas do naslednje pričakovane spremembe luči že potekel in da mora biti prikazovalnik zdaj prazen) in, dokler je števec nad 99, vsako sekundo tudi obrne vrednost prizgan. Poseben primer je, ko dobimo novo pozitivno vrednost parametra n, takrat pa to shranimo v stevec.

Nato lahko popravimo stanje prikazovalnika takole: če je števec nad 99, mora prikazovalnik ali kazati število 99 ali pa biti prazen, odvisno od spremenljivke prizgan; sicer pa prikažemo kar vrednost števca samo (pri vrednosti -1 bo prikazovalnik prazen, kar je točno to, kar takrat tudi hočemo).

```
int stevec = 0;
bool prizgan = false;

void VsakoSekundo(int n)
{
  if (n >= 0) // Postavimo števec na n. Spremenljivko „prizgan“ uporabljamo le,
```

```
            // ko je števec večji od 99.
      stevec = n, prizgan = true;
    else {
      // Zmanjšajmo števec za 1 (če ni bil že pod 0, kar pomeni ugasnjen prikazovalnik).
      if (stevec >= 0) stevec--;
      // Če je števec nad 99, poskrbimo za utripanje.
      if (stevec > 99) prizgan = ! prizgan; }
    // Če je števec nad 99, prikažimo 99 ali prazen prikazovalnik.
    if (stevec > 99) Prikazi(prizgan ? 99 : -1);
    // Sicer prikažimo trenutno vrednost števca.
    else Prikazi(stevec);
}
```

5. Prelom besedila

Recimo, da lomimo besedilo na vrstice dolžine največ w. Če se je neka vrstica začela z i-tim znakom vhodnega niza, so lahko v njej znaki največ do $(i+w-1)$-vega; najkasneje pri znaku $i+w$ pa bo treba iti v novo vrstico. Toda ne nujno točno pri njem; če je ta znak ne-prvi znak neke besede, bo novo vrstico treba začeti že na začetku te besede; če pa je $(i+w)$-ti znak presledek, bo treba novo vrstico začeti šele na začetku naslednje besede, saj naloga pravi, da presledki pri prelomu vrstice ostanejo v prejšnji vrstici, tudi če štrlijo čez rob stolpca.

Koristno bi bilo torej za vsak znak vedeti, kje se začne beseda, ki ji ta znak pripada (če ni presledek), oz. kje se začne naslednja beseda (če je ta znak presledek). Temu podatku za znak i recimo N_i. Tega ni težko računati z dvema prehodoma po nizu. Najprej gremo od leve proti desni in računamo N_i za ne-presledke: če i ni presledek, potem, če je znak $i-1$ presledek, je $N_i = i$, sicer pa je $N_i = N_{i-1}$. Nato gremo še od desne proti levi in ga računamo za presledke: če je znak i presledek, je $N_i = N_{i+1}$.

S pomočjo tabele N torej vemo, da če lomimo besedilo na vrstice dolžine največ w in se ena vrstica začne pri i, se mora naslednja začeti pri N_{i+w}. Tako lahko hitro skačemo od začetka ene vrstice do začetka naslednje, dokler ne pridemo do konca besedila. Sproti lahko vrstice še štejemo in rezultat na koncu zapišemo v zaporedje (vektor), ki ga na koncu vrnemo.

Poseben primer je, če je širina vrstice w ožja od dolžine najdaljše besede. Takrat se bo včasih zgodilo, da bo $N_{i+w} \leq i$; na to moramo paziti ne le zato, ker moramo takrat vrniti -1, pač pa tudi zato, ker bi se naš siceršnji algoritem za lomljenje vrstic (iz prejšnjega odstavka) zaciklal.

Oglejmo si implementacijo te rešitve v C++:

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

vector<int> PrelomBesedila(const string& s)
{
  int z = s.length();
  vector<int> rezultati(z), N(z);
  // Za vsak ne-presledek izračunajmo začetek njegove besede.
  for (int i = 0; i < z; i++) if (s[i] != ' ')
    N[i] = (i > 0 && s[i - 1] != ' ') ? N[i - 1] : i;
  // Za vsak presledek izračunajmo začetek naslednje besede.
  for (int i = z - 1; i >= 0; i--)
    if (s[i] == ' ') N[i] = (i == z - 1) ? z : N[i + 1];
    else i = N[i]; // skočimo na začetek trenutne besede

  // Za vsako širino izračunajmo število vrstic.
  for (int w = 1; w <= z; w++)
  {
    int stVrstic = 0, zacVrstice = N[0];
    while (zacVrstice < z) {
```

```
        // Določimo začetek naslednje vrstice.
        stVrstic++; int konVrstice = zacVrstice + w;
        if (konVrstice >= z) break;
        int zacNaslednje = N[konVrstice];
        // Pazimo na primer, ko je širina preozka za to besedo.
        if (zacNaslednje <= zacVrstice) { stVrstic = −1; break; }
        zacVrstice = zacNaslednje; }
      rezultati[w − 1] = stVrstic;
    }
    return rezultati;
}
```

Razmislimo še o časovni zahtevnosti te rešitve. Na začetku porabimo $O(z)$ časa za oba prehoda čez vhodni niz, s katerima izračunamo tabelo N. Pri lomljenju besedila imamo z vsako vrstico $O(1)$ dela, tako da za prelom pri širini w porabimo toliko časa, kolikor vrstic nastane, to pa je približno $O(z/w)$ (ker imamo niz dolžine z, v vsaki vrstici pa je prostora za w znakov).[1] Če to seštejemo po vseh w od 1 do z, dobimo $\sum_{w=1}^{z}(z/w) = zH_z$, kjer H_z pomeni z-to harmonično število; zanj velja $H_z \approx \ln z$, torej ima naš postopek časovno zahtevnost $O(z \ln z)$.

[1] Natančnejši razmislek: naj bo x_i število znakov v vrstici i (brez presledkov na koncu) in p_i število presledkov na koncu te vrstice. Če bi v dveh zaporednih vrsticah bilo, skupaj s presledki na koncu prve od teh dveh vrstic, le w ali manj znakov, bi ju lahko združili v eno. Imamo torej $x_i + p_i + x_{i+1} \geq w+1$ za $i = 1, \ldots, h-1$. Če vse to seštejemo in upoštevamo, da je $\sum_{i=1}^{h}(x_i+p_i) = z$ (dolžina celotnega vhodnega niza), dobimo $2z - x_1 - x_h - 2p_h \geq (h-1)/(w+1)$, torej $h \leq 1+(2z-x_1-x_h-2p_h)/(w+1) \leq 1+2z/w$.

REŠITVE NALOG ZA TRETJO SKUPINO

1. Vaje v slogu

Mislimo si neko konkretno dolžino d in recimo za začetek, da nas zanimajo Poldetove trojice iz podnizov te dolžine. Ker je vhodni niz dolg n znakov, se lahko podniz dolžine d začne na enem od indeksov $1, 2, \ldots, n-d+1$. Niso pa nujno vsi ti podnizi različni; recimo, da za vsak različen podniz pripravimo množico — pravzaprav bo še bolj koristno imeti urejen seznam — indeksov, na katerih se začenjajo pojavitve tega podniza. Vsak indeks od 1 do $n-d+1$ pripada natanko eni od teh množic, tako da te množice tvorijo razbitje (particijo) množice $\{1, \ldots, n-d+1\}$. Temu razbitju recimo R_d. Na primer, pri $s =$ `ababaccabab` (primer iz besedila naloge) za $d = 2$ dobimo $R_d = \{\{1,3,8,10\},\{2,4,9\},\{5\},\{6\},\{7\}\}$. Podmnožice, ki tvorijo to razbitje, se nanašajo (po vrsti, kot so tu napisane) na podnize `ab`, `ba`, `ac`, `cc` in `ca`.

Takšno razbitje pride zelo prav pri iskanju ali štetju Poldetovih trojic. Ker tvorijo trojico trije enaki podnizi, se lahko ukvarjamo z vsako množico v razbitju posebej. Recimo torej, da gledamo množico $\{t_1, t_2, \ldots, t_\ell\}$, ki nam pove, da se neki podniz dolžine d pojavlja ℓ-krat v nizu s, in sicer se njegove pojavitve začenjajo na indeksih $t_1, t_2, \ldots, t_\ell$. Recimo še, da so ti indeksi urejeni naraščajoče, torej je $t_1 < t_2 < \ldots < t_\ell$.

Če nas zanima, ali sploh obstaja kakšna Poldetova trojica, temelječa na tem podnizu, moramo torej le preveriti, ali se kakšne tri od teh pojavitev ne prekrivajo. Če se sploh kakšni dve od njih ne prekrivata, sta to gotovo prva in zadnja; in če se res ne, moramo potem le še preveriti, ali se kakšna od vmesnih pojavitev ne prekriva z njima, torej ali za kak t_i (za $1 < i < \ell$) velja $t_1 + d \le t_i$ in $t_i + d \le t_\ell$. To nam vzame $O(\ell)$ časa, kjer je ℓ število pojavitev tega podniza; in ker imajo vsi podnizi dolžine d skupaj $n-d+1$ pojavitev, nam preverjanje, ali kakšna trojica pri tem d obstaja, vzame $O(n)$ časa (če imamo razbitje že pripravljeno).

Podobno, le za odtenek bolj zapleteno, je štetje trojic. Pri vsakem indeksu t_j se lahko vprašamo: če bi hoteli vzeti tisto pojavitev našega podniza, ki se začne na indeksu t_j, kot srednji člen neke Poldetove trojice, na koliko načinov bi si lahko izbrali levi člen t_i in desni člen t_k tako, da se ne bi prekrivala s t_j? Zanima nas torej, do katerega i velja $t_i + d \le t_j$ in od katerega k naprej velja $t_j + d \le t_k$. Ko počasi povečujemo j, se počasi povečujeta tudi največji primerni i in najmanjši primerni k, zato je dovolj, če gremo v zanki enkrat po zaporedju $t_1, \ldots, t_\ell$ in sproti povečujemo vse tri števce:

$N := 0$; (* *tu bomo izračunali število trojic* *)
$i := 0$; $k := 1$;
for $j := 1$ **to** ℓ:
  while $t_{i+1} + d \le t_j$ **do** $i := i + 1$;
  while $k \le n$ **and** $t_j + d \ge t_k$ **do** $k := k + 1$;
  (* *Če za srednji člen trojice vzamemo pojavitev, ki se začne na* t_j*,*
    se mora levi člen začeti na enem od indeksov $t_1, \ldots, t_i$*,*
    desni člen pa na enem od indeksov $t_k, \ldots, t_n$*.* *)
  $N := N + i \cdot (n - k + 1)$;

Na koncu tega postopka imamo v N število vseh trojic, temelječih na tistem podnizu, na katerega se nanaša množica $\{t_1, \ldots, t_\ell\}$. Pravzaprav naloga pravi, da bomo morali izpisati ostanek po deljenju N z 1 000 037; lahko torej vsakič, ko v zadnji vrstici gornjega postopka povečamo N, obdržimo le ostanek po deljenju nove vrednosti z 1 000 037, lahko pa tudi računamo N v 64-bitni spremenljivki in ostanek po deljenju izračunamo šele na koncu, ob izpisu. (V nizu dolžine n je lahko največ $O(n^3)$ trojic, temelječih na podnizih dolžine d.)

Pri naši nalogi nas bo zanimal največji d, pri katerem sploh še obstaja kakšna Poldetova trojica. Opazimo lahko, da če obstaja neka trojica iz podnizov dolžine d, se v njej skrivajo tudi trojice iz podnizov dolžine $d-1$, $d-2$ in tako naprej. Zato lahko največji d poiščemo z bisekcijo: pri $d = 0$ trojica gotovo obstaja (če si jo predstavljamo iz treh praznih nizov), pri $d = \lfloor n/3 \rfloor + 1$ gotovo ne obstaja (ker je niz s prekratek, da bi v njem sploh lahko bili trije neprekrivajoči se podnizi dolžine d), vmes pa bomo najmanjši d iskali z bisekcijo.

Razmisliti pa moramo še o tem, kako za nek konkreten d sploh pripraviti razbitje R_d. Pri $d = 1$ je stvar enostavna; podnizi so tu kar posamezne črke, zato lahko za vsako črko abecede pripravimo po eno množico (pravzaprav urejen seznam) vseh indeksov, na katerih se pojavlja ta črka. Za daljše podnize lahko razmišljamo takole: recimo, da nas zanimajo podnizi dolžine d; zapišimo d kot vsoto dveh manjših dolžin, $d = d_1+d_2$. Potem lahko tudi vsak podniz t dolžine d zapišemo kot stik dveh krajših podnizov, $t = t_1t_2$, kjer je t_1 dolg d_1 znakov, t_2 pa d_2 znakov. Pojavitve podniza t se začenjajo natanko na tistih indeksih i, za katere velja, da se na i začenja tudi neka pojavitev podniza t_1 in da se poleg tega na $i+d_1$ začenja neka pojavitev podniza t_2. Začnemo lahko torej z množico indeksov, na katerih se pojavlja t_1 (ta množica je del razbitja R_{d_1}), in jo razdrobimo na manjše podmnožice tako, da za vsak indeks i v tej množici pogledamo, kateri podniz dolžine d_2 se pojavlja na indeksu $i+d_1$ (z drugimi besedami: pogledamo, kateri množici v razbitju R_{d_2} pripada indeks $i + d_1$). Tako nam množica pojavitev niza t_1 razpade na podmnožice, ki predstavljajo različne take podnize dolžine d, ki se začnejo na t_1. Ko naredimo to za vse množice iz R_{d_1}, nam sčasoma nastane razbitje R_d. Da lahko to počnemo učinkovito, je koristno imeti vsako razbitje predstavljeno ne le s seznamom urejenih seznamov (ki predstavljajo posamezne množice v razbitju), ampak tudi s tabelo, ki za vsak indeks od 1 do $n - d + 1$ pove, kateri podmnožici v razbitju pripada; recimo tej tabeli $\tilde{R}_d$; potem moramo le za vsak indeks i vzeti par $(\tilde{R}_{d_1}[i], \tilde{R}_{d_2}[i + d_1])$ in kjer nastanejo enaki pari, dodamo pripadajoče i-je v isto podmnožico nastajajočega R_d. Za preverjanje, kje nastanejo enaki pari, lahko uporabimo razpršeno tabelo ali pa urejanje, npr. urejanje s štetjem, tako da bomo imeli z vsakim parom v povprečju le $O(1)$ dela in bomo za izračun R_d iz R_{d_1} in R_{d_2} porabili $O(n)$ časa. Drobna izboljšava, ki tudi ne škodi, je, da pri pripravi razbitij R_d podnize z manj kot tremi pojavitvami (torej množice z manj kot tremi elementi) kar sproti zavržemo, saj iz njih gotovo ne bo nastala nobena Poldetova trojica, pa tudi pri nadaljnjem drobljenju (ko bomo računali razbitja za večje d) bodo iz njih nastale prav tako premajhne podmnožice.

S tem postopkom lahko iz R_1 dobimo R_2, nato R_4, R_8 in tako naprej za dolžine, ki so potence števila 2. Tudi pri bisekciji je zato koristno paziti na to, da je širina intervala d-jev, ki so pri bisekciji še aktualni, vedno potenca števila 2, tako da bomo lahko enostavno v $O(n)$ časa izračunali razbitje za dolžino na sredi intervala. Zato je koristno, če na začetku zgornjo mejo namesto na $\lfloor n/3 \rfloor + 1$, kot smo pisali zgoraj, postavimo na najmanjšo potenco števila 2, ki je večja ali enaka $\lfloor n/3 \rfloor + 1$. Bisekcija bo tako potrebovala $O(\log n)$ korakov, v vsakem pa porabimo $O(n)$ časa za izračun novega razbitja in za to, da ga pregledamo, če je v njem kaj trojic (oz. koliko jih je). Tudi izračun R_1, R_2, R_4, R_8 itd. na začetku nam vzame $O(n \log n)$ časa, tako da je časovna zahtevnost celotne rešitve $O(n \log n)$.

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

int n; string s; // Vhodni podatki.

struct Trojica { int i, j, k; };

struct Razbitje
{
  int d;                        // dolžina podnizov pri tem razbitju
  vector<vector<int>> m;        // množice
  vector<int> p;                // pripadnost; x je element m[p[x]]

  void Init1() // Pripravi razbitje za d = 1.
  {
    d = 1; m.resize(26); p.resize(n);
    for (int i = 0; i < n; i++) {
      p[i] = s[i] - 'a'; m[p[i]].push_back(i); }
  }
```

```
  // Iz R1 in R2 izračuna razbitje za d = R1.d + R2.d.
  void Zdruzi(const Razbitje &R1, const Razbitje &R2)
  {
    d = R1.d + R2.d; p = vector<int>(n - d + 1, -1); m.clear(); m.reserve(n - d + 1);
    unordered_map<long long, int> h;
    for (int a1 = 0; a1 < R1.m.size(); a1++) if (R1.m[a1].size() >= 3) for (int i : R1.m[a1])
    {
      if (i + d > n) continue;
      int a2 = R2.p[i + R1.d]; if (a2 < 0 || R2.m[a2].size() < 3) continue;
      // Indeks i predstavimo s parom (R1.p[i], R2.p[i + R1.d]) — no, pravzaprav
      // kar s številom R1.p[i] * n + R2.p[i + R1.d]. Če ima več indeksov
      // enak par, predstavljajo enak podniz dolžine d in jih moramo torej
      // dodati v isti seznam znotraj m. Te pare hranimo kot ključe
      // v h, kot spremljevalno vrednost pa indeks ustreznega seznama v m.
      auto [it, nov] = h.emplace(a1 * (long long) n + R2.p[i + R1.d], -1);
      if (nov) { it->second = m.size(); m.push_back({}); }
      p[i] = it->second; m[p[i]].push_back(i);
    }
    // Podnize z manj kot tremi pojavitvami lahko ignoriramo.
    for (auto &v : m) if (v.size() < 3) { for (int i : v) p[i] = -1; v.clear(); }
  }

  Trojica PrimerTrojice() const // Vrne primer trojice, če ta obstaja.
  {
    for (const auto &v : m) { // Preglejmo vse podnize dolžine d.
      // Če ima podniz manj kot tri pojavitve, iz njega ne bo trojic.
      int vd = v.size(); if (vd < 3) continue;
      // Sicer uporabimo prvo in zadnjo pojavitev.
      int i = v[0], k = v[vd - 1];
      // Poglejmo, če je vmes še kakšna, ki se ne prekriva z njima.
      for (int j : v) if (i + d <= j && j + d <= k) return {i, j, k}; }
    return {-1, -1, -1};
  }
};

int main()
{
  // Preberimo vhodne podatke.
  cin >> n >> s;
  // Pripravimo razbitja za d-je, ki so potence števila 2. Potrebujemo jih do
  // takega 2^g, dokler d = 1 + 2^g še vedno ustreza pogoju 3d ≤ n.
  int lgN3 = 0; while ((3 << lgN3) < n) lgN3++;
  vector<Razbitje> RP2(lgN3);
  RP2[0].Init1(); for (int i = 0; i < lgN3 - 1; i++) RP2[i + 1].Zdruzi(RP2[i], RP2[i]);
  // Z bisekcijo poiščimo največji d, pri katerem še obstaja kakšna trojica.
  int L = 1; Razbitje RL = RP2[0];
  for (int g = lgN3; g > 0; g--)
  {
    // Tu velja, da obstaja trojica z d = L, ne obstaja pa taka z d = L + 2^g.
    int M = L + (1 << (g - 1));
    Razbitje RM; RM.Zdruzi(RL, RP2[g - 1]);
    if (RM.PrimerTrojice().i >= 0) { L = M; RL = move(RM); }
  }
  // Izpišimo eno od najdaljših trojic.
  auto T = RL.PrimerTrojice();
  cout << (T.i + 1) << " " << (T.j + 1) << " " << (T.k + 1) << " " << L << endl;
  // Preštejmo trojice dolžine L.
  int stTrojic = 0;
  for (const auto &v : RL.m)
  {
```

```
      int vd = v.size(), pi = 0, pk = 0; if (vd < 3) continue;
      for (int j : v)
      {
        while (pi < vd && v[pi] + L <= j) pi++;
        while (pk < vd && v[pk] < j + L) pk++;
        // Pojavitve, ki se začnejo na v[0], ..., v[pi − 1], ležijo v celoti
        // levo od tiste, ki se začne na j; tiste pa, ki se začnejo na
        // v[pk], ..., v[vd − 1], ležijo v celoti desno od nje.
        stTrojic = (stTrojic + pi * (long long) (vd − pk)) % 1000037;
      }
    }
    cout << stTrojic << endl; return 0;
}
```

2. Zamik

Najprej opomba o notaciji: v tej rešitvi bomo zapis, ki spominja na potence, uporabljali za zaporedja več enakih znakov; tako na primer $\mathtt{0}^a\mathtt{1}^b$ pomeni niz, v katerem je naprej a zaporednih ničel in nato b zaporednih enic; kot običajno, če eksponent manjka, si mislimo, da je enak 1.

Recimo zdaj za začetek, da bi pripravili niz samih ničel, le z eno enico na koncu: $\mathtt{0}^{n-1}\mathtt{1}$. Hitro lahko opazimo, da s takim nizom ne moremo ugotoviti, ali ga je sistem prezrcalil ali ne: če ga zamakne za eno mesto v desno in ga ne prezrcali, je rezultat enak, kot če ga zamakne za 0 mest in ga prezrcali. Na enako težavo naletimo, če imamo eno daljšo skupino enic, $\mathtt{0}^a\mathtt{1}^b$; ali pa če imamo dve enako dolgi skupini enic, ločeni z ničlo: $\mathtt{0}^a\mathtt{1}^b\mathtt{0}\mathtt{1}^b$.

Šele če imamo dve različno dolgi skupini enic, lahko zaznamo zrcaljenje: če imamo na primer niz $\mathtt{0}^{n-4}\mathtt{1011}$, lahko zrcaljenje prepoznamo po tem, da bo po njem v nizu nastopal podniz `1101` namesto `1011`; zamik pa lahko določimo iz tega, kje v nizu zdaj eden ali drugi od teh dveh podnizov stoji.

Naslednja težava je, da moramo znati te enice v nizu najti z zelo malo poizvedbami, to pa bo težko, če predstavljajo tako kratek del celotnega niza. Tako nam lahko pride na misel, da bi imeli dve daljši skupini enic, ločeni z ničlo: $\mathtt{0}^a\mathtt{1}^b\mathtt{0}\mathtt{1}^d$, pri čemer naj bo $b \neq d$. Toda ko bomo s poizvedbami v zamaknjenem (in mogoče prezrcaljenem) nizu našli enice, bomo morali najti tudi tisto ničlo med obema skupinama enic; šele iz položaja te ničle (relativno glede na položaj enic) bomo lahko vedeli, da imamo v novem nizu $\mathtt{1}^b$ pred $\mathtt{1}^d$ (in torej niz ni bil prezrcaljen) ali $\mathtt{1}^d$ pred $\mathtt{1}^b$ (in je torej niz bil prezrcaljen). Najti eno samcato ničlo v dolgem zaporedju enic pa bo spet težko, zato je bolje, če obe skupini enic tudi ločimo z neko daljšo skupino ničel.

Tako smo pri nizu oblike $\mathtt{0}^a\mathtt{1}^b\mathtt{0}^c\mathtt{1}^d$ z $b \neq d$. Paziti pa moramo še na to, da morata biti različno dolgi ne le obe skupini enic, ampak tudi obe skupini ničel: kajti če bi bilo $a = c$, potem (zaradi cikličnega značaja našega niza pri zamikanju) ne bi mogli reči, ali leži $\mathtt{1}^b$ levo ali desno od $\mathtt{1}^d$, torej spet ne bi mogli zaznati, ali je bil niz prezrcaljen ali ne. Dodati moramo torej še pogoj $a \neq c$, ne samo $b \neq d$.

Znotraj teh omejitev je koristno, če so vsi štirje deli približno enako dolgi, torej okrog $n/4$. Ker n ni nujno večkratnik 4, pišimo $n = 4k + r$; vzemimo za začetek $a = b = k - 1$ in $c = d = k + 1$, nato pa prvih r izmed števil a, b, c povečajmo za 1. Tako dobimo niz naslednje oblike:

n	a	b	c	d	niz s
$n = 4k$	$k-1$	$k-1$	$k+1$	$k+1$	$\mathtt{0}^{k-1}\mathtt{1}^{k-1}\mathtt{0}^{k+1}\mathtt{1}^{k+1}$
$n = 4k+1$	k	$k-1$	$k+1$	$k+1$	$\mathtt{0}^{k}\mathtt{1}^{k-1}\mathtt{0}^{k+1}\mathtt{1}^{k+1}$
$n = 4k+2$	k	k	$k+1$	$k+1$	$\mathtt{0}^{k}\mathtt{1}^{k}\mathtt{0}^{k+1}\mathtt{1}^{k+1}$
$n = 4k+3$	k	k	$k+2$	$k+1$	$\mathtt{0}^{k}\mathtt{1}^{k}\mathtt{0}^{k+2}\mathtt{1}^{k+1}$

Da se ta načrt obnese, nobena od skupin ničel ali enic ne sme biti prazna, sicer niz ne bo prave oblike. Pri $n = 4k$ in $4k + 1$ je najkrajša skupina dolga $k - 1$, torej mora biti $k \geq 2$, torej $n \geq 8$; pri $n = 4k + 2$ in $4k + 3$ pa je najkrajša skupina dolga k, zato je

dovolj že $k \geq 1$, torej $n \geq 6$. Tako torej vidimo, da naš pristop deluje za vse $n \geq 6$, to pa so ravno vsi, ki lahko nastopijo v naši nalogi.[2]

Razmislimo zdaj o tem, kako točno pri takšnem nizu s čim manj poizvedbami določiti zamik in zrcaljenje. Naš cilj bo, da za dve sosednji skupini znakov, torej eno skupino ničel in eno skupino enic, točno določimo njuno dolžino in položaj. Ko namreč enkrat poznamo to, lahko določimo tudi zamik in ali je prišlo do zrcaljenja. Na primer, če najdemo skupino $\mathtt{0}^c$ in desno od nje skupino $\mathtt{1}^b$, potem vemo, da je bil niz prezrcaljen, saj bi drugače bila $\mathtt{1}^b$ levo od $\mathtt{0}^c$; nato lahko izračunamo, kje sta bili tidve skupini pred zrcaljenjem; iz tega pa lahko, ko njun položaj primerjamo s tistim v začetnem nizu, izračunamo tudi zamik.

Označimo položaje v našem nizu z indeksi od 0 do $n-1$. S prvo poizvedbo poglejmo, ali je na indeksu 0 ničla ali enica. Recimo, da je ničla; za enico bi bil razmislek zelo podoben. Zanimivo bi bilo vedeti, kje je prva enica desno od te ničle; potem bi vedeli, da se tam začenja neka skupina enic, in bi lahko s še eno poizvedbo ugotovili, ali je dolga b enic ali d enic; nato bi pa s še eno poizvedbo ugotovili, ali za njo pride skupina a ničel ali c ničel; potem pa imamo, kot smo ugotovili v prejšnjem odstavku, že vse, kar potrebujemo, da rešimo uganko.

Vprašanje je torej zdaj, koliko ničel še pride desno od tiste na indeksu 0, preden nastopi prva enica. Lahko ni nobene (in je enica že na indeksu 1), lahko pa jih je največ $c-1$ (če se je na indeksu 0 ravno začela skupina c ničel — spomnimo se, da je to daljša od obeh skupin ničel, saj je $a < c$). Med temi c možnostmi lahko pravo poiščemo z bisekcijo, za kar porabimo $\lceil \log_2 c \rceil$ poizvedb. Če prištejemo še eno poizvedbo na začetku (na indeksu 0) in dve na koncu (da določimo dolžino naslednje skupine enic in naslednje skupine ničel), imamo skupaj $3 + \lceil \log_2 c \rceil = \lceil \log_2 8c \rceil$ poizvedb.

Ali je to že najboljša rešitev? Skoraj, ne pa še čisto. Pri nizu dolžine n obstaja $2n$ možnosti glede tega, kaj lahko sistem naredi — n zamikov brez zrcaljenja in še n zamikov, ki jim sledi zrcaljenje. Ker lahko z vsako poizvedbo ločimo med dvema možnostma, lahko z m poizvedbami ločimo med 2^m možnostmi. Če naj se dá z m poizvedbami rešiti našo uganko, mora torej veljati $2^m \geq 2n$, torej je najmanjši primerni m enak $\lceil \log_2 2n \rceil$. Naša rešitev ima $\lceil \log_2 8c \rceil$ poizvedb, kar je, če upoštevamo, da je $c \approx n/4$, res približno enako $\lceil \log_2 2n \rceil$. Toda v resnici je c vedno malo večji od $n/4$, zato se včasih lahko zgodi, da je $\lceil \log_2 8c \rceil > \lceil \log_2 2n \rceil$. Do tega pride, če je na območju od $2n$ do $8c-1$ kakšna potenca števila 2, recimo 2^{t+1}; takrat je $\lceil \log_2 2n \rceil = t+1$ in $\lceil \log_2 8c \rceil = t+2$.

Imamo torej pogoj $2n \leq 2^{t+1} < 8c$, kar lahko predelamo v $n \leq 2^t < 4c$. Če je $r = 0, 1$ ali 2, je $c = k+1$ in naš pogoj je naprej enak $4k + r \leq 2^t < 4k+4$; edina potenca števila 2 na območju od $4k+r$ do $4k+3$ je lahko $4k$, pa še ta je dosegljiv le, če je $r = 0$; in takrat je $4k = n$. Pri $r = 3$ pa je $c = k+2$ in naš pogoj je naprej enak $4k+3 \leq 2^t < 4k+8$, edina potenca števila 2 na območju od $4k+3$ do $4k+7$ pa je lahko $4k+4 = n+1$.

Tako torej vidimo, da do neugodne situacije, ko je $\lceil \log_2 8c \rceil > \lceil \log_2 2n \rceil$, pride le tedaj, ko je n oblike 2^t ali $2^t - 1$. Hitro se lahko prepričamo, da je takrat $r = 0$ ali $r = 3$, v vsakem primeru pa je zato $c = 2^{t-2} + 1$ in $a = c - 2$. Razmislimo o tem, kako lahko v takih primerih število potrebnih poizvedb zmanjšamo za 1. Ko smo ugotavljali, kako dolga je skupina ničel na začetku niza (po tistem, ko ga je sistem zamaknil in mogoče prezrcalil), smo rekli, da moramo z bisekcijo ločiti med c različnimi možnostmi, zato potrebujemo $\lceil \log_2 c \rceil$ poizvedb, kar je naprej enako $\lceil \log_2(2^{t-2}+1) \rceil = t-1$. Teh c možnosti se nanaša na to, ali je desno od začetne ničle še 0, 1, …, $c-2$ ali $c-1$ ničel; zadnji dve lahko v mislih združimo v eno samo, „vsaj $c-2$ ničel“. Zdaj imamo le

[2]Hitro se lahko prepričamo, da pri manjših n problem sploh ni rešljiv. Če hočemo rešiti uganko ne glede na to, kakšen zamik si sistem izbere in ali prezrcali naš niz, mu moramo podati tak niz, pri katerem s temi $2n$ kombinacijami zamika in morebitnega zrcaljenja nastane $2n$ različnih nizov. Jasno pa je, da zamik in zrcaljenje nič ne spremenita števila enic in ničel v nizu. Recimo, da ima naš niz k enic. Takih nizov dolžine n je $\binom{n}{k} = \frac{n!}{k!(n-k)!}$. Če je $\binom{n}{k} < 2n$, je naloga gotovo nerešljiva, saj sploh ne obstaja dovolj različnih nizov dolžine n s k enicami. Hitro se lahko prepričamo, da do $n = 5$ to velja povsod (za $0 \leq k \leq n \leq 5$), razen pri $n = 5$ in $k = 2$ ali 3, kjer je $\binom{5}{2} = \binom{5}{3} = 10$. Tam bi bil problem rešljiv, če bi bilo vseh tistih 10 nizov dolžine 5 s po dvema enicama (in podobno za nize s po tremi enicami) takih, da lahko nastanejo z zamikanjem in zrcaljenjem enega niza. Toda jasno je, da to ni res, saj imajo nekateri eno skupino enic (npr. `00110`), nekateri pa dve ločeni (npr. `01010`), taki nizi pa ne morejo nastati eden iz drugega z zrcaljenjem in zamikanjem.

$c-1$ možnosti in bisekcija med njimi nam vzame le $\lceil \log_2(c-1) \rceil = \lceil \log_2 2^{t-2} \rceil = t-2$ poizvedb, torej eno manj kot prej. Če se izkaže, da je teh ničel od 0 do $c-3$, nadaljujemo enako kot v prvotni različici rešitve. Če pa se izkaže, da je teh ničel vsaj $c-2$, potem že vemo, da imamo tukaj opravka s skupino 0^c in ne 0^a, le tega še ne vemo, kje točno se začne: ali na indeksu 0 (in je potem desno od njega še $c-1$ ničel) ali na indeksu $n-1$ (in se od tam nadaljuje na indeksu 0, desno od tega pa je še $c-2$ ničel). Med tema dvema možnostma lahko ločimo s še eno poizvedbo. Zdaj za eno skupino števk že poznamo njen položaj in dolžino, tako da potrebujemo le še eno poizvedbo, da preverimo, ali desno od nje stoji skupina 1^b ali 1^d, potem pa bomo že vedeli dovolj, da bomo določili zamik in zrcaljenje. Tako smo porabili $t-2$ poizvedb za bisekcijo, eno pred njo in dve za njo, skupaj $t+1$, kar je ravno enako $\lceil \log_2 2n \rceil$, tako kot smo želeli.

Zdaj imamo torej rešitev, ki v vsakem primeru porabi največ $\lceil \log_2 2n \rceil$ poizvedb. Manj od tega, kot smo videli že zgoraj, niti ni razumno pričakovati. Besedilo naloge pravi, da gre lahko n do 1024 in da smemo porabiti največ 11 poizvedb, če hočemo vse točke, to pa je ravno $\log_2(2 \cdot 1024)$, tako da naša rešitev tudi v tem smislu ustreza pričakovanjem naloge.

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

int Poizvedba(int i) { // Pošlje sistemu poizvedbo in vrne njegov odgovor.
  cout << (i + 1) << endl;
  int odgovor; cin >> odgovor; return odgovor; }

int main()
{
  while (true)
  {
    // Preberimo n.
    int n; cin >> n; if (n < 0) return 0;

    // Pripravimo niz.
    int a[4]; for (int i = 0; i < 4; i++) a[i] = (n / 4) - 1 + (i & 2) + (i < (n % 4) ? 1 : 0);
    if (a[0] == a[2]) a[0]--, a[2]++;
    if (a[1] == a[3]) a[1]--, a[3]++;
    cout << string(a[0], '0') << string(a[1], '1')
         << string(a[2], '0') << string(a[3], '1') << endl;

    // Poglejmo, s kakšnimi vrednostmi se zamaknjeni niz začne in do kod segajo.
    // Naj bo b0 vrednost bita na indeksu 0 in recimo, da se ista vrednost pojavlja na
    // indeksih 0, ..., d0' - 1, na indeksu d0' pa je vrednost že drugačna. Namesto d0' bomo
    // v resnici računali d0 = min(d0', a[b0] + 1).
    int b0 = Poizvedba(0);
    int L = 0, D = a[b0] + 1;
    while (D - L > 1)
      // Zdaj vemo, da je L < d0 ≤ D.
      if (int M = (L + D) / 2; Poizvedba(M) == b0) L = M; else D = M;
    int d0 = D;

    // Če smo dobili d0 > a[b0], je to gotovo blok dolžine a[b0 + 2] in ne a[b0].
    int z1, d1, b1; // z = začetek, d = dolžina, b = vrednost bitov v tem bloku
    if (d0 > a[b0])
    {
      // Če se dolžini a[b0] in a[b0 + 2] razlikujeta za 2 (in ne le za 1),
      // imamo zdaj d0 = a[b0] + 1, blok pa je dolg a[b0] + 2 in zato še ne vemo,
      // ali se začne na indeksu 0 ali na indeksu n - 1.
      if (a[b0 + 2] == a[b0] + 2) if (Poizvedba(d0) == b0) d0++;
      d1 = a[b0 + 2]; z1 = (d0 - d1 + n) % n; b1 = b0;
    }
    else
    {
      // Sicer imamo bite z vrednostjo b0 na indeksih od 0 do d0 - 1,
      // pri d0 pa se začne nov blok. Poglejmo, kako dolg je.
```

```
      z1 = d0; b1 = 1 − b0;
      d1 = (Poizvedba(z1 + a[b1]) == b1) ? a[b1 + 2] : a[b1];
    }
    // Zdaj en blok poznamo; poglejmo, kako dolg je naslednji.
    int z2 = (z1 + d1) % n, b2 = 1 − b1;
    int d2 = (Poizvedba(z2 + a[b2]) == b2) ? a[b2 + 2] : a[b2];
    // Ali je bil niz prezrcaljen?
    int i1 = b1 + (d1 == a[b1] ? 0 : 2), i2 = b2 + (d2 == a[b2] ? 0 : 2);
    bool prezrcaljen = (i2 == (i1 + 3) % 4);
    // Če da, poglejmo, kakšen bi bil položaj teh dveh blokov brez zrcaljenja.
    if (prezrcaljen) { z1 = n − 1 − (z1 + d1 − 1) % n; z2 = n − 1 − (z2 + d2 − 1) % n;
      swap(z1, z2); swap(d1, d2); swap(b1, b2); swap(i1, i2); }
    // Kje se je blok i1 začel pred zamikom?
    int p1 = 0; for (int i = 0; i < i1; i++) p1 += a[i];
    int zamik = (z1 − p1 + n) % n;
    // Pošljimo svoj odgovor.
    cout << zamik << " " << (prezrcaljen ? 1 : 0) << endl;
  }
}
```

3. Zlaganje slik

Naloga je primerna za reševanje z dinamičnim programiranjem. Naj bo $f(k)$ najmanjša skupna višina vrstic, v katere je moč zložiti prvih k slik. Ko računamo to funkcijo, lahko razmišljamo takole: v zadnjo od teh vrstic bomo zložili zadnjih nekaj slik, recimo od j do k; potem nam ostane še problem, kako čim bolje zložiti v vrstice prvih $j-1$ slik, za to pa že vemo, da bo skupna višina teh vrstic $f(j-1)$. Ker ne moremo vnaprej vedeti, koliko slik je pametno dati v zadnjo vrstico, bomo morali preizkusiti vse možne j in med tako dobljenimi rešitvami obdržati najboljšo. Dobimo torej zvezo $f(k) = \min_j\{f(j-1) + v(j,k)\}$, kjer $v(j,k)$ predstavlja najnižjo možno višino vrstice, ki jo tvorijo slike od j do k. Z j moramo iti tako daleč navzdol (nazaj po zaporedju slik), dokler je še mogoče zlagati slike od j do k v eno vrstico; ko pa postanejo preširoke za v eno vrstico (ne glede na to, kako jih obračamo), se ustavimo. Funkcijo f bomo računali po naraščajočih k-jih, pri vsakem bomo šli v notranji zanki po padajočih j-jih, ko pa enkrat izračunamo $f(k)$, si to vrednost zapomnimo v neki tabeli, da nam bo kasneje pri roki, ko jo bomo potrebovali.

Vprašanje je zdaj še, kako računati $v(j,k)$, torej najmanjšo možno višino vrstice, v kateri so slike od j do k. Naloga pravi, da so pri polovici testnih primerov vse slike kvadratne; takrat torej ni nobene koristi od tega, da jih obračamo za 90 stopinj, in višina vrstice je preprosto enaka najdaljši stranici vseh slik v njej: $v(j,k) = \max\{h_i : j \le i \le k\}$. Ta maksimum lahko računamo sproti, ko zmanjšujemo j (pri fiksnem k): $v(j,k) = \max\{v(j+1,k), h_j\}$. Tako imamo pri vsakem j le $O(1)$ dela in časovna zahtevnost naše rešitve je v tej obliki $O(n^2)$.

Nekaj več dela pa je s primeri, ko so slike lahko poljubni pravokotniki. Pri i-ti sliki označimo krajšo od obeh stranic z a_i, daljšo pa z b_i. Rekli bomo, da slika *stoji*, če je obrnjena tako, da ima višino b_i in širino a_i, in da *leži*, če ima širino a_i in višino b_i. Kako naj zdaj obrnemo slike od j do k, da bo višina vrstice, v kateri so te slike, čim manjša? Najbolje seveda je, če vse slike ležijo; takrat bo višina najmanjša, vendar bo tudi širina največja, tako da si lahko to privoščimo le, dokler širina takega razporeda ne preseže s.

Ko dodajamo vse več slik v vrstico, utegnemo sčasoma priti v položaj, ko ne morejo več vse ležati, ampak jih moramo nekaj postaviti stoje. Če postavimo sliko i stoje, bo vrstica zaradi tega visoka vsaj b_i. Če je pri neki drugi sliki ℓ daljša stranica krajša od tega, torej $b_\ell \le b_i$, potem se višina vrstice ne bo nič povečala, če postavimo pokonci tudi sliko ℓ, pač pa se bo širina vrstice zaradi tega zmanjšala (ker bo slika ℓ potem v širino merila le a_ℓ namesto b_ℓ). Vidimo torej, da če postavimo pokonci eno sliko, se splača postaviti pokonci tudi vse, katerih daljša stranica je krajša od njene (ali pa enaka njeni). Lahko si predstavljamo, da imamo neko zgornjo mejo v, ki nam pove, da bodo vse slike z $b_i \le v$ stale, vse z $b_i > v$ pa ležale. Uporabiti pa hočemo seveda najmanjši tak v, pri katerem širina tako nastale vrstice še ne preseže s.

Kaj se zgodi, ko gremo z j na $j-1$, torej poskušamo v vrstico spraviti še eno novo sliko več poleg vseh dosedanjih (ki jih imamo še vedno v vrstici)? Ali je mogoče, da bi bil zdaj dovolj dober že kakšen manjši v kot prej? Gotovo ne, kajti pri tem nižjem v bi že slike $j, \ldots, k$ presegle širino s, torej bo ta širina presežena tudi, ko jim bomo dodali še sliko $j-1$, pa ne glede na to, ali bo ta slika pri tem v stala ali ležala. Ko se torej j zmanjšuje in prihajajo v vrstico nove slike, se lahko v (torej višina, do katere slike še stojijo) le povečuje ali pa ostaja enak, ne more pa se zmanjševati. Ko sliko enkrat postavimo pokonci, je (pri kasnejšem zmanjševanju j-ja) ne bomo nikoli več vrnili v ležeči položaj.

To pa pomeni, da nam ni več treba hraniti podrobnih podatkov o slikah, ki že stojijo; vse, kar nas še zanima o njih, bo zajeto v podatek o skupni širini trenutne vrstice. Slike, ki še vedno ležijo, pa je koristno hraniti urejene po b_i. Ko se j zmanjša in pride v vrstico nova slika, pogledamo, če je $b_j \leq v$, da vidimo, če jo moramo dodati med ležeče ali med stoječe; nato pa pogledamo, če je vrstica zdaj preširoka, in počasi spreminjamo ležeče slike v stoječe (pri čemer jih gledamo naraščajoče po b_i), dokler njena širina ne pade na s ali manj. Primerna podatkovna struktura za to je kopica, v kateri naj bodo ležeče slike urejene tako, da je v korenu tista z najmanjšo b_i; pri vsaki sliki pa kot spremljevalni podatek hranimo še razliko $b_i - a_i$, ki nam pove, za koliko se vrstica zoža, če to sliko premaknemo iz ležečega položaja v stoječega.

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

typedef long long int llint;

int main()
{
  // Preberimo vhodne podatke.
  int n; llint s; scanf("%d %lld", &n, &s);
  vector<int> a(n), b(n);
  for (int i = 0; i < n; i++) {
    int w, h; scanf("%d %d", &w, &h);
    a[i] = min(w, h); b[i] = max(w, h); }
  // Rešimo problem po naraščajočem številu slik.
  vector<llint> f(n + 1); f[0] = 0; llint INF = 1;
  for (int k = 0; k < n; k++)
  {
    // Poiščimo najboljšo rešitev za slike 0, ..., k.
    // V zadnjo vrstico bomo poskusili dati slike j, ..., k za različne j.
    llint &naj = f[k + 1] = INF += b[k], sirina = 0; int visina = 0;
    priority_queue<pair<int, int>> lezece;
    for (int j = k; j >= 0; j--)
    {
      // Dodajmo sliko j med ležeče ali stoječe, odvisno od daljše stranice.
      if (b[j] <= visina) sirina += a[j];
      else visina = max(visina, a[j]), lezece.emplace(-b[j], b[j] - a[j]), sirina += b[j];
      // Poglejmo, če je treba kaj ležečih slik postaviti pokonci.
      while (sirina > s && ! lezece.empty()) {
        auto p = lezece.top(); visina = max(visina, -p.first);
        sirina -= p.second; lezece.pop(); }
      // Mogoče so slike že preširoke za eno vrstico, četudi vse stojijo.
      if (sirina > s) break;
      // Če je to najboljša rešitev doslej, si jo zapomnimo.
      naj = min(naj, visina + f[j]);
    }
  }
  printf("%lld\n", f[n]); return 0; // Izpišimo rezultat.
```

```
}
```

Pri dodajanju v kopico smo uporabili $-b_i$ namesto b_i, ker C++ov razred priority_queue v korenu kopice hrani največji element namesto najmanjšega; to bi lahko rešili tudi tako, da bi kot tretji template argument podali greater<pair<int, int>> ali kaj podobnega. Vrednost INF uporabljamo za inicializacijo vrednosti f[k + 1], ker je večja od največje možne skupne višine vseh slik od 0 do k in zato tudi večja od prave vrednosti najboljše rešitve za prvih $k + 1$ slik.

4. Janko in Metka

Razdelimo v mislih sladkarije na štiri množice, A, B, C in D, pri čemer so v A tiste, ki so všeč tako Janku kot Metki, v B tiste, ki so všeč le Janku, v C tiste, ki so všeč le Metki, in v D tiste, ki niso všeč nobenemu od njiju.

Naloga pravi, da moramo izbrati k sladkarij, od tega vsaj x takih, ki so všeč Janku, in vsaj x takih, ki so všeč Metki. Z vidika teh omejitev je vseeno, *katere* sladkarije iz množice A izberemo, pomembno je le, *koliko* jih izberemo; in enako seveda tudi pri B, C in D. Ker pa želimo izbor s čim večjo vsoto vrednosti, je smiselno, da vsako od teh štirih množic uredimo padajoče po vrednosti in iz vsake izberemo prvih nekaj sladkarij (torej tistih z najvišjo vrednostjo).

Recimo zdaj, da iz množice A izberemo natanko p sladkarij (seveda tistih z najvišjo vrednostjo). Te so všeč tako Janku kot Metki; toda če je $p < x$, moramo izbrati nujno vsaj še $x - p$ takih, ki so všeč samo Janku, in $x - p$ takih, ki so všeč samo Metki, sicer naš izbor ne bo ustrezal zahtevam naloge. Pišimo $q := \max\{0, x - p\}$; dodajmo torej v naš izbor najvrednejših q sladkarij iz B in najvrednejših q iz C. (Lahko se izkaže, da v kakšni od množic B in C sploh ni q sladkarij; to pomeni, da smo vzeli premajhen p in z njim do veljavnega izbora sploh ni mogoče priti. Po drugi strani se lahko izkaže, da smo izbrali že preveč sladkarij, torej da je $p + 2q > k$; tudi to je znak, da je naš sedanji p premajhen.)

Tako nam ostane še $|B| - q$ sladkarij iz B, pa $|C| - q$ sladkarij iz C in vse iz D. Tej množici neizbranih sladkarij recimo R. Našemu izboru do velikosti k manjka še $r := k - (p + 2q)$ sladkarij. Vprašanje je torej, kako poceni določiti vsoto vrednosti najdražjih r sladkarij v množici R. (Lahko se tudi izkaže, da je $r > |R|$, torej sploh ni ostalo dovolj sladkarij — to je še en znak, da smo vzeli premajhen p.) Pri tem je dobro imeti v mislih, da smo doslej ves čas razmišljali o eni konkretni vrednosti p, toda v resnici pravega p vnaprej ne poznamo, zato bomo morali preizkusiti vse in si zapomniti najboljšo od tako dobljenih rešitev. Ko počasi povečujemo p, se q počasi zmanjšuje, zato se množica R počasi povečuje (vanjo prihajajo novi elementi); želena velikost izbora r pa pri povečevanju p-ja sprva povečuje (dokler se q še zmanjšuje), kasneje pa zmanjšuje (ko q doseže vrednost 0 in tam ostane).

Množico R lahko predstavimo na primer tako, da vse sladkarije v $B \cup C \cup D$ uredimo padajoče po vrednosti in nad tem seznamom zgradimo polno binarno drevo. Vsako vozlišče naj hrani dve števili: koliko je v R sladkarij iz poddrevesa, ki se začne pri tem vozlišču, ter kakšna je vsota njihovih vrednosti. Na začetku so v vseh vozliščih ničle, ko pa dodamo novo sladkarijo, recimo i, v množico R, povečamo v i-tem listu in vseh njegovih prednikih število sladkarij za 1 in vsoto njihovih vrednosti za c_i. Ko nas potem zanima vsota najvrednejših r sladkarij v množici R, lahko razmišljamo takole: začnimo v korenu drevesa in postavimo v na 0; če je števec sladkarij v levem otroku vsaj r, se premaknimo vanj, sicer pa povečajmo v za vsoto vrednosti sladkarij v levem otroku, zmanjšajmo r za število sladkarij v levem otroku in se premaknimo v desnega otroka. Ko pridemo v list, povečajmo v še za vrednost sladkarije v njem in imamo iskano vsoto. Vsaka operacija na takem drevesu (poizvedba ali dodajanje elementa) nam vzame $O(\log n)$ časa in pri vsaki vrednosti p-ja imamo največ dve dodajanji in eno poizvedbo, tako da nam ta rešitev vzame $O(n \log n)$ časa. (Toliko porabimo tudi za urejanje sladkarij po vrednosti na začetku.)

Še ena možnost je, da R predstavimo z dvema kopicama (prioritetnima vrstama), recimo R_1 in R_2; pri tem naj R_1 hrani najvrednejših r sladkarij in to tako, da je v korenu najmanj vredna med njimi, R_2 pa hrani ostale sladkarije in to tako, da je v korenu najvrednejša med njimi. Poleg tega vzdržujmo tudi vsoto vrednosti vseh sladkarij v R_1

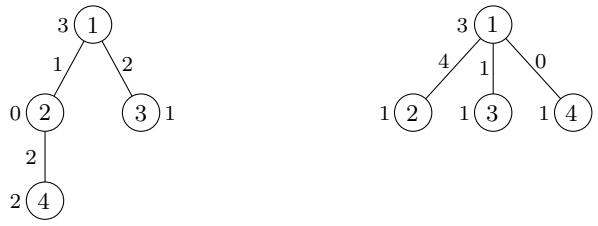
— to je ravno vsota najvrednejših r sladkarij iz R, ki nas zanima za potrebe našega izbora. Ko dodajamo nove elemente v R, to v praksi pomeni, da jih dodamo v R_1, če so vsaj tako vredni kot najmanj vredni element v R_1, sicer pa jih dodamo v R_2; in nato po potrebi premestimo nekaj elementov iz R_1 v R_2 ali obratno, tako da ima R_1 spet natanko r elementov. (Tudi če nismo dodali nobenega elementa, se je mogoče spremenil r in je premeščanje potrebno že zaradi tega.) Tudi tu imamo z vsako operacijo na kopicah $O(\log n)$ dela in konstantno mnogo takih operacij pri vsakem p, tako da ima tudi ta rešitev časovno zahtevnost $O(n \log n)$. Potencialna prednost v primerjavi s prejšnjo rešitvijo je, da ima marsikateri programski jezik primerno implementacijo kopice že v svoji standardni knjižnici.

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

typedef long long int llint;
priority_queue<int, vector<int>, greater<int>> R;
priority_queue<int, vector<int>, less<int>> S;
llint vr; // vsota elementov R-ja

void Prerazporedi(int r) { // poskrbi, da bo v R res točno r elementov
  while (R.size() > r) { int v = R.top(); S.push(v); R.pop(); vr -= v; }
  while (R.size() < r) { int v = S.top(); R.push(v); S.pop(); vr += v; } }

int main()
{
  // Preberimo vhodne podatke.
  int n, k, x; scanf("%d %d %d", &n, &k, &x);
  vector<int> C(n); for (int&ci : C) scanf("%d", &ci);
  vector<bool> DJ(n, false), DM(n, false); // dobri Janku/Metki
  for (int kdo = 0; kdo < 2; kdo++) {
    auto &v = (kdo == 0) ? DJ : DM; int d; scanf("%d", &d);
    while (d-- > 0) { int i; scanf("%d", &i); v[i - 1] = true; } }
  // Pripravimo sezname vrednosti sladkarij, ki so dobre Janku, Metki, obema ali nikomur.
  vector<int> D[4]; // dobri nikomur / samo Metki / samo Janku / obema
  for (int i = 0; i < n; i++) D[(DJ[i] ? 2 : 0) + (DM[i] ? 1 : 0)].push_back(C[i]);
  // Uredimo jih padajoče po ceni.
  for (auto &v : D) sort(v.begin(), v.end(), greater<int>());
  // Recimo, da uporabimo natanko p takih sladkarij, ki so všeč obema.
  // Naj bo vp vsota njihovih vrednosti. Potem moramo uporabiti še
  // q := max(0, x - p) takih, ki so všeč le Janku, in prav toliko
  // takih, ki so všeč Metki; naj bo vq vsota njihovih vrednosti.
  // Tiste, ki jih še nismo izbrali in niso všeč obema, bomo hranili v
  // kopicah R in S, in sicer največjih r v kopici R (njihova vsota bo vr),
  // ostale pa v S. Za r moramo vzeti r := k - p - 2q, da dobimo izbor k sladkarij.
  // — Za začetek pripravimo vse za p = 0.
  llint vp = 0, vq = 0, naj = -1; vr = 0; int q = x;
  for (int j = 0; j <= 2; j++) for (int i = 0; i < D[j].size(); i++)
    if (i >= (j == 0 ? 0 : q)) S.push(D[j][i]);
    else if (j > 0) vq += D[j][i];
  // Preglejmo vse možne p.
  for (int p = 0; p <= D[3].size() && p <= k; p++)
  {
    int qPrej = q; q = max(0, x - p); if (p > 0) vp += D[3][p - 1];
    // Ko se q zmanjša za 1, izpadeta dva elementa iz vsote vq
    // in se premakneta v kopici R oz. S.
    if (q < qPrej) for (int j = 1; j <= 2; j++) if (q < D[j].size()) {
      int v = D[j][q]; vq -= v;
      if (R.empty() || v < R.top()) S.push(v);
```

```
      else { vr += v; R.push(v); } }
    // Če je q prevelik (ker je p premajhen), izbor pri tem p sploh ni mogoč.
    if (q > D[1].size() || q > D[2].size()) continue;
    // Izmed še neizbranih sladkarij, ki niso všeč obema, moramo izbrati največjih r.
    int r = k - p - 2 * q;
    // Če jih sploh ni toliko, je p premajhen in izbor pri njem ni mogoč.
    if (r < 0 || r > R.size() + S.size()) continue;
    // Poskrbimo, da bo v R res točno r sladkarij (ostale pa v S).
    Prerazporedi(r);
    // Najboljšo rešitev si zapomnimo.
    naj = max(naj, vp + vq + vr);
  }
  // Izpišimo rezultat.
  printf("%lld\n", naj); return 0;
}
```

5. Ključavničarstvo

Ker je vsaka soba prek hodnikov povezana z vsako drugo, hodniki pa ne tvorijo ciklov, ima tloris stavbe obliko drevesa, torej povezanega acikličnega neusmerjenega grafa, v katerem točke predstavljajo sobe, povezave pa hodnike med njimi. Lažje si ga predstavljamo, če ga narišemo v hierarhični obliki: točko 1, v kateri naš igralec začne svoj sprehod, vzemimo za koren drevesa, vse ostale povezave pa naj visijo navzdol od tam. Za primera iz besedila naloge na primer dobimo (ob vsaki točki smo napisali število ključev v njej, ob vsaki povezavi pa število ključavnic na njej):



Če imamo povezavo med u in v in je u bližje korenu kot v, temu z drugimi besedami rečemo, da je v *otrok* točke u, ta pa je *starš* točke v. Ker naš igralec začne svoj sprehod v korenu, vemo, da preden prvič obišče točko v, mora gotovo že obiskati tudi njenega starša.

Ko pride igralec prvič v točko u, pobere k_u ključev v njej. Spomnimo se, da nas zanima scenarij, pri katerem on porabi čim več ključev, po možnosti toliko, da mu jih zmanjka, še preden uspe obiskati vse točke v grafu. Recimo, da je povezava od u do nekega njegovega otroka v zaklenjena z ℓ ključavnicami in da je $\ell > 1$. V tem primeru lahko naš igralec, ko prvič pride v u, takoj porabi $\ell-1$ ključev, da odklene na tej povezavi vse ključavnice razen ene; s tem je svoje število ključev nekoliko zmanjšal, povezava pa je še vedno zaklenjena in s tem neprehodna. Zato lahko v mislih kar zmanjšamo k_u (število ključev v u) za $\ell - 1$, nato pa število ključavnic na prej omenjeni povezavi zmanjšamo z ℓ na 1. Po tej spremembi imajo vse povezave 0 ali 1 ključavnico, število ključev v posamezni točki pa je lahko po novem tudi negativno (kar pomeni, da lahko takoj po prihodu v tisto točko z odklepanjem ključavnic na povezavah do otrok porabimo več ključev, kot smo jih v tisti točki dobili).

Za poddrevo, ki se začne pri točki u (torej obsega to točko in vse njene potomce), naj bo $f(u)$ največja možna razlika med številom odklenjenih ključavnic in številom pobranih ključev pri sprehajanju po tem poddrevesu (z začetkom v točki u), če si primerno izberemo, katere dele poddrevesa bi obiskali. Če je $f(u) > 0$, to pomeni, da lahko s sprehajanjem po tem poddrevesu porabimo več ključev (za odklepanje ključavnic), kot jih pridobimo z obiski sob v njem; če pa je $f(u) < 0$, to pomeni, da v tem poddrevesu neizogibno poberemo več ključev, kot jih porabimo.

Funkcijo f lahko računamo od spodaj navzgor po drevesu: preden jo izračunamo za točko u, jo je koristno poznati za njene otroke. Recimo, da ima u otroke $v_1, v_2, \ldots, v_k$

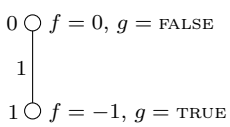
in da do otroka v_i vodi povezava z $\ell_i \in \{0,1\}$ ključavnicami. Pri izračunu $f(u)$ lahko razmišljamo takole: za začetek ob vstopu v u pridobimo k_u ključev, torej za začetek postavimo $f(u)$ na $-k_u$; nato pa moramo za vsakega otroka v_i razmisliti, ali se ga splača obiskati ali ne. Pravzaprav, če povezava od u do v_i nima nobene ključavnice ($\ell_i = 0$), potem tistega otroka naš igralec neizogibno lahko obišče, če je obiskal tudi u (z drugimi besedami, ni takega scenarija, v katerem bi on obiskal u, točka v_i pa bi mu bila nedosegljiva); in ko takega otroka obišče, si lahko (s sprehajanjem po njegovem poddrevesu) število ključev zmanjša za $f(v_i)$, ne pa za več kot toliko. Za take otroke moramo torej nujno prišteti $f(v_i)$ k vrednosti $f(u)$.

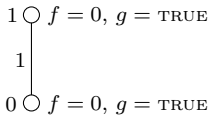
Pri tistih otrocih v_i pa, do katerih vodi povezava s ključavnico ($\ell_i = 1$), se lahko odločimo: lahko v otroka v_i ne gremo, s čimer se nam število ključev ne spremeni, lahko pa vanj gremo, ob čemer porabimo en ključ in nato s sprehajanjem po v_i-jevem poddrevesu zmanjšamo število svojih ključev še za $f(v_i)$. Od teh dveh možnosti je boljša tista, ki nam število ključev najbolj zmanjša. Za take otroke moramo torej prišteti k vrednosti $f(u)$ maksimum $\max\{0, 1 + f(v_i)\}$.

Na koncu nas bo zanimalo, kakšna je vrednost funkcije f v korenu, torej v točki 1. Spomnimo se, da naš igralec začne sprehod v tej točki z 0 ključi (preden pobere tistih k_1 ključev, ki ga čakajo v tej točki). Če je $f(1) \geq 0$, to pomeni, da je mogoč tak sprehod, pri katerem mu število ključev sčasoma pade nazaj na začetno vrednost, torej na 0 (če je $f(1) = 0$) ali celo pod 0 (če je $f(1) > 0$). Takrat pa igralec ne more odkleniti nobene nove povezave več, saj je na vsaki zaklenjeni povezavi vsaj ena ključavnica, torej ne more obiskati celotnega drevesa. (Povezave, ki nimajo ključavnic, smo upoštevali že pri izračunu funkcije f, saj smo takrat rekli, da je otroka, do katerega vodi povezava brez ključavnic, vedno nujno treba obiskati; spremembe v številu ključev zaradi obiskovanja takih otrok in njihovih poddreves so torej že zajete v vrednosti funkcije f.) Po drugi strani pa, če je $f(1) < 0$, to pomeni, da se po drevesu ni mogoče sprehajati tako, da bi porabili več ključev, kot jih poberemo, torej scenarij, po kakršnem sprašuje naloga, ne obstaja.

V primeru, ko je $f(1) = 0$, nastopi pravzaprav še ena drobna komplikacija. Mogoče je, da število ključev pade nazaj na 0 šele takrat, ko že obiščemo celotno drevo. Takrat scenarij, po karkšnem sprašuje naloga, tudi ne obstaja, čeprav je $f(1) = 0$. Zato si moramo pri izračunu funkcije $f(u)$ zapomniti še to, ali smo njeno vrednost dobili tako, da smo obiskali vse točke v poddrevesu z začetkom pri u ali ne. Recimo tej vrednosti $g(u)$. Izračunamo jo takole: če u nima otrok, je $g(u) = \text{TRUE}$; če ima otroke in smo pri izračunu $f(u)$ kakšnega od njih preskočili, je $g(u) = \text{FALSE}$; sicer pa je $g(u)$ konjunkcija vrednosti $g(v_i)$ po vseh u-jevih otrocih v_i.

Primer te težave kažeta naslednji sliki. Koren je v obeh primerih zgornja točka in vrednost funkcije f v njej je 0. Pri levi stavbi se naš igralec lahko zatakne (celo mora se, saj nima ključa, da bi odklenil ključavnico na povezavi), pri desni pa se ne more (saj v korenu dobi ključ, s katerim lahko odklene povezavo in tako obišče še drugo sobo):





Ta razmislek se nekoliko zaplete le v primeru, ko do nekega otroka v_i pelje povezava z eno ključavnico in je $f(v_i) = -1$ (do tega pride na primer pri levi stavbi na gornji sliki); če torej gremo vanj, najprej porabimo en ključ, da odklenemo ključavnico na povezavi do njega, nato pa se nam število ključev pri sprehodu po v_i-jevem poddrevesu poveča za 1, zato smo z vidika izračuna vrednosti $f(u)$ na istem, kot če ga sploh ne bi obiskali; z vidika izračuna $g(u)$ pa ne, kajti če otroka v_i preskočimo, bo $g(u)$ takoj postala FALSE, sicer pa ne nujno. Kaj naj naredimo? Razmišljati moramo takole: če to, ali obiščemo otroka v_i ali ne, nič ne vpliva na vrednost $f(u)$, potem tudi ne vpliva nič na vrednost $f(1)$ v korenu. In to, ali smo nekega otroka obiskali ali ne, je pomembno le v primeru, ko je $f(1) = 0$, kajti le takrat nam je pomembno, ali smo obiskali celotno drevo (preden nam je število ključev padlo na 0) ali ne. Tedaj pa — ker, kot smo pravkar videli, ostane vrednost $f(1)$ enaka ne glede na to, ali v_i obiščemo ali ne — lahko torej $f(1) = 0$ dosežemo tudi tako, da otroka v_i ne obiščemo; torej je mogoč tak sprehod, ki obišče

nekaj točk (mogoče tudi v_i-jevega starša u, ne pa nujno), ostane na koncu pri 0 ključih, točke v_i pa še ni obiskal in vanjo tudi ne more, ker je na povezavi od u do v_i še vedno zaklenjena ključavnica. V takem primeru je torej pravilni odgovor ta, da scenarij, po kakršnem sprašuje naloga, obstaja; da bomo pri $f(1) = 0$ dali tak odgovor, moramo torej imeti v korenu $g(1) = \text{FALSE}$, to pa lahko zagotovimo le, če pri izračunu $f(u)$ in $g(u)$ štejemo, da smo otroka v_i preskočili, tako da bo $g(u)$ zagotovo FALSE, to pa se bo kasneje preneslo dalje navzgor po drevesu vse do korena.

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

struct Tocka
{
  int stars = -1;                    // starš te točke
  vector<pair<int, int>> sosedje;    // pari ⟨sosed, št. ključavnic na hodniku⟩
  int kljuci;                        // število ključev v tej sobi
  int maxPoraba; bool obisceVse;     // f(u) in g(u) za to točko
};

int main()
{
  int T; scanf("%d", &T);
  while (T-- > 0)
  {
    // Preberimo naslednjo stavbo.
    int n; scanf("%d", &n);
    vector<Tocka> G(n); for (auto &U : G) scanf("%d", &U.kljuci);
    for (int j = 0; j < n - 1; j++) {
      int u, v, l; scanf("%d %d %d", &u, &v, &l); u--; v--;
      G[u].sosedje.emplace_back(v, l);
      G[v].sosedje.emplace_back(u, l); }

    // Preglejmo graf iz korena navzven.
    const int koren = 0; auto &R = G[koren]; R.stars = -1;
    vector<int> q; q.push_back(koren); int glava = 0;
    while (glava < q.size()) {
      int u = q[glava++]; auto &U = G[u];

      for (auto &[v, l] : U.sosedje) if (v != U.stars) {
        auto &V = G[v]; V.stars = u; // Označimo, da je u starš točke v.

        // Na povezavi od starša u do otroka v odklenimo vse ključavnice razen ene
        // in ustrezno zmanjšajmo število ključev v staršu u.
        if (l > 1) { U.kljuci -= l - 1; l = 1; }
        q.push_back(v); } }

    // Rešimo problem od spodaj navzgor.
    for (int i = q.size() - 1; i >= 0; i--) {
      int u = q[i]; auto &U = G[u];

      // Ob vstopu v u poberemo vse ključe, ki nas čakajo v njem.
      U.maxPoraba = -U.kljuci; U.obisceVse = true;
      for (auto [v, l] : U.sosedje) if (v != U.stars) { auto &V = G[v];

        // Otroka obiščemo, če je povezava do njega brez ključavnic ali pa
        // če se nam z obiskom tega otroka število ključev zmanjša. Sicer ga preskočimo.
        if (l > 0 && V.maxPoraba + 1 < 0) { U.obisceVse = false; continue; }
        U.maxPoraba += V.maxPoraba + l; U.obisceVse = U.obisceVse && V.obisceVse; } }

    printf("%s\n", R.maxPoraba > 0 || R.maxPoraba == 0 && ! R.obisceVse ? "da" : "ne");
  }
  return 0;
}
```

Naloge so sestavili: Janko in Metka — Tomaž Hočevar; prepisovanje, metanje na koš — Boris Horvat; obračanje jogija, lenoba, zamik, ključavničarstvo — Vid Kocijan; ne odlašaj na jutri — Samo Kralj; ključ, zobna ščetka — Mark Martinec; prelom besedila — Jure Slak; vesoljske vsote — Jasna Urbančič; semafor, dvojni podnizi, zlaganje slik — Janez Brank.

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