16. tekmovanje ACM v znanju računalništva za srednješolce

27. marca 2021

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(f"{a} + {b} = {a + b}"
```

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

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

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

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

Še isti trije primeri v javi:

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

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

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

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

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

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

16. tekmovanje ACM v znanju računalništva za srednješolce

27. marca 2021

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 tekmovalca diskvalificira, če bi se kasneje izkazalo, da nalog ni reševal sam. 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“. Ker je vgrajeni urejevalnik dokaj preprost in ne omogoča označevanja kode z barvami, predlagamo, da rešitev pripraviš v urejevalniku na svojem računalniku in jo nato prekopiraš v okno spletnega urejevalnika. Naj te ne moti, da se bodo barvne oznake kode pri kopiranju izgubile.

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

Med reševanjem lahko vprašanja za tekmovalno komisijo postavljaš prek zasebnih sporočil na tekmovalnem strežniku (ikona oblačka zgoraj desno), izjemoma pa tudi po elektronski pošti na `rtk-info@ijs.si`. Prek zasebnih sporočil bomo pošiljali tudi morebitna pojasnila in popravke, če bi se izkazalo, da so v besedilu nalog kakšne nejasnosti ali napake. Zato med reševanjem redno preverjaj, če so se pojavila kakšna nova zasebna sporočila.

Če imaš 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. Gesla

Marjan je pozabil geslo za Apple ID in se ga ne spomni. Kot mnogi drugi ljudje ima tudi on nekaj različnih gesel, ki jih ponavadi uporablja za vse možne storitve (Gmail, Facebook, Instagram itd.). Ta gesla vsebujejo samo male črke angleške abecede in številke (na primer: „`iec4oovi`“, „`eipe9thu`“ in podobno); vsako geslo vsebuje vsaj eno črko. Marjan ni prepričan, katero geslo je sprva nameraval uporabiti za Apple ID, spomni pa se, da je geslo moralo biti „varno“, kar pomeni, da je moral Marjan v svojem geslu uporabiti tudi en znak, ki ni črka ali številka in pa vsaj eno veliko črko. Spomni se le še tega, da je nekje znotraj gesla (mogoče celo čisto na začetku ali na koncu) dodal piko in spremenil eno od obstoječih črk v veliko, ne spomni pa se natančno, kje je dodal piko in katero črko je spremenil v veliko. **Napiši podprogram** (funkcijo) MoznaGesla(geslo), ki kot parameter dobi niz geslo z Marjanovim prvotnim geslom iz samih malih črk in številk ter izpiše vse možne nize, ki bi lahko bili Marjanovo geslo za Apple ID. (Na primer: iz `eipe9thu` lahko dobimo `eip.E9thu` ali `e.ipe9tHu` ali `.eiPe9thu` ali še marsikaj drugega.)

2. Marsovci

Vsak marsovec se specializira za natanko 5 opravil. Če je za izvedbo naloge treba več kot 5 opravil, se povežejo v skupine. Imamo skupino m marsovcev in za vsakega marsovca imamo podatke o tem, katerih 5 opravil zna opravljati. Opravila so predstavljena s celimi števili od 1 do 100. **Napiši program**, ki za podano skupino marsovcev ugotovi, ali so vsa tista opravila, ki jih opravlja vsaj en marsovec v skupini, približno enako zastopana; natančneje povedano, preveriti moraš, ali se število marsovcev, ki so specializirani za posamezno opravilo, od enega opravila do drugega razlikuje največ za 1. Podatke naj tvoj program prebere s standardnega vhoda ali pa iz datoteke `marsovci.txt` (karkoli ti je lažje); v prvi vrstici je število marsovcev m, v vsaki od naslednjih m vrstic pa je po 5 števil, ki povedo, katera opravila obvlada posamezni marsovec. Če so vsa opravila približno enako zastopana, naj izpiše `da`, sicer pa `ne`.

Primer vhodnih podatkov:

```
4
75 12 96 57 28
96 28 12 75 9
96 9 57 28 75
12 57 9 28 75
```

Pripadajoči izhod:

```
da
```

Še en primer vhoda:

```
4
75 12 96 57 28
96 28 12 75 9
96 9 57 28 75
12 57 96 28 75
```

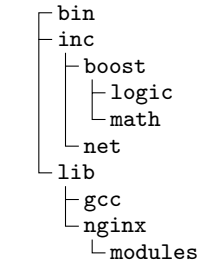
Pripadajoči izhod:

```
ne
```

Komentar: v prvem primeru se vsako opravilo pojavlja pri treh ali štirih marsovcih, zato so približno enakomerno zastopana. V drugem primeru pa se opravilo 9 pojavlja le pri dveh marsovcih, nekatera opravila pa pri štirih, zato niso približno enakomerno zastopana (glede na definicijo iz besedila naloge).

3. Rekonstrukcija poti

Direktorije oziroma mape na disku si pogosto predstavljamo kot zložene v drevesasto hierarhično strukturo, na primer takole:



Če bi hoteli takšno drevo direktorijev predstaviti samo z besedilom, brez črt, nam lahko prideta na misel naslednja dva načina:

(1) Pri vsakem imenu direktorija lahko zapišemo njegovo globino v drevesu. Pri zgornjem drevesu bi tako dobili:

```
bin 1
inc 1
boost 2
logic 3
math 3
net 2
lib 1
gcc 2
nginx 2
modules 3
```

(2) Lahko pa za vsak direktorij izpišemo polno pot od korena do njega. Pri zgornjem drevesu bi tako dobili:

```
/bin
/inc
/inc/boost
/inc/boost/logic
/inc/boost/math
/inc/net
/lib
/lib/gcc
/lib/nginx
/lib/nginx/modules
```

Napiši program, ki prebere predstavitev drevesa v prvi obliki (torej z imeni direktorijev in njihovimi globinami v drevesu) in ga izpiše v drugi obliki (torej s polnimi potmi). Delovati mora seveda za poljuben vhod, ne le za tistega iz gornjega primera. Če v vhodnem seznamu manjka kakšen direktorij in zato v nekem trenutku poti ni več mogoče rekonstruirati, naj program izpiše „`Napaka!`“ in se neha izvajati. Podatke lahko bereš s standardnega vhoda in pišeš na standardni izhod ali pa bereš iz datoteke `vhod.txt` in pišeš na `izhod.txt` (karkoli ti je lažje). Imena direktorijev so sestavljena le iz črk, brez presledkov ali kakšnih drugih posebnih znakov.

Primer vhoda, kjer rekonstrukcija ni mogoča:

```
abc 1
def 3
```

Tvoj program bi moral tu izpisati:

```
/abc
Napaka!
```

4. Kako dobri so virusni testi?

Prebivalce testiramo na okužbo z virusom covida s hitrimi testi in s testi PCR. Prvi so, kot že ime pove, hitri (in poceni) in dajo rezultat v nekaj minutah, so pa nezanesljivi, drugi, tako imenovani testi PCR, pa so zanesljivejši, vendar precej dražji in je na rezultate treba čakati en dan.

Da bi ugotovili kvaliteto hitrih testov, občasno testiramo skupino ljudi hkrati z obema vrstama testov, hitrimi in testi PCR. Rezultate lahko predstavimo z dvema enako dolgima nizoma znakov, pri čemer i-ti znak prvega niza pove rezultat hitrega testa na i-tem pacientu (`1` = okužen in `0` = neokužen), i-ti znak drugega niza pa rezultat testa PCR na istem pacientu (enako `1` = okužen in `0` = neokužen). Primerjava obeh nizov nam pokaže kvaliteto hitrih testov, ki smo jih pri tem poskusu uporabili.

Napiši podprogram (funkcijo) `Primerjava(s, t, n)`, ki kot parametra dobi dva enako dolga niza `s` (rezultati hitrih testov) in `t` (rezultati testov PCR) in ugotovi, pri katerih n zaporednih pacientih je bilo največ razhajanj med hitrimi in testi PCR. Tvoja funkcija naj vrne indeks, na katerem se začne ta skupina n zaporednih pacientov; če je takšnih skupin več, vrni indeks najbolj leve od njih (tiste z najmanjšim začetnim indeksom). Tvoja rešitev naj bo čim bolj učinkovita, da bo delovala hitro tudi za zelo dolge nize in velike n. Predpostavi, da sta `s` in `t` dolga po vsaj n znakov, tako da rešitev gotovo obstaja.

5. Zlaganje loncev

V kuhinjsko omaro zlagamo lonce. Lonci so v obliki odprtih valjev različnih premerov. Zaradi prihranka prostora lahko natanko en manjši lonec položimo v večjega, kadar ima manjši premer osnovne ploskve kot večji lonec. V ta manjši lonec pa lahko kasneje položimo še en manjši lonec in tako naprej, da dobimo nekakšen sklad loncev. Ne želimo pa v en lonec neposredno postaviti dveh ali več manjših (npr. da bi v lonec premera 20 cm postavili neposredno lonca premerov 5 cm in 3 cm; v tem primeru bi v lonec premera 20 cm postavili lonec premera 5 cm, v slednjega pa potem lonec premera 3 cm). Želimo preveriti, ali je naša kuhinjska omara dovolj prostorna, da lahko v njo na ta način postavimo vse svoje lonce.

Opiši postopek (ali napiši program ali podprogram oz. funkcijo, če ti je lažje), ki kot vhodni podatek dobi seznam premerov vseh loncev na kuhinjski mizi ter izračuna najmanjše število skladov, ki jih lahko sestavimo iz teh loncev. Izračuna pa naj tudi najnižjo možno vsoto, ki jo lahko dobimo, če vzamemo premer najbolj spodnjega lonca v vsakem skladu in te premere seštejemo po vseh skladih. Dobro tudi utemelji pravilnost svojega postopka.

Primer: če imamo lonce s premeri

$$28, 17, 14, 29, 12, 22, 28, 28, 13, 20, 30, 18, 4, 18, 4,$$

potrebujemo najmanj tri sklade, najmanjša možna vsota premerov pa je 86. (Eden od možnih načinov, kako lahko zložimo lonce na optimalen način, so takšni trije skladi: $[4, 14, 28, 29, 30]$, $[13, 17, 18, 28]$ in $[4, 12, 18, 20, 22, 28]$; vsota premerov najbolj spodnjih loncev je takrat $30 + 28 + 28 = 86$.)

16. tekmovanje ACM v znanju računalništva za srednješolce

27. marca 2021

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 tekmovalca diskvalificira, če bi se kasneje izkazalo, da nalog ni reševal sam. 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“. Ker je vgrajeni urejevalnik dokaj preprost in ne omogoča označevanja kode z barvami, predlagamo, da rešitev pripraviš v urejevalniku na svojem računalniku in jo nato prekopiraš v okno spletnega urejevalnika. Naj te ne moti, da se bodo barvne oznake kode pri kopiranju izgubile.

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

Med reševanjem lahko vprašanja za tekmovalno komisijo postavljaš prek zasebnih sporočil na tekmovalnem strežniku (ikona oblačka zgoraj desno), izjemoma pa tudi po elektronski pošti na `rtk-info@ijs.si`. Prek zasebnih sporočil bomo pošiljali tudi morebitna pojasnila in popravke, če bi se izkazalo, da so v besedilu nalog kakšne nejasnosti ali napake. Zato med reševanjem redno preverjaj, če so se pojavila kakšna nova zasebna sporočila.

Če imaš 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. Sredinec

V šoli je pri športni vzgoji navada, da se pred začetkom šolske ure učenci postavijo v vrsto od najmanjšega do največjega. Prav tako je navada, da učenci zamujajo. Vsak učenec, ki vstopi v telovadnico, se vrine na svoje mesto v vrsti glede na velikost. Učitelj športne vzgoje se med tem zamudnim procesom zabava z opazovanjem, kdo se po vsakem novem prihodu nahaja na sredini vrste. Učenci vstopajo posamično, učitelja pa zanima, kako visok je tisti izmed n prisotnih učencev, ki se trenutno nahaja na $\lceil n/2 \rceil$-tem mestu v vrsti od najmanjšega do največjega. (Zapis $\lceil n/2 \rceil$ pomeni, da rezultat po deljenju n z 2 zaokrožimo navzgor. Na primer: pri $n = 5$ in $n = 6$ ga zanima tretji po vrsti, pri $n = 7$ in $n = 8$ četrti po vrsti in podobno.)

Opiši postopek, ki to nalogo reši čim bolj učinkovito (recimo, da učencev ni le nekaj deset, ampak na milijone): prebira naj višine učencev v takem vrstnem redu, kakor vstopajo v telovadnico, in po vsakem prebranem učencu sproti izpiše višino tistega, ki je zdaj srednji po višini. Oceni tudi časovno zahtevnost svoje rešitve, torej kako se povečuje čas izvajanja v odvisnosti od števila učencev. Višine učencev so podane v obliki seznama po vrsti, tako kot vstopajo v telovadnico. Višine niso večje od dveh metrov in so podane s celimi števili, ki predstavljajo višino v centimetrih.

2. Svetilka

Žepna baterijska svetilka je lahko ugasnjena ali pa sveti v dveh možnih načinih: sveti stalno ali pa utripa tako, da vsako sekundo posveti za eno desetinko sekunde (in je potem devet desetink sekunde ugasnjena). Za preklop med temi tremi stanji služi tipka.

Takoj ko pritisnemo tipko (t.j. ob začetku pritisnjenosti tipke) naj se svetilka vklopi: če je bila prej ugasnjena, naj se vklopi v stalni način, če je bila v stalnem načinu, naj se preklopi v utripanje, in če je utripala, naj se preklopi v stalni način. Če je tipka pritisnjena tri sekunde ali več, naj se po teh treh sekundah svetilka izklopi.

Podana je funkcija Luc(vklop), s katero lahko program upravlja svetilo: vrednost **true** vklopi svetilo, **false** ga izklopi.

Napiši naslednji dve **funkciji**, ki ju bo operacijski sistem malega računalnika v svetilki avtomatsko klical takole:

- Tiktak() — ta funkcija bo poklicana vsako desetinko sekunde;
- Tipka(pritisnjena) — ta funkcija bo poklicana vsakokrat, ko se bo stanje pritisnjenosti tipke spremenilo; vrednost argumenta bo **true**, če je bila tipka pravkar pritisnjena (t.j. začetek pritiska), in **false**, če je bila tipka pravkar spuščena.

Za ohranitev stanja programa lahko uporabiš poljubne globalne spremenljivke in jih tudi po svoje inicializiraš. Na začetku delovanja programa je luč ugasnjena, tipka pa spuščena.

Glede na to, da nimamo možnosti merjenja časa z večjo ločljivostjo od desetinke sekunde, ne bo nič narobe, če ob preklopu na utripanje prvi blisk ne traja točno eno desetinko sekunde, prav tako lahko trosekundni interval (za ugašanje svetilke) odstopa za malenkost.

Če si želiš poenostaviti nalogo, lahko opustiš stanje utripanja in poskrbiš le za vklop in izklop svetilke — pri tem boš dobil največ polovico točk naloge.

3. Pletenje puloverja

Neža se je med karanteno lotila novega konjička. Naučila se je plesti. Nekaj preglavic pa ji povzročajo sheme za pletenje vzorcev. V knjigah so pogosto narisane velike sheme, na primer:

```
---0-0---0-0---0-0---0-0---0-0
---000---000---000---000---000
---0-0---0-0---0-0---0-0---0-0
---000---000---000---000---000
---0-0---0-0---0-0---0-0---0-0
---000---000---000---000---000
---0-0---0-0---0-0---0-0---0-0
---000---000---000---000---000
```

Zgornja shema predstavlja osnovni vzorec

```
---0-0
---000
```

Neža je hitro ugotovila, da si mora pri pletenju izdelka zapomniti oziroma zapisati le osnovni vzorec in ne celotne velike sheme iz knjige. Prosi te, da **napišeš program** ali podprogram (funkcijo), ki v poljubni dani shemi poišče osnovni vzorec. Osnovni vzorec je najmanjši (po površini) tak vzorec, iz katerega lahko s ponavljanjem sestavimo celotno shemo (pri čemer se mora vzorec lepo zaključiti na vseh robovih sheme). Program naj izpiše širino in višino osnovnega vzorca. Za primer zgoraj je rešitev `6 2`. Če je možnih več enako dobrih rešitev, je vseeno, katero od njih izpišeš. Shemo lahko tvoj program prebere iz datoteke ali s standardnega vhoda ali pa predpostavi, da je že podana v neki tabeli ali seznamu nizov (ali dvodimenzionalni tabeli znakov). Shemo sestavljajo le znaki „`-`“ in „`0`“.

4. Pangramski podniz

Pangram je niz, ki vsebuje vsako črko abecede vsaj enkrat; pri tej nalogi pa nas bodo zanimali malo bolj posebni pangrami — taki, ki vsebujejo vsako črko abecede vsaj k-krat. **Napiši podprogram** oz. funkcijo, ki za dani niz s in naravno število k vrne dolžino najkrajšega takega strnjenega podniza niza s, v katerem se vsaka črka abecede pojavi vsaj k-krat. Če takega podniza sploh ni, naj funkcija vrne -1. Niz s je sestavljen le iz malih črk angleške abecede, lahko pa je zelo dolg, zato naj bo tvoja rešitev čim bolj učinkovita.

Primer: če bi namesto cele abecede gledali le črke $\{\texttt{a}, \texttt{b}, \texttt{c}\}$ in če bi imeli $k = 2$, bi bil najkrajši primerni podniz v nizu $s = \texttt{aabaaccabaacccbabb}$ dolg 7 znakov. Taki podnizi so celo trije: `baaccab`, `baacccb`, `acccbab`. Poudarimo pa, da je to samo primer in da mora tvoja rešitev delovati za celotno abecedo in za poljuben k in poljubno dolg niz s.

5. Tetris

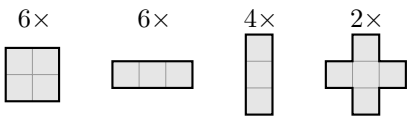
Imamo ploščo, sestavljeno iz 8×8 kvadratnih polj, ki bi jo radi pokrili s ploščki v obliki raznih likov, podobnih tistim iz igre Tetris. **Napiši program** ali podprogram (funkcijo), ki bo ploščo v celoti pokril s ploščki, pri čemer se le-ti med seboj ne smejo prekrivati ali štrleti čez rob plošče. Na voljo so ploščki n različnih oblik, ki so oštevilčene od 1 do n; v vsaki obliki pa je na voljo le omejeno število ploščkov. Za delo s ploščki naj tvoj program uporablja naslednje funkcije (zanje torej predpostavi, da že obstajajo in ni mišljeno, da jih ti implementiraš sam):

- **int** StOblik() — vrne n, torej število, ki pove, koliko različnih oblik ploščkov je na voljo.
- **int** StPlosckov(**int** oblika) — vrne število razpoložljivih ploščkov oblike oblika. To je celo število, večje od 0, vanj pa so všteti tudi tisti ploščki, ki jih je tvoj program mogoče že postavil na ploščo.
- **bool** JePokrito(**int** x, **int** y) — vrne logično vrednost, ki pove, ali je polje (x, y) na plošči trenutno pokrito (torej ali ga pokriva kakšen od že doslej postavljenih ploščkov). Na začetku izvajanja tvojega programa je plošča prazna (torej ni na njej še nobenega ploščka). Koordinate polj na plošči gredo od $x = 0$ (levo) do $x = 7$ (desno) in od $y = 0$ (zgoraj) do $y = 7$ (spodaj).
- **bool** PreveriPloscek(**int** oblika, **int** x, **int** y) — vrne logično vrednost, ki pove, ali je mogoče na ploščo dodati plošček oblike oblika tako, da najbolj levo polje v najbolj zgornji vrstici tega ploščka pokrije polje (x, y) na plošči. (Funkcija preverja le obliko, ne pa tudi tega, ali imaš še na voljo kaj ploščkov te oblike ali pa si jih morda že vse postavil na ploščo.)
- **void** PostaviPloscek(**int** oblika, **int** x, **int** y, **bool** b) — če je b == **true**, ta funkcija položi plošček oblike oblika na ploščo tako, da najbolj levo polje v najbolj zgornji vrstici tega ploščka pokrije polje (x, y) na plošči. Če je b == **false**, pa funkcija ta plošček s tega položaja odstrani.

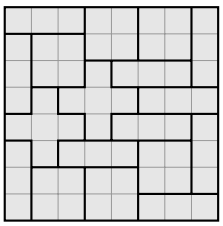
Funkcija PostaviPloscek prekine izvajanje tvojega programa, če zahtevaš od nje operacijo, ki je ni mogoče izvesti (npr. dodajanje ploščka neke oblike, če si vse razpoložljive ploščke te oblike že položil na mrežo; ali dodajanje ploščka tako, da bi se prekrival z že obstoječimi ali štrlel čez rob mreže; ali brisanje ploščka, ki ga v resnici ni tam).

Ploščkov se pri tej nalogi ne dá obračati ali vrteti in funkciji PreveriPloscek in PostaviPloscek tega tudi ne poskušata početi. To pomeni, da štejeta na primer  in za dve različni obliki in ploščkov ene oblike ne moremo zasukati in uporabiti kot ploščke druge oblike.

Primer. Recimo, da imamo ploščke naslednjih štirih oblik v naslednjih količinah:



Potem lahko ploščo pokrijemo takole:



Še deklaracije gornjih funkcij v drugih jezikih:

{ *V pascalu:* }
function StOblik: integer;
function StPlosckov(oblika: integer): integer;
function JePokrito(x, y: integer): boolean;
function PreveriPloscek(oblika, x, y: integer): boolean;
procedure PostaviPloscek(oblika, x, y: integer; b: boolean);

// *V javi: deklaracije so kot v besedilu naloge, le z* **boolean** *namesto* **bool**.

V pythonu:
def StOblik() –> int: ...
def StPlosckov(oblika: int) –> int: ...
def JePokrito(x: int, y: int) –> bool: ...
def PreveriPloscek(oblika: int, x: int, y: int) –> bool: ...
def PostaviPloscek(oblika: int, x: int, y: int, b: bool) –> None: ...

16. tekmovanje ACM v znanju računalništva za srednješolce

27. marca 2021

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-2021-3/` najdeš opise nalog v elektronski obliki. Prek iste strani lahko oddaš tudi rešitve svojih nalog. Pred začetkom tekmovanja lahko poskusiš oddati katero od nalog iz arhiva `https://putka-rtk.acm.si/tasks/s/test-sistema/list/`. Uporabniško ime in geslo za Putko 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. Pri tretji in četrti nalogi je testnih primerov po 20 in vsak je vreden po 5 točk, pri ostalih pa je testnih primerov po 10 in vsak je vreden po 10 točk.

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));
  }
}
```

16. tekmovanje ACM v znanju računalništva za srednješolce

27. marca 2021

NALOGE ZA TRETJO SKUPINO

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

1. Kapniki

Jamarji so odkrili dolgo nizko jamo s številnimi kapniki. Tla in strop jame sta vzporedna. Jama je višine v in dolžine n, na vsakem metru jame pa raste s tal stalagmit ali s stropa stalaktit. Za vsak kapnik poznamo njegov tip (stalagmit ali stalaktit) in velikost kapnika k_i ($1 \leq k_i < v$). V jami bi radi postavili turistično železnico, ki bo potekala vzporedno s tlemi in stropom na neki celoštevilski višini y ($1 \leq y \leq v$). Ker so stalagmiti velikosti $k_i \geq y$ in stalaktiti velikosti $k_i > v - y$ taki železnici v napoto, jih bo treba podreti. **Napiši program**, ki bo poiskal višine železnice y, pri katerih bi bilo treba podreti čim manj kapnikov.

Vhodni podatki: v prvi vrsti sta celi števili v in n, ločeni s presledkom. V drugi vrstici je podan niz n znakov `'M'` ali `'T'`, kjer i-ti znak v nizu predstavlja tip kapnika, ki raste na i-tem metru jame. Če je enak `'M'`, gre za stalagmit, ki raste s tal, če je enak `'T'`, pa za stalaktit, ki raste s stropa. V tretji vrstici je podan s presledki ločen seznam n celih števil k_i, kjer i-to število predstavlja velikost i-tega kapnika.

Omejitve: veljalo bo $1 \leq v \leq 10^{18}$ in $1 \leq n \leq 10^5$.

- V prvih 20 % testnih primerov bo $n \leq 1000$ in $v \leq 1000$.
- V naslednjih 40 % testnih primerov bo $v \leq 10^6$.

Izhodni podatki: izpiši najmanjše število kapnikov, ki jih bo treba podreti, in število višin železnice, pri katerih lahko dosežemo to število podrtih kapnikov. Števili izpiši v isti vrstici, ločeni pa naj bosta z enim presledkom.

Primer vhoda:

```
8 9
TTMTMMTTM
2 1 6 5 2 5 3 7 2
```

Pripadajoči izhod:

```
3 1
```

2. Socialno omrežje

V neki demokratični deželi daleč daleč stran se je blaženi vodja odločil prepovedati socialna omrežja tehnoloških gigantov, kot sta npr. Twitter in Facebook, zaradi morebitnega širjenja neprimernih vsebin. Namesto tega pa so zgradili svoje lastno socialno omrežje, kjer lahko uporabniki med seboj sklepajo prijateljstva in sovraštva. Vemo, da uporabniki omenjenega omrežja prijateljstva in sovraštva sklepajo po naslednjih pravilih: „prijatelj mojega prijatelja je moj prijatelj“, „sovražnik mojega prijatelja je moj sovražnik“ in „sovražnik mojega sovražnika je moj prijatelj“. Omenjena pravila so bolj formalno definirana kasneje. Dobili smo dostop do podatkov o prijateljih in sovražnikih na tem omrežju, zanima pa nas, ali so omenjena pravila dosledno spoštovana, torej ali niti nikoli niso kršena niti z njihovim upoštevanjem ne moremo skleniti novih prijateljstev ali sovraštev.

Vhodni podatki: v prvi vrstici je podano celo število t, to je število omrežij v tem testnem primeru. Sledijo opisi vseh t omrežij. Vsako omrežje se začne z vrstico, ki vsebuje celi števili n (število ljudi v omrežju) in m (število prijateljstev ali sovraštev med njimi). Uporabniki so označeni s števili od 1 do n. Sledi m vrstic; vsaka izmed njih vsebuje tri števila a_i, b_i in p_i. To nam pove, da sta osebi a_i in b_i povezani med seboj. Če je p_i enak 0, sta a_i in b_i sovražnika, če je p_i enak 1, pa prijatelja. Čustva so obojestranska in vsako je navedeno samo enkrat, prav tako ni mogoče, da bi bili dve osebi hkrati prijatelja in sovražnika.

Za vsako podano omrežje želimo preveriti, ali dosledno spoštuje naslednja tri pravila:

1. Prijatelj mojega prijatelja je moj prijatelj: če sta osebi A in B prijatelja in osebi B in C prijatelja, potem morata biti tudi osebi A in C prijatelja.
2. Sovražnik mojega prijatelja je moj sovražnik: če sta osebi A in B prijatelja in osebi B in C sovražnika, potem morata biti osebi A in C sovražnika.
3. Sovražnik mojega sovražnika je moj prijatelj: če sta osebi A in B sovražnika in osebi B in C sovražnika, potem morata biti osebi A in C prijatelja.

Izhodni podatki: za vsako izmed t omrežij izpiši „`DA`“, če omenjena pravila dosledno veljajo, sicer pa „`NE`“. Vsak odgovor naj bo podan v svoji vrstici.

Omejitve vhodnih podatkov: vedno bo veljalo $1 \leq n \leq 10^5$, $1 \leq m \leq 10^5$ in $t \leq 10$.

Podnaloge:

- V prvih 20 % testnih primerov velja $n \leq 100$ in $m \leq 100$.
- V naslednjih 40 % testnih primerov velja $n \leq 1000$ in $m \leq 1000$.
- Pri preostalih 40 % testnih primerov ni dodatnih omejitev.

Primer vhoda:

```
2
5 6
1 2 1
1 3 1
2 3 1
1 4 0
3 4 0
2 4 0
3 3
1 2 0
2 3 0
1 3 0
```

Pripadajoči izhod:

```
DA
NE
```

3. Proizvodnja cepiva

Za farmacevtsko podjetje, ki proizvaja cepiva proti COVID-19, moramo sestaviti načrt proizvodnje cepiva. Trenutno proizvajamo 0 odmerkov cepiva na dan. Vsak dan znova se odločimo, ali bomo proizvajali cepivo ali pa nadgrajevali proizvodnjo.

Če se odločimo za nadgradnjo proizvodnje, ta dan ne proizvedemo nič cepiva, dnevno proizvodnjo odmerkov cepiva pa povečamo za 1. Če se odločimo za proizvodnjo, pa ta dan proizvedemo toliko odmerkov, kolikor je naša trenutna dnevna proizvodnja cepiva.

Naš cilj je v čim krajšem času ustvariti k odmerkov cepiva. A to še ni vse! Zaradi cepljenja najbolj ranljivih skupin moramo nekaj cepiva dostaviti že vnaprej. Natančneje rečeno, podanih imamo d omejitev, vsaka od njih pa pravi, da moramo v roku x_i dni skupno ustvariti vsaj y_i odmerkov cepiva, kjer so dnevi oštevilčeni začenši z 1.

Napiši program, ki izračuna, koliko najmanj dni potrebujemo, da ustvarimo k odmerkov cepiva, če upoštevamo vse omejitve in optimalno izbiramo strategijo za nadgradnjo in proizvodnjo cepiva.

Vhodni podatki: v prvi vrstici bosta podani dve števili, k (število odmerkov cepiva, ki jih moramo proizvesti) in d (število omejitev, ki jih moramo pri tem upoštevati). Nato sledi d vrstic, ki opisujejo omejitve, v vsaki izmed njih pa sta dani števili x_i in y_i. V prvih x_i dneh je treba skupno proizvesti vsaj y_i odmerkov cepiva.

Izhodni podatki: tvoj program naj izpiše eno število, namreč minimalno število dni, ki jih potrebujemo, da ustvarimo dovolj odmerkov cepiva. Zagotovljeno je, da bo rešitev vedno obstajala.

Omejitve podatkov: $0 \leq d \leq 100$, $1 \leq k \leq 10^6$, $1 \leq y_i \leq k$, $1 \leq x_i le 10^6$. Pri prvih 20 % testnih primerov bo $d = 0$; pri naslednjih 30 % testnih primerov bo $k \leq 10^4$.

Primer vhoda:	Pripadajoči izhod:
`100 1` `4 4`	`22`

4. Virus v Timaniji

V deželi Timaniji so slišali, da po svetu razsaja nov smrtonosni virus. Oseba, ki je okužena, postane tudi sama kužna po natanko k dneh, v natanko ℓ dneh po nastopu kužnosti pa oseba v strašnih krčih in mukah umre. (Primer: če je $k = 2$ in $\ell = 3$ in se je nekdo okužil v ponedeljek, bo sam okuževal druge v sredo, četrtek in petek, umrl pa bo v soboto in tisti dan ne bo okužil nikogar.) Drugih simptomov pred smrtjo ni, tako da živih okuženih državljanov ni mogoče poslati v osamo.

V strahu pred izbruhom epidemije je vlada Timanije sprejela ukrepe, kjer je omejila srečanja med državljani, tako da se vsak državljan lahko na vsak dan v tednu sreča le z enim preostalim državljanom, skupno z največ 7 državljani v tednu. Vsak državljan je moral na seznam napisati, koga bo srečal kateri dan v tednu, seznama pa kasneje ne smejo spreminjati in se ga morajo vsi državljani strogo držati (seznam je torej vsak teden enak).

Napiši program, ki bo preveril, ali bo po vnosu virusa celotno prebivalstvo izumrlo ali ne. Vlado zanima, ali za dan urnik srečanj za celotno prebivalstvo in določena k in ℓ obstaja scenarij, kjer se bo na točno določen dan d okužil (od zunaj) točno določen državljan in se bo tako sčasoma okužilo (in pomrlo) celotno prebivalstvo (število prebivalcev je n). Če tak scenarij ne obstaja, pa poišči scenarij z najmanjšim številom preživelih državljanov po epidemiji. Dni v tednu je 7, kjer nedeljo predstavlja število 0, ponedeljek število 1, soboto pa število 6. Predpostaviš lahko, da se vsa srečanja zgodijo ob istem času zjutraj in da na dan smrti okuženi ne okuži nikogar več.

Vhodni podatki: v prvi vrstici so števila n, k in ℓ, ločena s po enim presledkom. Sledi n vrstic s po sedmimi številkami, ki predstavlja urnik srečanj državljana: j-ta številka v i-ti vrstici predstavlja številko državljana, ki ga bo i-ti državljan srečal v j-tem dnevu vsakega tedna. (Državljani so oštevilčeni s številkami od 0 do $n - 1$.)

Veljalo bo $1 \leq k \leq 15$, $1 \leq \ell \leq 15$. Veljalo bo še:

- v prvih 20 % primerov: $\ell = 1$ in $n \leq 1000$;
- v naslednjih 40 % primerov: $\ell = 1$, $n \leq 10^5$;
- v preostalih 40 % primerov: $\ell \leq 15$, $n \leq 1000$.

Izhodni podatki: izpiši štiri cela števila i, d, p in r, ločena s po enim presledkom. Pri tem naj bo i številka državljana in d številka dneva v tednu za tisti scenarij, pri katerem umre največ ljudi, če se okužba začne s tem, da se državljan i okuži na dan d; število p naj bo za ta scenarij število preživelih po epidemiji; število r pa naj pove, koliko scenarijev s tem številom preživelih obstaja, torej za koliko parov (i', d') velja, da na koncu ostane p preživelih, če se epidemija začne s tem, da se človek i' okuži na dan d'. Če obstaja več enako dobrih rešitev, je vseeno, katero od njih izpišeš.

Primer vhoda:

```
8 3 2
3 6 2 1 3 5 2
7 3 3 0 5 6 5
4 4 0 4 4 3 0
0 1 1 6 0 2 7
2 2 6 2 2 7 6
6 7 7 7 1 0 1
5 0 4 3 7 1 4
1 5 5 5 6 4 3
```

Pripadajoči izhod:

```
1 5 0 24
```

Komentar. Scenarij, na katerega se sklicuje izhod v gornjem primeru, je naslednji:

- V petek (dan 5) prvega tedna se okuži državljan 1.
- Drugi teden, dan 1: državljan 1 okuži državljana 3. (Naslednji dan se spet srečata, ampak je 3 že okužen.)
- Drugi teden, dan 4: državljan 3 okuži državljana 0.
- Drugi teden, dan 5: državljan 3 okuži državljana 2.
- Tretji teden, dan 1: državljan 0 okuži državljana 6, državljan 2 pa okuži državljana 4.

- Tretji teden, dan 4: državljan 6 okuži državljana 7.
- Četrti teden, dan 1: državljan 7 okuži državljana 5.

Tako so se okužili vsi državljani in preživelih ni. To je eden od 24 možnih scenarijev, pri katerih za te vhodne podatke umrejo vsi ljudje.

Še en primer vhoda:

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

Pripadajoči izhod:

```
3 4 0 16
```

5. Tja in spet nazaj

Hobit se odpravlja na dogodivščino. V roki ima zemljevid, na katerem je označil n točk, ki jih želi obiskati vsaj enkrat. Trenutno se nahaja doma na najbolj zahodni točki (tisti z najmanjšo x-koordinato), kamor se želi na koncu tudi vrniti. Odločil se je, da ga bo njegova pot najprej vodila ves čas proti vzhodu v smeri naraščajočih x-koordinat točk, nato pa se bo obrnil in se ves čas premikal nazaj proti zahodu v smeri padajočih x-koordinat točk. **Napiši program**, ki bo izračunal dolžino najkrajše hobitove poti, na kateri obišče vse točke vsaj enkrat in se vrne domov.

Vhodni podatki: v prvi vrstici je število točk n, ki so podane v sledečih n vrsticah. V vsaki vrstici sta podani s presledkom ločeni koordinati x_i in y_i neke točke na zemljevidu. Vse koordinate x_i bodo med seboj različne.

Omejitve: veljalo bo $2 \leq n \leq 5000$. Koordinate točk x_i in y_i bodo celoštevilske z intervala $[0, 100\,000]$.

- V prvih 30 % testnih primerov bo $n \leq 20$.
- V naslednjih 40 % testnih primerov bo $n \leq 500$.

Izhodni podatki: izpiši dolžino najkrajše hobitove poti. Rešitev bo sprejeta, če se bo od uradne razlikovala za največ 10^{-4}.

Primer vhoda:

```
8
7 5
3 4
4 0
2 4
1 2
5 1
6 3
9 4
```

Eden od možnih pripadajočih izhodov:

```
20.013352
```

16. tekmovanje ACM v znanju računalništva za srednješolce

27. marca 2021

REŠITVE NALOG ZA PRVO SKUPINO

1. Gesla

Nalogo lahko rešimo z dvema gnezdenima zankama. Zunanja zanka bo pregledala vse možne položaje pike; na začetku dodamo piko recimo na konec niza, nato pa po vsaki iteraciji te zanke premaknemo piko za en znak nazaj (tisti znak pa, ki je bil prej tik pred piko, se pri tem premakne tik za piko). Pazimo le na to, da po zadnji iteraciji, ko je pika že na začetku niza, ne poskušamo premakniti pike še bolj nazaj.

V notranji zanki pa bomo (pri vsakem položaju pike) poskušali na vse možne načine spremeniti po eno malo črko v veliko. Ker niz ni pretirano dolg in je verjetno večina znakov v njem črk, gremo lahko v tej drugi zanki kar po vseh znakih niza in pri vsakem najprej preverimo, ali je črka; če ni, gremo takoj na naslednji znak. Če pa je trenutni znak (mala) črka, jo spremenimo v veliko, niz izpišemo in spremenimo črko nazaj v malo, da povrnemo niz v prejšnje stanje.

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

void MoznaGesla(string geslo)
{
  geslo.push_back('.'); // Dodajmo piko na konec niza.
  // Z zanko preizkusimo vse možne položaje pike.
  for (int pika = geslo.length() - 1; pika >= 0; --pika)
  {
    // Na vse možne načine spremenimo eno malo črko v veliko.
    for (char &c : geslo) if (isalpha(c))
    {
      c = toupper(c); // Spremenimo to črko v veliko.
      cout << geslo << endl;
      c = tolower(c); // Spremenimo črko nazaj v malo.
    }
    // Premaknimo piko eno mesto nazaj.
    if (pika > 0) swap(geslo[pika], geslo[pika - 1]);
  }
}
```

Oglejmo si še primer rešitve v pythonu. Tu niza ne moremo spreminjati, zato bomo morali delati kopije niza. V zunanji zanki bomo šli po vseh znakih niza; ne-črke preskočimo, če pa je trenutni znak črka, pripravimo kopijo niza, pri kateri to (malo) črko zamenjamo z ustrezno veliko črko. Nato izvedemo še notranjo zanko, ki gre po vseh možnih položajih pike in izpiše različico niza, v kateri je pika vrinjena na ta položaj.

```
def MoznaGesla(geslo):
  n = len(geslo)
  # Na vse možne načine spremenimo po eno črko v veliko.
  for velika in range(n):
    # Ne-črkovne znake preskočimo.
    if not geslo[velika].isalpha(): continue
    # Pripravimo kopijo gesla, v kateri je trenutna črka velika.
    geslo2 = geslo[:velika] + geslo[velika].upper() + geslo[velika + 1:]
```

```
    # Na vse možne načine dodajmo piko.
    for pika in range(n + 1):
        # Izpišimo različico gesla, v kateri je pika na indeksu „pika“.
        print(geslo2[:pika] + "." + geslo2[pika:])
```

2. Marsovci

Ker so opravila oštevilčena od 1 do 100, lahko uporabimo tabelo 100 elementov (ali 101, ker gredo indeksi od 0, nam pa bo lažje uporabljati indekse do 100), v kateri bomo šteli, koliko marsovcev se specializira za posamezno opravilo. Na začetku vse elemente te tabele inicializiramo na 0, nato pa v zanki beremo podatke o marsovcih in ustrezno povečujemo števce v tabeli. Na koncu se sprehodimo po celotni tabeli in poiščemo najmanjši in največji element, pri tem pa pazimo, da tiste z vrednostjo 0 preskočimo, saj ni nujno, da se vsa števila od 1 do 100 res pojavljajo v naših vhodnih podatkih. Če je razlika med največjim in najmanjšim elementom največ 1, so opravila približno enako zastopana, sicer pa ne.

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

int main()
{
  int zastopanost[101] = {};
  int m; cin >> m; // Preberimo število marsovcev.
  while (m-- > 0)
    // Preberimo opravila naslednjega marsovca.
    for (int i = 0; i < 5; i++)
    {
      int opravilo; cin >> opravilo; // Preberimo naslednje opravilo.
      ++zastopanost[opravilo];     // Povečajmo števec zastopanosti tega opravila.
    }
  // Poiščimo najmanjšo in največjo zastopanost.
  int min = -1, max = -1;
  for (int z : zastopanost)
  {
    if (z == 0) continue; // Ta številka opravila sploh ni v rabi.
    if (min < 0 || z < min) min = z;
    if (max < 0 || z > max) max = z;
  }
  // Izpišimo rezultat.
  cout << (max - min <= 1 ? "da" : "ne") << endl; return 0;
}
```

Zapišimo podobno rešitev še v pythonu. Za iskanje največje in najmanjše zastopanosti po vseh opravilih lahko uporabimo pythonovi funkciji min in max, če iz tabele prej pobrišemo ničle (ki predstavljajo neuporabljene številke opravil).

```
import sys

zastopanost = [0] * 101
m = int(sys.stdin.readline()) # Preberimo število marsovcev.
for i in range(m):
  # Preberimo opravila naslednjega marsovca.
  for opravilo in sys.stdin.readline().split():
    # Povečajmo števec zastopanosti tega opravila.
    zastopanost[int(opravilo)] += 1

# Pobrišimo ničle iz tabele, ker predstavljajo številke
# opravil, ki se v vhodnih podatkih sploh ne pojavljajo.
zastopanost = [z for z in zastopanost if z > 0]

# Izpišimo rezultat.
print("da" if max(zastopanost) - min(zastopanost) <= 1 else "ne")
```

3. Rekonstrukcija poti

Recimo, da pri branju vhoda preberemo podatek, da imamo direktorij s na globini g. Da dobimo polno pot do njega, moramo vzeti polno pot do njegovega starša (naddirektorija) in ji pritakniti poševnico / ter niz s. Koristno je torej, če imamo takrat to pot do starša že nekje pri roki. Ker pa ne moremo vnaprej vedeti, kakšen g bomo v naslednji vrstici dobili, moramo pravzaprav imeti pri roki poti do trenutnega direktorija in vseh njegovih prednikov. Hranili jih bomo v nekakšnem seznamu, ki ga uporabljamo bolj ali manj kot sklad, torej elemente dodajamo in brišemo le na koncu.

Ko potem preberemo ime direktorija s na globini g, moramo z vrha sklada pobrisati toliko elementov, da jih ostane le $g - 1$; zadnji med temi je potem neposredni naddirektorij našega pravkar prebranega direktorija in iz polne poti do tega naddirektorija lahko izračunamo polno pot do našega pravkar prebranega ter jo dodamo na vrh sklada. (V praksi ni treba brisati toliko elementov, da jih ostane $g - 1$, ampak jih lahko pustimo g in potem g-tega povozimo z novo potjo do pravkar prebranega direktorija).

Poseben primer nastopi, če je na skladu že zdaj manj kot $g - 1$ elementov; takrat poti do pravkar prebranega direktorija ni mogoče določiti (kot npr. pri drugem primeru v besedilu naloge), zato lahko le še javimo napako in končamo z izvajanjem programa.

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

int main()
{
  // Sprva bo na skladu le prazen niz, ki predstavlja koren drevesa (na globini 0).
  vector<string> sklad = { "" };
  while (true)
  {
    // Preberimo naslednji direktorij.
    string s; int globina;
    cin >> s >> globina; if (! cin.good()) break;

    // Če je globina prevelika, sporočimo napako.
    if (globina > sklad.size()) { cout << "Napaka!"; break; }

    // Če je globina za 1 večja od dosedanje, dodajmo na sklad nov element
    // s polno potjo do pravkar prebranega direktorija.
    else if (globina == sklad.size())
      sklad.push_back(sklad.back() + "/" + s);
    else {
      // Sicer pobrišimo toliko elementov, da bo zadnji tisti na indeksu „globina“.
      while (sklad.size() > globina + 1) sklad.pop_back();

      // Vanj vpišimo polno pot do pravkar prebranega direktorija.
      sklad[globina] = sklad[globina - 1] + "/" + s; }

    // Izpišimo polno pot do trenutnega direktorija.
    cout << sklad.back() << endl;
  }
  return 0;
}
```

Zapišimo to rešitev še v pythonu:

```
import sys
# Sprva bo na skladu le prazen niz, ki predstavlja koren drevesa (na globini 0).
sklad = [""]
for vrstica in sys.stdin:

  # Preberimo naslednji direktorij.
  s, globina = vrstica.split(); globina = int(globina)

  # Če je globina prevelika, sporočimo napako.
  if globina > len(sklad): print("Napaka!"); break
```

```
    # Če je globina za 1 večja od dosedanje, dodajmo na sklad nov element
    # s polno potjo do pravkar prebranega direktorija.
    elif globina == len(sklad):
        sklad.append(sklad[-1] + "/" + s)

    else:
        # Sicer pobrišimo toliko elementov, da bo zadnji tisti na indeksu „globina“.
        del sklad[globina + 1:]

        # Vanj vpišimo polno pot do pravkar prebranega direktorija.
        sklad[-1] = sklad[-2] + "/" + s

    # Izpišimo polno pot do trenutnega direktorija.
    print(sklad[-1])
```

4. Kako dobri so virusni testi?

V mislih lahko po obeh nizih hkrati pomikamo „okno“ širine n znakov. Pri tem bomo v neki spremenljivki (v spodnji rešitvi je to razlik) vzdrževali število mest znotraj okna, kjer se istoležna znaka nizov s in t razlikujeta. Ko se okno premakne za en znak naprej (v desno), tega števila ni težko popraviti: če je zadnji indeks v oknu zdaj recimo i, to pomeni, da je ta indeks zdaj na novo prišel v okno in moramo števec razlik povečati za 1, če se niza na tem indeksu razlikujeta (torej če sta $s[i]$ in $t[i]$ različna). In če je zadnji indeks v oknu i, okno pa je dolgo n znakov, to pomeni, da je prvi indeks v oknu $i - n + 1$; indeks $i - n$ pa, ki je bil malo prej še v oknu, je zdaj na levi izpadel iz okna, tako da moramo števec razlik zmanjšati za 1, če je bila na tistem mestu med nizoma razlika (torej če sta bila $s[i - n]$ in $t[i - n]$ različna). Tako lahko po vsakem premiku okna izračunamo novo število razlik s samo konstantno mnogo operacijami, torej v $O(1)$ časa, neodvisno od širine okna n.

Po vsakem premiku okna moramo novo število razlik primerjati z največjim doslej in če je novo večje, si ga zapomnimo, skupaj z njim pa tudi i, pri katerem smo ga dobili (v spodnji rešitvi je to spremenljivka najKje). Pomembno je, da najKje popravimo le, če je novo število razlik strogo večje od največjega doslej, ne pa, če je enako; s tem bomo zagotovili, da bomo med več enako dobrimi položaji okna vrnili najbolj levega, tako kot zahteva naloga. Paziti moramo še na to, da naloga zahteva indeks najbolj levega znaka v oknu, naša spremenljivka i pa pove indeks najbolj desnega, tako da moramo na koncu še odšteti $n - 1$.

Ker smo imeli pri vsakem možnem položaju okna le $O(1)$ dela, je časovna zahtevnost tega postopka $O(d)$, če je d dolžina nizov s in t.

```
int Primerjava(const char *s, const char *t, int n)
{
    int najRazlik = -1, najKje = -1, razlik = 0;
    // Z oknom širine n znakov se pomikajmo v desno po obeh nizih in v spremenljivki
    // „razlik“ hranimo število mest (v oknu), kjer se niza razlikujeta.
    for (int i = 0; s[i]; ++i)
    {
        // Desni rob okna premaknimo na znak i.
        if (s[i] != t[i]) ++razlik;

        // Na levem robu zato znak i - n izpade iz okna.
        if (i >= n && s[i - n] != t[i - n]) --razlik;

        // Najboljšo rešitev si zapomnimo.
        if (razlik > najRazlik) najRazlik = razlik, najKje = i;
    }
    // Vrnimo rezultat, vendar indeks na levem koncu okna, ne na desnem.
    return najKje - n + 1;
}
```

Zapišimo to rešitev še v pythonu:

```
def Primerjava(s, t, n):
    najRazlik = -1; najKje = -1; razlik = 0

    # Z oknom širine n znakov se pomikajmo v desno po obeh nizih in v spremenljivki
```

```
# „razlik“ hranimo število mest (v oknu), kjer se niza razlikujeta.
for i in range(len(s)):

    # Desni rob okna premaknimo na znak i.
    if s[i] != t[i]: razlik += 1

    # Na levem robu zato znak i − n izpade iz okna.
    if i >= n and s[i − n] != t[i − n]: razlik −= 1

    # Najboljšo rešitev si zapomnimo.
    if razlik > najRazlik: najRazlik = razlik; najKje = i

# Vrnimo rezultat, vendar indeks na levem koncu okna, ne na desnem.
return najKje − n + 1
```

5. Zlaganje loncev

Največji lonec ne more biti drugje kot na dnu svojega sklada; imeti moramo torej vsaj en sklad s takšnim premerom, kot ga ima največji lonec. Če zdaj pogledamo drugi največji lonec, ga lahko položimo v prvega in tako nadaljujemo isti sklad; podobno položimo tretji največji lonec v drugega in tako naprej. Edino, kar nam lahko pri tem postopku povzroči težave, je, če naletimo na dva ali več loncev z enakim premerom. Ker taki lonci ne gredo eden v drugega, lahko damo na prvi sklad le enega od njih, za ostale pa bomo morali načeti nove sklade (za vsak tak lonec po enega). Pri naslednjem manjšem polmeru lahko lonec spet damo v prvi sklad in tako naprej; sčasoma mogoče spet naletimo na več loncev z enakim premerom in jih damo po vsakega v en sklad; če imamo skladov premalo, pa za preostale take lonce začnemo nove sklade. Tako nadaljujemo, dokler ne razporedimo vseh loncev.

Da bo pregledneje, zapišimo ta postopek še s psevdokodo. V spremenljivki s bomo hranili število skladov, v v pa vsoto polmerov najnižjih loncev v njih. Pri pregledovanju loncev bo p polmer prejšnjega lonca, t pa število doslej pregledanih loncev s tem polmerom.

$v := 0$; $s := 0$; $p := -1$; $t := 0$;
pregleduj lonce padajoče po polmeru:
  naj bo r polmer trenutnega lonca;
  if $r \neq p$ **then** $p := r$, $t := 1$
  else $t := t + 1$;
  (* *To je že t-ti lonec s polmerom r, potrebujemo torej vsaj t skladov. Če jih še nimamo toliko, začnimo nov sklad.* *)
  if $t > s$ **then** $s := s + 1$, $v := v + r$;

Na koncu tega postopka sta s in v rezultata, po katerih sprašuje naloga.

Vidimo lahko, da odpre ta postopek t skladov le, če vidi t loncev z enakim polmerom; na koncu bo torej skladov toliko, kolikor je največ loncev z enakim polmerom, tako da je število skladov res minimalno. Da je minimalna tudi vsota njihovih polmerov, pa se lahko prepričamo takole. Na naš postopek odpira sklade po padajočem (oz. natančneje: nenaraščajočem) polmeru: vsak naslednji sklad ima na dnu kvečjemu tako velik lonec kot prejšnji sklad; in t-ti sklad odpre pri največjem takem polmeru r, pri katerem imamo vsaj t loncev enakega polmera. Če bi bil polmer t-tega največjega sklada manjši od tega r, bi bili polmeri vseh nadaljnjih skladov tudi manjši od r, torej bi obstajalo kvečjemu $t - 1$ skladov s polmerom vsaj r, to pa je premalo za naših (vsaj) t loncev s polmerom r. Tako torej vidimo, da če bi polmer kateregakoli sklada zmanjšali, bi rešitev postala neveljavna, torej naš postopek res najde najmanjšo možno vsoto polmerov.

REŠITVE NALOG ZA DRUGO SKUPINO

1. Sredinec

Ker so pri tej nalogi podane višine v centimetrih in ker učenci niso večji od dveh metrov, je možnih razmeroma malo višin — to so cela števila od 1 do 200. Četudi je učencev na milijone, imajo lahko največ 200 različnih višin; vrsta, v katero se učenci razporejajo v telovadnici, ima torej vedno takšno obliko: najprej nekaj učencev z višino 1, nato nekaj učencev z višino 2, ... in končno nekaj učencev z višino 200. (Pri vsakem od teh „nekaj“ je seveda mogoče tudi, da ni nobenega s tisto višino.) Učencev z enako višino nam ni treba nikakor ločiti med seboj, saj nas zanima vedno le to, kako visok je srednji učenec v vrsti, ne pa, kdo točno je ta srednji učenec. Vrste nam torej ni treba predstaviti s seznamom višin, ki bi vseboval po en element za vsakega učenca, pač pa je dovolj že tabela, ki za vsako možno višino od 1 do 200 pove, koliko učencev s to višino je trenutno v telovadnici. Lepo pri tem je, da ko vstopi nov učenec, moramo le povečati en element te tabele za 1, kar je veliko ceneje, kot če bi hoteli vzdrževati urejen seznam višin vseh učencev in vrivati novega učenca na pravo mesto v tem seznamu.

Višino srednjega, torej $\lceil n/2 \rceil$-tega učenca, bi lahko zdaj določili tako, da bi šli v zanki po višinah od 1 naprej in seštevali število učencev posamezne višine. Pri tisti višini, kjer ta vsota doseže ali preseže $\lceil n/2 \rceil$, vemo, da je v skupini učencev s to višino tudi srednji ($\lceil n/2 \rceil$-ti) učenec in moramo to višino izpisati.

Toda ko je v telovadnici že veliko učencev in jih ima tudi po več enako višino kot srednji učenec, se lahko pogosto zgodi, da ostane višina srednjega učenca nespremenjena tudi po prihodu novega učenca. Na primer: če imamo učence $[10, 20, 20, 20, 30]$, je višina srednjega učenca 20; in če vstopi zdaj en nov učenec, bo višina srednjega še vedno 20 ne glede na višino novega učenca.

Zato je koristno, če višine srednjega ne računamo vsakič znova z zanko po višinah od 1 naprej, ampak le pogledamo, če je treba dosedanjo višino srednjega kaj popraviti. V ta namen si bomo poleg višine srednjega vzdrževali še skupno število učencev, ki so manjši od srednjega; recimo, da je višina srednjega učenca m, da ima tako višino v_m učencev, manjših od te višine pa je $v_{<m}$ učencev. Ko pride nov učenec, za začetek pustimo m pri miru in le povečamo v_m ali $v_{<m}$ za 1, če je novi učenec visok m ali $< m$; za 1 povečamo tudi števec vseh učencev, torej n; nato pa preverimo, ali ni zdaj slučajno $v_{<m} > \lceil n/2 \rceil$ — če je, to pomeni, da je učencev, manjših od m, že preveč, zato m že previsok, da bi bila to lahko višina srednjega učenca; tedaj m zmanjšujmo po 1 (in ustrezno zmanjšujmo $v_{<m}$), dokler ta pogoj ni izpolnjen. Podobno preverimo tudi, ali ni zdaj slučajno $v_{<m} + v_m < \lceil n/2 \rceil$ — če je, to pomeni, da zdaj preveč učencev večjih od m in je višina m prenizka, da bi bila to višina srednjega učenca; tedaj m povečujmo po 1 (in ustrezno povečujmo $v_{<m}$), dokler ta pogoj ni izpolnjen. Tako bomo v večini primerov dobili primerno novo vrednost m že brez popravkov ali pa mogoče z enim povečanjem ali zmanjšanjem za 1.[1]

Oglejmo si še implementacijo te rešitve v C++. Višine učencev bomo brali s standardnega vhoda in po vsakem prebranem učencu izpisali na standardni izhod višino srednjega učenca:

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

int main()
{
  int stZVisino[201] = {};  // št. učencev s posamezno višino
  int mediana = 0;          // višina srednjega učenca
```

[1] Mogoče pa je sestaviti patološke primere, kjer ta izboljšava nič ne pomaga; na primer, če dobimo zaporedje učencev z višinami 200, 1, 200, 1, 200, 1 in tako naprej, nam tudi višina srednjega učenca enako preskakuje z 200 na 1 in nazaj, zato mora naša zanka, ki popravlja m, vsakič pravzaprav iti po vseh možnih višinah. Če bi se hoteli izogniti tej težavi, bi morali imeti način, da preskočimo višine, ki jih nima noben učenec; namesto tabele bi lahko uporabili kakšno (primerno uravnoteženo) drevo, npr. rdeče-črno, ali pa bi vzdrževali par kopic (eno za učence, manjše od sredinskega, in eno za ostale; podoben prijem smo videli že leta 2020 pri 4. nalogi v tretji skupini). Takšne rešitve bi se potem obnesle tudi v primerih, ko višine niso le cela števila od 1 do 200. Z vsakim novim učencem imamo potem $O(\log V)$ dela, če je V število različnih višin med doslej prebranimi učenci.

```
  int stPodMediano = 0;    // št. učencev z višino < mediana
  int n = 0;               // število doslej prebranih učencev

  while (true)
  {
    // Preberimo višino naslednjega učenca.
    int visina; cin >> visina;
    if (! cin.good()) break;

    // Povečajmo števec vseh učencev in učencev te višine.
    ++n; ++stZVisino[visina];

    // Če je manjši od mediane, povečajmo tudi števec takih.
    if (visina < mediana) stPodMediano++;

    // Če je mediana zdaj previsoka, jo zmanjšajmo.
    while (stPodMediano >= (n + 1) / 2)
      stPodMediano -= stZVisino[--mediana];

    // Če pa je mediana zdaj prenizka, jo povečajmo.
    while (stPodMediano + stZVisino[mediana] < (n + 1) / 2)
      stPodMediano += stZVisino[mediana++];

    // Izpišimo višino srednjega učenca.
    cout << mediana << endl;
  }
  return 0;
}
```

Časovna zahtevnost te rešitve je $O(n)$ za obdelavo zaporedja n učencev, saj imamo z vsakim novim le konstantno mnogo dela, neodvisno od n; bolj natančno pa bi morali reči, da je zahtevnost v najslabšem primeru $O(n \cdot V)$, kjer je V število vseh možnih višin — v našem primeru 200.

2. Svetilka

Razmislimo najprej, kakšne globalne spremenljivke bomo potrebovali. Funkcija Tiktak mora vedeti, ali je tipka pritisnjena in kako dolgo, da bo lahko po treh sekundah držanja tipke ugasnila luč. V spodnji rešitvi imamo v ta namen spremenljivki tipkaPritisnjena in casPritiska (ki šteje čas pritiska v desetinkah sekunde), šlo pa bi tudi z eno samo spremenljivko (pri čemer bi npr. vrednost casPritiska == −1 pomenila, da tipka sploh ni pritisnjena). Poleg tega moramo poznati tudi trenutni način delovanja, saj je od tega odvisno, kaj se zgodi ob naslednjem pritisku in ali moramo skrbeti za utripanje. Pri utripanju pa moramo vedeti še, čez koliko časa naj se luč spet prižge; spodnja rešitev ima za to spremenljivko casDoUtripa.

```
typedef enum { Ugasnjena, Sveti, Utripa } Nacin;
Nacin nacin = Ugasnjena;
bool tipkaPritisnjena = false;
int casPritiska, casDoUtripa;
```

Oglejmo si zdaj funkcijo Tipka, ki je enostavnejša. Novo stanje tipke si zapomnimo v tipkaPritisnjena; če je bila tipka spuščena, je to tudi vse, sicer pa določimo novi način luči: če je prej stalno svetila, mora zdaj utripati, sicer pa mora zdaj svetiti. V slednjem primeru lahko luč takoj tudi prižgemo; pri utripanju pa bi bila škoda, če bi jo zdaj prižgali in potem pri naslednjem klicu funkcije Tiktak ugasnili, saj lahko do takrat mine manj kot desetinka sekunde. Namesto tega bomo raje postavili casDoUtripa na 1 in tako zagotovili, da bo luč prižgala funkcija Tiktak ob naslednjem klicu (in jo potem še en klic kasneje spet ugasnila; tako bo luč gotovo gorela eno desetinko sekunde).

```
void Tipka(bool pritisnjena)
{
  // Zapomnimo si novo stanje tipke.
  tipkaPritisnjena = pritisnjena;
  if (! pritisnjena) return;

  // Ob pritisku začnemo meriti čas pritiska.
```

```
  casPritiska = 0;
  // Preklopimo na novo stanje.
  nacin = (nacin == Sveti) ? Utripa : Sveti;
  // Pri preklopu na utripanje bomo prižgali luč v naslednji
  // desetinki namesto takoj, da bomo lažje odmerili čas.
  Luc(nacin == Sveti);
  casDoUtripa = 1;
}
```

Funkcija Tiktak mora skrbeti za utripanje luči in za izklop po treh sekundah držanja na tipko. Za to slednje poskrbimo s števcem casPritiska, ki ga ob vsakem klicu povečamo, ko pa doseže 31, luč ugasnemo. Ker ga je Tipka postavila na 0, ko je uporabnik pritisnil tipko, in ker ne vemo točno, kje v času med dvema klicema funkcije Tiktak je prišel klic Tipka, to pomeni, da se bo luč ugasnila po vsaj treh sekundah (gotovo pa manj kot 3,1 sekundah) držanja na tipko.

Za utripanje poskrbimo tako, da zmanjšujemo števec casDoUtripa; ko pade na 0, prižgemo luč in postavimo števec na 10; ko pa pade števec na 9 (torej eno desetinko sekunde po tistem, ko smo luč prižgali in postavili števec na 10) luč spet ugasnemo. Tako bo luč res gorela eno desetinko sekunde in bo potem devet desetink ugasnjena.

```
void Tiktak()
{
  if (tipkaPritisnjena && casPritiska <= 30)
    // Povečajmo števec, ki meri čas pritiska tipke.
    if (++casPritiska > 30) {
      // Po treh sekundah luč ugasnemo.
      Luc(false); nacin = Ugasnjena; }
  // Poskrbimo za utripanje.
  if (nacin == Utripa)
    // Zmanjšajmo čas do utripa za 1; ko pade na 0, luč prižgemo.
    if (--casDoUtripa == 0) { Luc(true); casDoUtripa = 10; }
    // Ko je do naslednjega utripa še 9 desetink sekunde, luč spet ugasnemo.
    else if (casDoUtripa == 9) Luc(false);
}
```

3. Pletenje puloverja

Recimo, da je shema široka w stolpcev in visoka h vrstic; naj bo $s(x, y)$ znak na preseku x-tega stolpca in y-te vrstice.

Recimo zdaj, da je v shemi prisoten vzorec velikosti $w_p \times h_p$, ki se lepo zaključi na robovih sheme. Iz tega sledi, da je prvih h_p vrstic sheme (ki tvorijo prvo vrsto pojavitev vzorca) enakih naslednjim h_p vrsticam (ki tvorijo drugo vrsto pojavitev vzorca) in potem spet naslednjim h_p vrsticam in tako naprej. Z drugimi besedami, velja torej $s(x, y) = s(x, y - h_p)$ za vse x in y (natančneje povedano: za vse $1 \leq x \leq w$ in $h_p < y \leq h$; tovrstnih pogojev v nadaljevanju ne bomo posebej pisali, jih pa imejmo v mislih). Poleg tega vidimo tudi, in da je višina sheme h večkratnik višine vzorca h_p (torej da h_p deli h), saj se sicer vzorec na spodnjem robu ne bi lepo zaključil, pač pa bi bila zadnja vrsta pojavitev vzorca delno odrezana. Podobno lahko razmišljamo tudi za stolpce, kjer ugotovimo, da velja $s(x, y) = s(x - w_p, y)$ za vse x in y ter da w_p deli w.

Kaj pa obratno? Recimo, da v naši shemi pri nekem w_p, ki deli w, velja $s(x, y) = s(x - w_p, y)$ (za vse x in y) in da pri nekem h_p, ki deli h, velja $s(x, y) = s(x, y - h_p)$ (za vse x in y). Iz prve od teh dveh predpostavk vidimo, da se vsebina pravokotnika $w_p \times h_p$ v zgornjem levem kotu sheme potem spet ponavlja, če jo zamikamo po w_p enot desno; in ker w_p deli w, bomo s ponavljanjem tega pravokotnika zapolnili prvih h_p vrstic mreže po celi širini. Druga predpostavka pa nam potem pove, da se nam vsebina teh prvih h_p vrstic v nadaljevanju ponavlja v vsakih naslednjih h_p vrsticah in (ker h_p deli h) tako sčasoma točno zapolni celo shemo. Tako torej vidimo, da je v shemi prisoten vzorec velikosti $w_p \times h_p$.

Če oba prejšnja odstavka združimo, lahko zaključimo, da je v shemi prisoten vzorec $w_p \times h_p$ natanko tedaj, ko w_p deli w, h_p deli h in ko za vse primerne x in y velja

$s(x,y) = s(x-w_p,y)$ in $s(x,y) = s(x,y-h_p)$. Za to zadnjo skupino pogojev pa vidimo, da se eni nanašajo samo na w_p, eni pa samo na h_p. Tako vidimo, da nam pri iskanju vzorcev ni treba preverjati para (w_p,h_p) skupaj, ampak lahko iščemo primerne w_p posebej in primerne h_p posebej. Najmanjši vzorec — in to je tisti, po katerem nas sprašuje naloga — bomo torej dobili tako, da bomo vzeli najmanjši primerni w_p in najmanjši primerni h_p. Tako se tudi ne bo moglo zgoditi, da bi obstajalo več enako dobrih rešitev; vedno je en sam vzorec najmanjši.

Pojdimo torej v zanki po naraščajočih w_p in pri vsakem najprej preverimo, ali deli w; če je to res, preglejmo, če se vsebina sheme ponavlja na vsakih w_p vrstic. Najmanjši w_p, pri katerem se to izide, je potem širina našega osnovnega vzorca (če ne prej, bo ta pogoj gotovo izpolnjen pri $w_p = w$). Nato podobno naredimo še za h_p, kjer preverjamo, če h_p deli h in če se vsebina sheme ponavlja na vsakih h_p stolpcev.

Oglejmo si implementacijo te rešitve v C++. Ker je preverjanje po vrsticah in po stolpcih zelo podobno, smo si pomagali z zanko z dvema iteracijama; v prvi iščemo najmanjši w_p, v drugi pa najmanjši h_p, vmes pa v mislih zamenjamo vrstice in stolpce, da lahko potem obakrat uporabimo isto kodo.

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

void OsnovniVzorec(const vector<string>& shema)
{
  int w = shema[0].length(), h = shema.size(), w0, h0;
  // Pri smer == 0 iščemo širino osnovnega vzorca, pri smer == 1 pa višino.
  for (int smer = 0; smer < 2; ++smer)
  {
    // Naslednji podprogram prebere en znak sheme.
    auto Znak = [&shema, smer] (int x, int y) { return smer ? shema[x][y] : shema[y][x]; };
    // Če ne bomo našli ožjega, bo osnovni vzorec pokrival celo širino sheme.
    w0 = w;
    // Preiskusimo ožje širine, seveda le take, ki delijo širino sheme.
    for (int d = 1; d < w; ++d) if (w % d == 0)
    {
      // Preverimo, če se vzorec s to širino ponavlja po celi shemi.
      bool ok = true;
      for (int y = 0; y < h && ok; ++y) for (int x = d; x < w; ++x)
        if (Znak(x, y) != Znak(x - d, y)) { ok = false; break; }
      // Če se, smo našli širino osnovnega vzorca.
      if (ok) { w0 = d; break; }
    }
    // Obrnimo osi, da bomo v naslednji iteraciji našli še višino.
    swap(w, h); swap(w0, h0);
  }
  cout << w0 << ' ' << h0 << endl; // Izpišimo rezultate.
}
```

To rešitev bi se dalo še izboljšati s kakšnimi hevristikami, ki bi nam pomagale čim prej in čim ceneje prepoznati neobetavne w_p ali h_p. Recimo, da za vsak stolpec izračunamo nekakšno kontrolno vsoto ali zgoščevalno kodo: $k[1], k[2], \ldots, k[w]$ (če drugega ne, lahko preštejemo ničle v stolpcu in to vzamemo za kontrolno vsoto). Ko nas kasneje zanima, ali se shema ponavlja na vsakih w_p stolpcev, bi lahko za začetek preverili, ali se tako ponavljajo tudi te kontrolne vsote, torej ali velja $k[x] = k[x-w_p]$ za vse x; šele če se to izide, je smiselno preverjati vse znake sheme, kot to počne gornji podprogram. Lahko gremo še korak naprej: če se kontrolne vsote res ponavljajo na vsakih w_p stolpcev, to pomeni, da se vsaka od vsot $k[1], \ldots, k[w_p]$ pojavi (w/w_p)-krat. Če bi torej za vsako različno kodo prešteli, kolikokrat se pojavi, bi morala biti vsa ta števila pojavitev večkratniki vrednosti w/w_p; ali še drugače, w_p je lahko kandidat za širino vzorca le, če w/w_p deli vsa

števila pojavitev kontrolnih vsot stolpcev; to pa pomeni, da mora deliti njihov najmanjši skupni delitelj; slednjemu recimo D. Tega lahko izračunamo na začetku, še preden se začnemo ukvarjati s posameznimi w_p, in potem pri vsakem w_p najprej preverimo, če w_p deli w in če w/w_p deli D; če se to izide, preverimo, ali se kontrolne vsote $k[1], \ldots, k[w]$ ponavljajo na vsakih w_p stolpcev; in šele nato preverimo, ali se tudi vsebina stolpcev ponavlja na vsakih w_p stolpcev. Podobno lahko seveda naredimo tudi pri vrsticah.

4. Pangramski podniz

Recimo, da je naš vhodni niz s dolg n črk, $s = s_1 s_2 \ldots s_n$. Podniz, ki nas zanima, bo oblike $s_i s_{i+1} \ldots s_{j-1} s_j$ za neka i in j. Najkrajši pangramski podniz lahko najdemo tako, da gremo v zanki po vseh možnih j (od 1 do n) in se pri vsakem vprašamo, kateri je najkrajši pangramski podniz, ki se konča pri tem j; to pa je seveda tisti, ki ima največji i. Kaj se dogaja s tem i, torej položajem levega konca podniza, če počasi povečujemo j, torej položaj desnega konca podniza? Če je bil $s_i \ldots s_j$ pangram in če potem premaknemo desni konec na s_{j+1}, bo niz $s_i \ldots s_{j+1}$ še vedno pangram, tako da se prav gotovo ne bo moglo zgoditi, da bi bilo treba kdaj pomakniti i nazaj v levo; mogoče pa je, da bo zdaj pangram tudi $s_{i+1} \ldots s_{j+1}$ in da smemo torej premakniti i v desno (in podniz tako še kaj skrajšati).

Tako imamo torej naslednji postopek: povečujemo j za 1 in po vsakem povečanju j-ja pogledamo, kako daleč smemo še povečati i, ne da bi podniz $s_i \ldots s_j$ prenehal biti pangram. Med vsemi tako dobljenimi podnizi si zapomnimo dolžino najkrajšega in jo na koncu vrnemo.

Stvar se malo zaplete le na začetku, kjer se lahko zgodi, da pri kakšnem j sploh ni nobenega pangrama, ker mogoče niti pri $i = 1$ podniz $s_i \ldots s_j$ še ne vsebuje vsake črke vsaj k-krat (prav gotovo se to zgodi npr. pri $j < 26 \cdot k$). Na začetku moramo torej pustiti i na 1 (da se podniz začne na začetku niza s) in povečevati j tako dolgo, dokler $s_1 \ldots s_j$ ne postane pangram (mogoče je tudi, da se to ne zgodi nikoli, ker morda niti celoten s ni pangram).

Vprašanje je še, kako lahko poceni preverjamo, ali je opazovani podniz $s_i \ldots s_j$ pangram ali ne. V ta namen je koristno vzdrževati tabelo, ki za vsako črko abecede vsebuje število pojavitev te črke v podnizu. Ko povečamo j za 1, pride v podniz nova črka in zato ustrezni element tabele povečamo za 1; ko pa povečamo i za 1, izpade ena črka iz podniza in zato ustrezni element tabele zmanjšamo za 1.

Podniz je pangram, če se vsaka črka pojavlja vsaj k-krat. Ko torej po vsakem povečanju j-ja razmišljamo o tem, ali smemo zdaj tudi i povečati za 1, vidimo, da ga smemo povečati, če se črka s_i pojavlja v podnizu več kot k-krat, saj bo v tem primeru podniz ostal pangram, četudi eno pojavitev te črke izgubimo.

Na začetku, ko je nekaterih črk manj kot k, naš podniz sploh še ni pangram; da bomo lažje ugotovili, kdaj postane pangram, je koristno vzdrževati še podatek o tem, koliko črk abecede se pojavlja manj kot k-krat. V spodnjem podprogramu imamo v ta namen spremenljivko premalo; vzdrževati je ni težko: ko se število pojavitev neke črke poveča s $k-1$ na k, zmanjšamo premalo za 1; in ko pade premalo na 0, vemo, da je naš podniz pangram (in bo odtlej to tudi ostal).

```
int PangramskiPodniz(const char *s, int k)
{
  enum { Abeceda = 26 };
  int n[Abeceda] = { };     // število pojavitev vsake črke v podnizu s[i..j]
  int premalo = Abeceda;  // koliko črk ima manj kot k pojavitev
  int naj = -1;             // najboljša rešitev doslej
  for (int i = 0, j = 0, c; s[j]; ++j)
  {
    // Trenutno imamo števila pojavitev črk v s[i...j-1].
    // Popravimo jih, da se bodo nanašala na s[i...j].
    if (++n[s[j] - 'a'] == k) --premalo;

    // Če se kakšna črka pojavlja premalokrat, bomo morali podniz
    // na desni še podaljšati.
    if (premalo > 0) continue;

    // Mogoče lahko levi konec podniza premaknemo proti desni:
```

```
    // če ima črka s[i] več kot k pojavitev, jo smemo vreči iz podniza.
    while (n[c = s[i] - 'a'] > k) ++i, --n[c];

    // Če je to najboljša rešitev doslej, si jo zapomnimo.
    if (naj < 0 || j - i + 1 < naj) naj = j - i + 1;
  }
  return naj; // Vrnimo najboljšo rešitev.
}
```

5. Tetris

Nalogo lahko rešujemo z rekurzijo. Ploščo bomo pokrivali sistematično: na vsakem koraku bomo poiskali najvišje nepokrito polje (če je takih več, pa najbolj levo med njimi) in ga na vse možne načine poskušali pokriti z enim od še razpoložljivih ploščkov. Pri tem moramo najprej preveriti, ali je plošček sploh mogoče postaviti tja; če da, ga postavimo in nadaljujemo z rekurzivnim klicem, ki bo poskušal s postavljanjem ostalih ploščkov do konca zapolniti ploščo. Če se je to posrečilo, lahko končamo, sicer pa pravkar postavljeni plošček spet odstranimo, da bomo poskusili še s kakšnim drugim.

Med rekurzijo je torej koristno imeti podatke o tem, do kod v plošči smo že pokrili vsa polja, tako da bomo lahko z iskanjem prvega nepokritega nadaljevali od tam in nam ne bo treba iti vsakič znova od začetka mreže. Poleg tega potrebujemo tudi tabelo oz. vektor števil, ki nam povedo, koliko ploščkov posamezne oblike je še na voljo (torej da jih še nismo položili na ploščo); ko položimo plošček, zmanjšamo ustrezni števec v tem vektorju, ko pa ga poberemo s plošče, njegov števec spet povečamo.

```
enum { W = 8, H = 8 }; // širina in višina plošče

// Funkcija vrne true, če je uspela v celoti pokriti ploščo.
// Sicer vrne false, plošča pa je ob vrnitvi iz funkcije v enakem stanju
// kot na začetku klica.
bool NadaljujPokrivanje(int x, int y, vector<int> &prosti)
{
  // Poiščimo naslednje nepokrito polje.
  while (y < H && JePokrito(x, y))
    if (++x == W) x = 0, ++y;
  if (y >= H) return true; // Če je vse že pokrito, smo končali.

  // Na vse možne načine ga poskusimo pokriti.
  for (int oblika = 0; oblika < prosti.size(); ++oblika)
  {
    // Ali lahko sem postavimo plošček te oblike?
    if (prosti[oblika] <= 0) continue;
    if (! PreveriPloscek(oblika, x, y)) continue;

    // Postavimo ga in nadaljujmo z rekurzijo.
    PostaviPloscek(oblika, x, y, true); --prosti[oblika];
    if (NadaljujPokrivanje(x, y, prosti)) return true;

    // Če nismo uspeli pokriti cele plošče, plošček spet odstranimo.
    PostaviPloscek(oblika, x, y, false); ++prosti[oblika];
  }
  return false; // Če pridemo do sem, se plošče ni dalo pokriti do konca.
}
```

Glavna funkcija mora le inicializirati vektor z začetnim številom ploščkov vsake oblike in pognati rekurzijo od začetka mreže (torej od zgornjega levega kota):

```
bool PokrijPlosco()
{
  int n = StOblik();
  vector<int> prosti(n);
  for (int oblika = 0; oblika < n; ++oblika) prosti[oblika] = StPlosckov(oblika);
  return NadaljujPokrivanje(0, 0, prosti);
}
```

Ob vrnitvi iz funkcije je plošča bodisi v celoti pokrita (tedaj funkcija vrne **true**) bodisi v enakem stanju kot na začetku, torej prazna (če se je sploh ni dalo pokriti; tedaj funkcija vrne **false**).

REŠITVE NALOG ZA TRETJO SKUPINO

1. Kapniki

Preprosta, a neučinkovita rešitev je, da gremo v zanki po vseh možnih višinah železnice (od 1 do v) in pri vsaki od njih s še eno vgnezdeno zanko pregledamo vseh n kapnikov ter preštejemo, koliko od njih bi bilo treba pri tej višini železnice podreti. Ta rešitev ima časovno zahtevnost $O(nv)$ in bi pri testnih primerih z našega tekmovanja dobila 20 % točk.

Poskusimo to rešitev izboljšati. Kako se spreminja število motečih kapnikov, ko v zanki počasi dvigujemo železnico za 1? Stalagmit višine k_i nam je v napoto pri $1 \leq y \leq k_i$, stalaktit višine k_i pa pri $v - k_i + 1 \leq y \leq v$. Na začetku, na višini $y = 1$, nas motijo vsi stalagmiti in noben stalaktit; nato pa, ko počasi dvigujemo železnico, nam je počasi v napoto vse manj stalagmitov in vse več stalaktitov, dokler nas na koncu pri $y = v$ ne motijo vsi stalaktiti in noben stalagmit. Ko dvignemo železnico z višine $y - 1$ na višino y, nas nehajo motiti stalagmiti višine $y - 1$, začnejo pa nas motiti stalaktiti višine $v - y + 1$.

Lahko bi imeli torej tabelo, v kateri bi za vsako možno višino označili, koliko kapnikov nas tam začne ali neha motiti; to tabelo lahko pripravimo med branjem kapnikov, nato pa gremo v zanki po vseh višinah od 1 do v in te spremembe v številu motečih kapnikov seštevamo, pa bomo v vsakem trenutku točno vedeli, koliko kapnikov nas na tej višini moti. Ta rešitev porabi $O(n + v)$ časa in prostora, kar bo pri naših testnih primerih dovolj za 60 % točk.

Opazimo pa lahko, da ko pregledujemo možne višine železnice od 1 do v, lahko posamezni kapnik povzroči spremembo le pri eni višini: pri tisti, kjer se začne (če je stalaktit) ali konča (če je stalagmit). Čeprav imamo lahko morda 10^{18} višin, je možnih sprememb le toliko kot kapnikov, kar je največ 10^5. Na območju med dvema zaporednima spremembama je število motečih kapnikov ves čas enako, zato se nam ni treba ukvarjati z vsako višino na tem območju posebej.

Pripravimo si torej seznam parov (y, d), ki povedo, da se število motečih kapnikov zmanjša za d, ko višina železnice naraste z $y - 1$ na y. Stalagmit višine k_i torej v ta seznam prispeva par $(k_i + 1, 1)$, stalaktit višine k_i pa par $(v - k_i + 1, -1)$. Nazadnje dodajmo v seznam še par $(v + 1, 0)$, ki ponazarja konec jame, tako da bomo lahko upoštevali tudi višine od zadnje spremembe v številu motečih kapnikov do stropa jame $(y = v)$.

Nato te pare uredimo, tako da tisti z isto višino pridejo skupaj, med njimi pa pridejo najprej tisti za stalaktite (ki število motečih kapnikov povečujejo), nato pa tisti za stalagmite (ki to število zmanjšujejo). Tako bomo, če pride do več sprememb na isti višini, najprej prekomerno povečali število motečih kapnikov, ko bomo prištevali stalaktite, in ga nato zmanjšali na pravo vrednost, ko bomo odštevali stalagmite. Na ta način ne bomo nikoli imeli premajhnega števila motečih kapnikov, na koncu take skupine sprememb na isti višini pa bomo imeli točno pravo število motečih kapnikov na tej višini. Tako lahko po vsaki spremembi preverimo, če je trenutno število motečih kapnikov najnižje doslej, in če je, si ga zapomnimo. Razlika v višini med prejšnjo in trenutno spremembo pa nam pove, na koliko višinah velja tisto število motečih kapnikov, ki je veljalo pred trenutno spremembo; to bo prišlo prav, ker moramo izpisati ne le najmanjšega možnega števila motečih kapnikov, ampak tudi to, na koliko višinah ga dosežemo.

Ta rešitev ima časovno zahtevnost $O(n \log n)$, namreč zaradi urejanja sprememb; vse ostalo je $O(n)$.

```
#include <iostream>
#include <utility>
#include <algorithm>
#include <vector>
#include <string>
using namespace std;
typedef long long int llint;

int main()
{
```

```
  // Preberimo število in tip kapnikov.
  llint v; int n, m = 0; string s; cin >> v >> n >> s;

  // Preberimo višine kapnikov in pripravimo tabelo parov (y, d), ki povedo, da nas neki
  // kapnik na novo (d < 0 ? začne : neha) motiti, ko višina železnice zraste na y.
  vector<pair<llint, int>> spremembe(n + 1);
  for (int i = 0; i < n; ++i)
  {
    llint ki; cin >> ki;

    // Če je to stalagmit, nas neha motiti pri višini ki + 1.
    if (s[i] == 'M') ++m, spremembe[i] = {ki + 1, 1};

    // Če je stalaktit, nas začne motiti pri višini v − ki + 1.
    else spremembe[i] = {v − ki + 1, −1};
  }
  spremembe[n] = {v + 1, 0};

  // Pregledujmo spremembe po naraščajoči višini. „m“ pove, koliko
  // kapnikov nas moti; trenutno (pri y = 1) so to vsi stalagmiti.
  sort(spremembe.begin(), spremembe.end());
  int naj = n + 1; llint koliko = 0;
  for (int i = 0; i <= n; i++)
  {
    llint dv = spremembe[i].first − (i == 0 ? 1 : spremembe[i − 1].first);

    // Od prejšnje spremembe do trenutne je dv možnih višin, kjer nas moti m kapnikov.
    if (m < naj) naj = m, koliko = dv; // Nova najboljša rešitev.
    else if (m == naj) koliko += dv;   // Izenačena dosedanja najboljša rešitev.

    // Upoštevajmo spremembo v številu motečih kapnikov.
    m −= spremembe[i].second;
  }
  printf("%d %lld\n", naj, koliko); return 0;
}
```

Namesto urejanja parov (y, d) bi lahko uporabili kakšno primerno uravnoteženo drevesasto podatkovno strukturo, na primer rdeče-črno drevo (v C++ lahko uporabimo razred `map` iz standardne knjižnice); kot ključe v njem bi uporabili vrednosti y, pripadajoča vrednost pa bi bilo število, ki pove, za koliko se pri tem y spremeni število kapnikov, ki so nam v napoto. Na koncu se lahko s pomočjo te strukture sprehodimo po vseh takih višinah v naraščajočem vrstnem redu in sproti primerno popravljamo število kapnikov, ki so nam trenutno v napoto. Časovna zahtevnost te rešitve je še vedno $O(n \log n)$, ker nam vsaka operacija na drevesu vzame $O(\log n)$ časa.

2. Socialno omrežje

Vhodne podatke si lahko predstavljamo kot neusmerjen graf, pravzaprav dva grafa na isti množici točk (točke predstavljajo ljudi v omrežju): v enem povezave predstavljajo prijateljstva, v drugem pa sovraštva.

V grafu prijateljstev poiščimo povezane komponente, torej skupine točk, za katere velja, da je mogoče iz vsake točke v skupini priti po prijateljskih povezavah do vsake druge točke v skupini in da ni mogoče v skupino dodati nobene točke, ne da bi ta pogoj prenehal veljati. Tega ni težko narediti z iskanjem v širino: začnemo v poljubni točki in z iskanjem v širino obiščemo vse, ki so dosegljive iz nje; to je zdaj ena povezana komponenta; nato začnemo v poljubni taki točki, ki je doslej še nismo obiskali, in poženemo iskanje v širino iz nje; dobimo drugo komponento; in tako naprej, dokler niso obiskane vse točke.

Če dve točki pripadata isti povezani komponenti, to pomeni, da je mogoče od ene do druge priti po verigi prijateljstev, na primer: $u_0 \to u_1 \to \ldots \to u_{k-1} \to u_k$. Če velja prvi pogoj iz naloge (prijatelj mojega prijatelja je tudi moj prijatelj), to pomeni, da mora biti u_0 tudi prijatelj osebe u_2 (ker je u_2 prijatelj u_1, ta pa prijatelj u_0); in zato tudi osebe u_3 (ker je u_3 prijatelj u_2, ta pa prijatelj u_0); in tako naprej vse do u_k. Vidimo torej, da morajo biti vsi člani iste komponente prijatelji drug drugega. Če ima komponenta t članov, mora imeti torej $t \cdot (t-1)/2$ prijateljstev. Pogoj lahko torej

preverimo tako, da za vsako komponento preštejemo, koliko prijateljstev obstaja med člani te komponente, in če jih je manj kot $t(t-1)/2$, potem vemo, da omrežje ne ustreza prvemu pogoju. (Pri tem pazimo na to, da gre lahko t do 10^5, zato je vrednost $t(t-1)/2$ prevelika za 32-bitne celoštevilske tipe; bodisi moramo računati s 64-bitnimi števili ali pa preverjati ta pogoj kako drugače: na primer deliti dvakratnik števila povezav s t in preveriti, če dobimo količnik $t-1$ in ostanek 0.)

Recimo zdaj, da smo prvi pogoj že preverili in da mu omrežje ustreza. Drugi pogoj (sovražnik mojega prijatelja je tudi moj sovražnik) zdaj zahteva, da če sta u in v prijatelja, potem morata imeti popolnoma enako množico sovražnikov, kajti če je neki w sovražnik u-ja, mora biti po tem pogoju tudi sovražnik v-ja in obratno. Še več: če nekaj ljudi pripada isti komponenti v grafu prijateljstev, so si torej vsi tudi paroma prijatelji med sabo (to nam je zagotovil prvi pogoj, ki smo ga že preverili) in morajo imeti vsi popolnoma enako množico sovražnikov.

Za vsako komponento gremo lahko torej v zanki po njenih članih in nato v vgnezdeni zanki po sovražnikih vsakega člana; za vsakega sovražnika štejemo, pri koliko članih komponente smo ga srečali; na koncu mora biti to število pri vsakem sovražniku enako številu ljudi v komponenti, sicer očitno niso imeli vsi člani komponente enake množice sovražnikov.

Razmislimo zdaj še o tretjem pogoju (sovražnik mojega sovražnika je moj prijatelj). To pomeni, da če ima neki človek u dva sovražnika, v in w, je torej w sovražnik v-jevega sovražnika, torej morata biti v in w prijatelja. Z drugimi besedami, vsi u-jevi sovražniki morajo biti med seboj prijatelji; ker že vemo, da velja prvi pogoj, je dovolj, če preverimo, ali vsi u-jevi sovražniki pripadajo isti povezani komponenti grafa prijateljstev; takrat vemo, da so tudi vsi med seboj drug drugemu prijatelji.

Časovna zahtevnost te rešitve je $O(n+m)$, saj ne počnemo drugega, kot da nekajkrat (konstantno mnogokrat) pregledamo vse točke in vse povezave grafa.

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

bool ObdelajOmrezje()
{
  // Preberimo podatke o omrežju.
  int n, m; scanf("%d %d", &n, &m);
  vector<vector<int>> P(n), S(n); // prijatelji in sovražniki vsakega človeka
  for (int i = 0; i < m; ++i) {
    int ai, bi, pi; scanf("%d %d %d", &ai, &bi, &pi);
    auto &V = pi ? P : S; V[--ai].push_back(--bi); V[bi].push_back(ai); }

  // Poiščimo povezane komponente v grafu prijateljstev.
  // kompTocke[u] = komponenta, ki ji pripada točka u;
  // komp[i] = seznam točk v komponenti i.
  int nKomp = 0; vector<int> kompTocke(n, -1); vector<vector<int>> komp;
  for (int u0 = 0; u0 < n; ++u0) if (kompTocke[u0] < 0)
  {
    // Točke u0 doslej še nismo videli; začnimo zanjo novo komponento.
    kompTocke[u0] = nKomp; komp.push_back({});
    auto &vrsta = komp.back(); vrsta.push_back(u0); int glava = 0;
    while (glava < vrsta.size()) {
      int u = vrsta[glava++];
      // Dodajmo v komponento u-jeve prijatelje (če jih še nismo).
      for (int v : P[u]) if (kompTocke[v] < 0) {
        kompTocke[v] = nKomp; vrsta.push_back(v); } }
    ++nKomp;
  }

  // Preverimo, če je prijatelj našega prijatelja vedno tudi naš prijatelj.
  // To pomeni, da morajo biti v vsaki komponenti prisotne vse možne povezave;
  // komponenta s k točkami mora imeti k (k - 1) / 2 povezav.
  vector<int> stPovezav(nKomp, 0); // št. povezav v vsaki komponenti, štetih dvojno
  for (int u = 0; u < n; ++u) stPovezav[kompTocke[u]] += P[u].size();
```

```
  for (int i = 0; i < nKomp; ++i) { int M = komp[i].size();
      if (stPovezav[i] % M != 0 || stPovezav[i] / M != M - 1) return false; }

  // Preverimo, če je sovražnik našega prijatelja vedno tudi naš sovražnik.
  // To pomeni, da morajo imeti vsi ljudje v posamezni komponenti popolnoma enako
  // množico sovražnikov.
  vector<int> stSovraznikov(n, 0), priKomp(n, -1);
  for (int i = 0; i < nKomp; ++i)
  {
    auto &K = komp[i];
    int nS = S[K[0]].size();

    // Vsi člani komponente morajo imeti enako število sovražnikov.
    for (int u : K) if (S[u].size() != nS) return false;

    // stSovraznikov[v] naj bo število sovražnikov, ki jih ima v v komponenti i.
    // Da ne bo treba pri vsaki komponenti inicializirati cele tabele, si v
    // priKomp[v] zapomnimo, pri kateri komponenti smo točko v nazadnje videli.
    for (int u : K) for (int v : S[u])
      if (priKomp[v] != i) priKomp[v] = i, stSovraznikov[v] = 1;
      else ++stSovraznikov[v];

    // Preverimo, če je vsakdo, ki je sovražnik vsaj enega člana komponente,
    // tudi sovražnik vseh članov komponente.
    for (int u : K) for (int v : S[u])
      if (stSovraznikov[v] != K.size()) return false;
  }

  // Preverimo, če je sovražnik našega sovražnika vedno naš prijatelj.
  // To pomeni, da morajo biti vsi sovražniki posameznega človeka med seboj
  // prijatelji, torej biti v isti povezani komponenti.
  for (int u = 0; u < n; ++u) for (int v : S[u])
    if (kompTocke[v] != kompTocke[S[u][0]]) return false;

  return true;
}

int main()
{
  // Preberimo število omrežij in jih obdelajmo v zanki.
  int t; scanf("%d", &t);
  while (t-- > 0) printf("%s\n", ObdelajOmrezje() ? "DA" : "NE");
  return 0;
}
```

3. Proizvodnja cepiva

V tej rešitvi bomo besedo *proizvodnja* vedno uporabljali za skupno količino proizvedenega cepiva do vključno določenega dne, številu opravljenih nadgradenj do vključno določenega dne pa bomo rekli *zmogljivost* (ker določa, koliko cepiva bomo odtlej lahko proizvedli v enem dnevu).

Za začetek je koristno urediti omejitve (x_i, y_i) naraščajoče po času x_i. Če imamo več omejitev z istim rokom x_i, obdržimo le tisto z najvišjo zahtevano proizvodnjo y_i, saj bo vsak razpored nadgradenj, ki ustreza tej omejitvi, ustrezal tudi tistim z istim rokom in nižjo zahtevano proizvodnjo. Podobno, če opazimo, da ima naslednja omejitev sicer kasnejši rok kot prejšnja, vendar nižjo ali enako zahtevano proizvodnjo, lahko to naslednjo omejitev pobrišemo, saj bo vsak razpored, ki bo ustrezal prejšnji, s tem že ustregel tudi tej naslednji. V nadaljevanju torej predpostavimo, da smo omejitve na ta način uredili in prečistili, tako da zdaj velja $0 < x_1 < x_2 < \ldots < x_d$ in $0 < y_1 < y_2 < \ldots < y_d$. Za potrebe opisa naše rešitve si mislimo še $x_0 = y_0 = 0$.

Recimo, da imamo dva zaporedna dneva, t in $t+1$, da dan t ni rok nobene omejitve in da na dan t proizvajamo cepivo, naslednji dan pa nadgrajujemo tovarno. Če bi to dvoje zamenjali in raje nadgrajevali na dan t, proizvajali pa na dan $t+1$, bi bila zmogljivost na koncu teh dveh dni enaka kot pred to zamenjavo, proizvodnja pa za 1 višja, ker bi na dan $t+1$ po nadgradnji proizvedli eno enoto več kot v prvotnem scenariju na dan t pred nadgradnjo. To, da se je proizvodnja, do katere bi sicer prišlo na dan t, zdaj zamaknila

na dan $t+1$, bi lahko bila težava le, če bi na dan t padel rok kakšne omejitve, kar pa se, kot smo predpostavili, ne zgodi.

Roki omejitev nam razdelijo časovno premico na intervale, pri čemer r-ti omejitvi ustreza interval, ki ga tvorijo dnevi $x_{r-1}+1, \ldots, x_r$. Dolžino r-tega intervala označimo s $t_r := x_r - x_{r-1}$. Če imamo nek veljaven razpored nadgradenj (tak, ki ustreza vsem omejitvam) in če znotraj nekega intervala ne nastopijo vse nadgradnje na začetku intervala, lahko razpored izboljšamo, če jih premaknemo na začetek intervala, saj smo v prejšnjem odstavku videli, da se pri tem proizvodnja poveča, zmogljivost pa ostane enaka, torej bo tako spremenjen razpored še vedno ustrezal omejitvam. Ni se nam torej treba bati, da bi spregledali optimalno rešitev, če se v nadaljevanju omejimo na take razporede, ki znotraj vsakega intervala vedno najprej nekaj dni (lahko tudi 0 dni) samo nadgrajujejo, potem pa odtlej do konca intervala samo proizvajajo.

Pri iskanju optimalnega razporeda vsaj do konca zadnjega intervala, se pravi za prvih x_d dni (z vprašanjem, kako od tam potem najhitreje priti do proizvodnje k, se bomo ukvarjali malo kasneje), je torej vprašanje predvsem, koliko nadgradenj izvesti v vsakem od teh intervalov — to, *kdaj* jih izvesti, pa smo že videli: na začetku vsakega intervala. Na misel nam lahko pride, da bi razpored gradili postopoma: najprej izberemo število nadgradenj v prvem intervalu, nato v drugem in tako naprej. Toda kaj je pravzaprav najboljše število nadgradenj v prvem intervalu? To ni nujno tisto, ki maksimizira proizvodnjo v prvem intervalu; včasih je bolje posvetiti več časa nadgrajevanju in proizvesti le toliko, kolikor je nujno treba, da dosežemo omejitev y_1, zato pa imamo potem višjo zmogljivost za kasnejšo proizvodnjo. Po drugi strani tudi ni nujno najbolje izvesti največjega možnega števila nadgradenj, s katerim še lahko dosežemo omejitev y_1, ker bo tako dosežena proizvodnja mogoče preslabo izhodišče za naslednjo omejitev (ki ima mogoče rok kmalu za prvo, vendar občutno višjo y_2). Ker torej vnaprej ne moremo vedeti, koliko nadgradenj je pametno izvesti v prvem intervalu, bomo preizkusili vse možnosti in si za vsako zapomnili doseženo proizvodnjo.

Ker velja podoben razmislek tudi pri kasnejših intervalih, si zastavimo podprobleme takšne oblike: naj bo $f_r(z)$ največja proizvodnja, ki jo je mogoče doseči do konca r-tega intervala (torej v prvih x_r dneh), če moramo pri tem ustreči prvim $r-1$ omejitvam in opraviti natanko z nadgradenj. Če ta problem sploh ni rešljiv, si mislimo $f_r(z) = -\infty$. Tudi če je rešljiv, še ni nujno, da doseže proizvodnjo y_r, ki jo zahteva r-ta omejitev; definirajmo $z_r^{\min}$ in $z_r^{\max}$ kot najmanjšo in največjo vrednost z-ja, pri kateri je $f_r(z) \geq y_r$.

Ko rešujemo tak podproblem — torej ko računamo vrednost $f_r(z)$ — se moramo med drugim odločiti, koliko nadgradenj bi opravili v zadnjem (torej r-tem) intervalu; recimo, da jih opravimo u. V prejšnjih $r-1$ intervalih moramo torej opraviti $z-u$ nadgradenj in ustreči vsem tamkajšnjim omejitvam; največja proizvodnja, ki jo je mogoče pri tem doseči, je $f_{r-1}(z-u)$. V r-tem intervalu potem porabimo u dni za nadgradnje, s čimer dosežemo zmogljivost z, tako da potem v preostalih $t_r - u$ dneh tega intervala proizvedemo še $(t_r - u) \cdot z$ enot. Tako dosežemo proizvodnjo

$$F_r(z,u) := f_{r-1}(z-u) + (t_r - u) \cdot z.$$

Ker hočemo za $f_r(z)$ čim večjo proizvodnjo, bomo seveda vzeli tak u, pri katerem je $F_r(z,u)$ čim večja. Toda kateri u sploh pridejo v poštev? Seveda mora biti $0 \leq u \leq t_r$, saj ima r-ti interval le t_r dni; poleg tega pa mora biti $z_{k-1}^{\min} \leq z-u \leq z_{k-1}^{\max}$, saj drugače razpored za prvih $r-1$ intervalov z $z-u$ nadgradnjami sploh ne more ustreči prvim $r-1$ omejitvam.[2] Tako lahko zaključimo:

$$f_r(z) = \max\{F_r(z,u) : \max(0, z - z_{r-1}^{\max}) \leq u \leq \min(t_r, z - z_{r-1}^{\min})\}.$$

[2]Načeloma bi morali dodati še pogoj, da mora biti $f_{r-1}(z-u) \geq y_{r-1}$, saj drugače razpored, ki ga dobimo pri tem u, ne bo ustrezal omejitvi $r-1$. Toda kot bomo videli kasneje, je funkcija f_{r-1} konkavna, kar pomeni, da najprej nekaj časa samo narašča, od nekega trenutka naprej pa ves čas samo še pada. Spomnimo se, da ima f_{r-1} pri $z_{k-1}^{\min}$ in $z_{k-1}^{\max}$ vrednost vsaj y_r ter da $z-u$ leži nekje na območju od $z_{k-1}^{\min}$ do $z_{k-1}^{\max}$; torej, če bi bila $f_{r-1}(z-u) < y_r$, bi to pomenilo, da je funkcija f_{r-1} od $z_{r-1}^{\min}$ do $z-u$ nekje padala, kasneje pa je od $z-u$ do $z_{r-1}^{\max}$ nekje naraščala. To pa je pri konkavni funkciji nemogoče, saj taka funkcija, ko enkrat začne padati, kasneje nikoli več ne narašča. Toda tudi če tega razmisleka ne opravimo, lahko v našem programu, ko bomo z zanko pregledovali možne u, pri vsakem brez težav še pogledamo, če zanj velja $f_{r-1}(z-u) \geq y_{r-1}$; ta pogoj bo sicer vedno izpolnjen in z njim bomo le zapravili nekaj časa, vendar ne toliko, da bi se nam bilo treba zaradi tega vznemirjati.

Za katere z ima sploh smisel razmišljati o tem? Z manj kot $z_{r-1}^{\min}$ nadgradnjami ne moremo ustreči niti prvim $r-1$ omejitvam, torej tudi prvim r omejitvam ne; po drugi strani, če bi hoteli več kot $z_{r-1}^{\max}+t_r$ nadgradenj, bi lahko porabili za nadgradnje celoten r-ti interval (t_r dni), pa bi jih še vedno ostalo več kot $z_{r-1}^{\max}$ za prejšnjih $r-1$ intervalov, s čimer spet ne bo mogoče ustreči prvim $r-1$ omejitvam. Tako torej vidimo, da je $f_r(z)$ smiselno računati za $z_{r-1}^{\min} \le z \le z_{r-1}^{\max}+t_r$. (Tema dvema mejama recimo $\hat{z}_r^{\min} := z_{r-1}^{\min}$ in $\hat{z}_r^{\max} = z_{r-1}^{\max}+t_r$; to bo prišlo prav kasneje. Zunaj tega območja je f_r nedefinirana, kajti z manj kot $\hat{z}_r^{\min}$ nadgradnjami ni mogoče ustreči prvim $r-1$ omejitvam, pri več kot $\hat{z}_r^{\max}$ nadgradnjah pa nam, četudi v intervalu r ves čas samo nadgrajujemo, še vedno ostane preveč nadgradenj, da bi jih lahko izvedli v prvih $r-1$ intervalih in pri tem ustregli prvim $r-1$ omejitvam.)

Vidimo, da za izračun funkcije f_r potrebujemo vrednosti funkcije f_{r-1}, zato je koristno te funkcije računati naraščajoče po r. Pri vsakem r gremo v gnezdeni zanki po z in v še eni po u, izračunane vrednosti $f_r(z)$ pa shranjujemo v tabelo, da nam bodo pri roki kasneje za izračun funkcije f_{r+1}. Začnemo pa pri $r=0$, kjer imamo le $z=0$ in $f_0(0)=0$.

Ta postopek se konča, če ne prej, takrat, ko pridemo do konca omejitev in izračunamo tudi funkcijo f_d; tedaj nam ostane še vprašanje, kako od tam čim hitreje priti do želene končne proizvodnje k. Lahko pa se že prej pri nekih r in z zgodi, da dobimo $f_r(z) \ge k$; to pomeni, da lahko že v r-tem intervalu dosežemo zahtevano proizvodnjo k, s tem pa gotovo ustrežemo tudi vsem preostalim omejitvam $(y_r, \ldots, y_d)$, saj besedilo naloge pravi, da za vse omejitve velja $y_i \le k$. V tem primeru se lahko s trenutno omejitvijo in vsemi preostalimi nehamo ukvarjati in se delamo, da je bilo omejitev le $r-1$ (v mislih postavimo d na $r-1$) ter takoj skočimo na vprašanje, kako nadaljevati po tej zadnji omejitvi, da bomo čim hitreje dosegli proizvodnjo k.

Recimo torej zdaj, da smo ustregli vsem omejitvam in v prvih x_d dneh izvedli z nadgradenj ter proizvedli $f_d(z)$ enot cepiva. Kako od tu čim hitreje doseči proizvodnjo k — z drugimi besedami, kako čim prej proizvesti še $k-f_d(z)$ enot? Ker nas od tu naprej ne vežejo več roki omejitev, je tudi zdaj smiselno najprej nekaj časa le nadgrajevati, nato pa le proizvajati. Če porabimo u dni za nadgradnje, bomo imeli zmogljivost $z+u$ in za izdelavo $k-f_d(z)$ enot bomo potrebovali še $\lceil (k-f_d(z))/(z+u) \rceil$ dni. Skupaj z u dnevi za nadgradnje smo tako porabili $g(u) := u + \lceil (k-f_d(z))/(z+u) \rceil$ dni. Iščemo seveda tak u, pri katerem bo $g(u)$ najmanjša. Preprosta rešitev je, da v zanki povečujemo u po 1, računamo $g(u)$ in si zapomnimo najmanjšo doslej doseženo vrednost te funkcije; recimo ji $g^\star$. Sčasoma nam u doseže $g^\star$ in takrat vemo, da bo odtlej $g(u) > u \ge g^\star$, torej odtlej ne bomo več našli rešitve, boljše od $g^\star$ (z drugimi besedami, pri tem u bi porabili že samo za nadgradnje toliko časa, kolikor ga pri najboljši rešitvi porabimo za nadgradnje in proizvodnjo skupaj).

Pri razmisleku v prejšnjem odstavku smo začeli z zmogljivostjo z in proizvodnjo $f_d(z)$. Ker vnaprej ne moremo vedeti, katera z bo dala najboljšo rešitev, moramo seveda v zanki preizkusiti vse (od $z_d^{\min}$ do $z_d^{\max}$).

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

int main()
{
  // Preberimo vhodne podatke.
  int k, d, r; scanf("%d %d", &k, &d);
  struct Omejitev { int x, y; };
  vector<Omejitev> omejitve; omejitve.reserve(d + 1); omejitve.resize(d);
  for (auto &O : omejitve) scanf("%d %d", &O.x, &O.y);
  omejitve.push_back({0, 0}); ++d; // Dodajmo še "omejitev"(0, 0) na začetku.

  // Uredimo omejitve po času.
  sort(omejitve.begin(), omejitve.end(), [] (const auto & a, const auto & b) {
    return a.x < b.x || a.x == b.x && a.y > b.y; });

  // Izračunajmo dosegljiva stanja na koncu vsakega intervala.
```

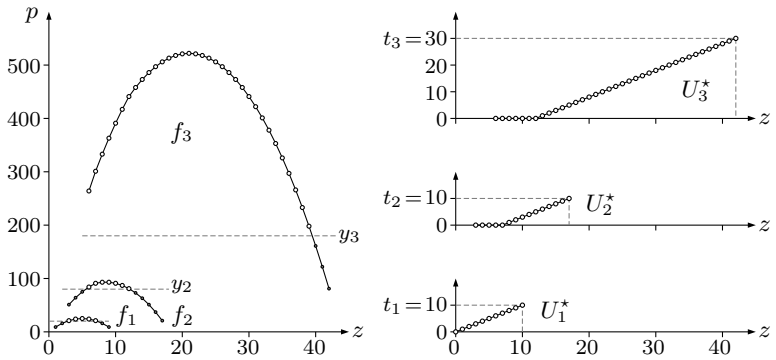
```
  vector<int> L = { 0 }, L2; // L[z] = maks. proizvodnja pri končni zmogljivosti z
  for (r = 1; r < d; ++r)
  {
    if (L.empty()) break; // V L so stanja na konec dne omejitve[r - 1].x;
    L2.clear();           // v L2 izračunajmo stanja na konec dne omejitve[r].x.
    const Omejitev &O = omejitve[r], &OP = omejitve[r - 1];
    int tr = O.x - OP.x; // trajanje trenutnega intervala
    if (tr <= 0) continue; // od omejitev z enakim x upoštevamo le prvo (z največjim y)
    if ((tr / 2) * (tr - tr / 2) >= k - OP.y)
      break; // če je interval dovolj dolg, bomo gotovo dosegli k
    int zMax = L.size() - 1; bool konec = false;
    for (int z = 0; z <= zMax + tr; ++z)
    {
      // Recimo, da hočemo na koncu intervala doseči zmogljivost z.
      // Izračunajmo največjo možno proizvodnjo.
      int p = -1;
      for (int u = max(0, z - zMax); u <= min(tr, z); ++u)
        if (L[z - u] >= OP.y) p = max(p, L[z - u] + (tr - u) * z);
      if (p < O.y) p = -1; // nismo dosegli omejitve
      L2.push_back(p);
      if (p >= k) { konec = true; break; }
    }
    if (konec) break; // Ali se da doseči proizvodnjo k?
    while (! L2.empty() && L2.back() < 0) L2.pop_back();
    swap(L, L2);
  }
  // Zdaj imamo v L stanja bodisi na koncu zadnjega intervala ali pa na koncu
  // nekega takega intervala, kjer lahko v naslednjem že dosežemo k. Vprašanje
  // je torej le še, kako čim hitreje doseči k.
  int tMin = -1, tZdaj = omejitve[r - 1].x;
  for (int z = 0; z < L.size(); ++z)
    // Če začnemo v (z, p) in nadgrajujemo u dni, bomo porabili skupaj
    // f(u) = u + ⌈(k - p) / (z + u)⌉ dni. Poiščimo minimum tega.
    if (int p = L[z]; p >= 0) for (int u = (z == 0 ? 1 : 0); ; ++u)
    {
      // Mogoče pri tem u že samo nadgrajevanje traja dlje kot nadgrajevanje in
      // proizvodnja skupaj pri najboljši doslej znani rešitvi. Če je tako,
      // bo pri večjih u samo še slabše in lahko končamo.
      if (tMin >= 0 && tZdaj + u > tMin) break;
      int zSkupaj = z + u;
      int kand = tZdaj + u + (k - p + zSkupaj - 1) / zSkupaj;
      if (tMin < 0 || kand < tMin) tMin = kand;
    }
  printf("%d\n", tMin); return 0;
}
```

Ta rešitev je za testne primere na našem tekmovanju več kot dovolj dobra, dalo pa bi se jo na razne načine še izboljšati. Po urejanju omejitev lahko omejitve r, pri katerih je $x_r > x_{r-1}$ in $y_r \le y_{r-1}$, preprosto zavržemo, saj vsak razpored, ki ustreže omejitvi $r-1$, gotovo ustreže tudi omejitvi r (ki ima kasnejši rok in nič višjo zahtevano proizvodnjo). Z zanko po z smo šli v gornji rešitvi od 0 do $z_{r-1}^{\max}+t_r$, toda lahko bi začeli pri $z_{r-1}^{\min}$ namesto pri 0, kajti če prvim $r-1$ omejitvam ni mogoče ustreči z manj kot toliko nadgradnjami, potem tudi prvim r-omejitvam ne bo; še več, lahko bi zanko začeli pri tistem z, kjer $f_{r-1}(z)$ doseže svoj maksimum, kajti pri manjših z vstopimo v r-ti interval z nižjo zmogljivostjo *in* nižjo proizvodnjo, od tega pa ne more biti nobene koristi. Še nekaj drugih izboljšav pa zahteva malo več razmisleka, da se prepričamo o njihovi pravilnosti, zato si jih bomo ogledali v naslednjih razdelkih.

Najboljši u pri izračunu $f_r(z)$. Naša dosedanja rešitev porabi največ časa za izračun vrednosti funkcije $f_r(z)$, ki je, kot se spomnimo, definirana kot $\max_u F_r(z)$, mi pa smo ta maksimum iskali z zanko po vseh primernih u, od $\max(0, z - z_{r-1}^{\max})$ do $\min(t_r, z - z_{r-1}^{\min})$.

Toda izkaže se, da je ta maksimum vedno dosežen pri najmanjšem možnem u — recimo mu $U_r(z) := \max(0, z - z_{r-1}^{\max})$. Ko se bomo prepričali, da to res drži, se bomo lahko zanki po u odpovedali in tako prihranili ogromno časa.

Oglejmo si za začetek majhen primer: recimo, da imamo tri omejitve, $(10, 20)$, $(20, 80)$ in $(50, 120)$, in da po dosedanjem postopku izračunamo zanje funkcije f_1, f_2, f_3. Spomnimo se, da je $f_r(z) = \max_u F_r(z, u)$; zapomnimo si vsakič, pri katerem u je bil ta maksimum dosežen; recimo mu $U_r^\star(z)$; zanj je torej $f_r(z) = F_r(z, U_r^\star(z))$. Graf spodaj levo prikazuje funkcije f_r, pri čemer črni krožci predstavljajo tiste vrednosti, ki ležijo pod omejitvijo y_r in jih zato v nadaljevanju postopka (pri izračunu f_{r+1}) ne uporabljamo; vodoravna črtkana črta kaže proizvodnjo y_r. Grafi spodaj desno pa prikazujejo funkcije U_r, ki nam povedo, koliko nadgradenj ima v r-tem intervalu najboljši razpored za $f_r(z)$; vodoravna črtkana črta kaže $u = t_r$, navpična pa $z = \hat{z}_r^{\max}$.



Na levem grafu vidimo, kako lepo konkavne oblike so funkcije f_r; vsaka torej najprej nekaj časa le narašča (in to vse počasneje), nato pa odtlej le še pada (in to vse hitreje). Še zanimivejši pa so grafi na desni, kjer vidimo, kako preprosta je funkcija $U_r^\star(z)$: sprva je ves čas 0, vzpenjati pa se začne v korakih po 1 in to šele takrat, da potem najvišjo možno vrednost t_r doseže pri najvišjem možnem argumentu, $z = z_{r-1}^{\max} + t_r$. Z drugimi besedami, velja torej $U_r^\star(z) = \max\{0, z - z_{r-1}^{\max}\} = U_r(z)$, tako da bomo najboljši u res lahko računali po preprosti formuli za $U_r(z)$.

Prepričajmo se zdaj, da ta opažanja res držijo v splošnem. Vpeljimo zapis $\triangle f_r(z) := f_r(z-1) - f_r(z)$, ki nam pove, za koliko se vrednost funkcije zmanjša, ko se ji argument poveča z $z-1$ na z. Neformalni povzetek našega razmisleka je naslednji: recimo, da bi radi prvih r intervalov končali z natanko z nadgradnjami (in čim večjo proizvodnjo), in da razmišljamo o tem, da bi na zadnjem od teh intervalov izvedli kakšno nadgradnjo več, kot je nujno potrebno, npr. $u+1$ namesto le u nadgradenj. Z vsako tako dodatno nadgradnjo izgubimo na r-tem intervalu en dan proizvodnje, ko bi lahko proizvedli z enot; toda po drugi strani na predhodnih intervalih, kjer moramo zdaj izvesti eno nadgradnjo manj, morda kaj pridobimo — natančneje, pridobimo $\triangle f_{r-1}(z-u)$ enot. Vprašanje je torej, ali pridobimo več, kot smo izgubili; z drugimi besedami, ali funkcija f_{r-1} kdaj pada tako hitro, da s tem, ko se ji argument zmanjša z $z-u$ na $z-u-1$, njena vrednost naraste za z ali več. Pokazali bomo, da niti pri največji vrednosti argumenta ne pada tako hitro (čeprav je že zelo blizu tega); da je konkavne oblike, zato pri nižjih vrednostih argumenta pada še počasneje; in da iz tega res sledi, da je najmanjši dopustni u tudi najboljši.

Zapišimo zdaj naš razmislek bolj formalno. Spomnimo se, da je funkcija $f_r(z)$ definirana na območju $\hat{z}_r^{\min}$ od $\hat{z}_r^{\max}$. Trdimo, (a) da pri $z = \hat{z}_r^{\max}$ velja $\triangle f_r(z) = z - 1$ in (b) da pri $\hat{z}_r^{\min} < z < \hat{z}_r^{\max}$ velja $\triangle f_r(z) \le \triangle f_r(z+1) - 2$ (kar lahko zapišemo tudi kot $\triangle f_r(z+1) - \triangle f_r(z) \ge 2$); in še, (c) da pri vseh z velja $f_r(z) = F_r(z, U_r(z))$.

Preden se lotimo dokazovanja te trditve, omenimo njeno koristno posledico: če neenačbo $\triangle f_r(z) \le \triangle f_r(z+1) - 2$ uporabimo po večkrat zapored, dobimo $\triangle f_r(z) \le \triangle f_r(z+1) - 2 \le \triangle f_r(z+2) - 4 \le \triangle f_r(z+3) - 6$ in tako naprej; v splošnem torej $\triangle f_r(z) \le \triangle f_r(z+w) - 2w$. Če gremo do $w = \hat{z}_r^{\max} - z$, dobimo

$$\begin{aligned} \triangle f_r(z) \le \;& \triangle f_r(\hat{z}_r^{\max}) - 2(\hat{z}_r^{\max} - z) = \\ =\; & (\hat{z}_r^{\max} - 1) - 2(\hat{z}_r^{\max} - z) = 2z - \hat{z}_r^{\max} - 1. \end{aligned} \tag{$\dagger$}$$

Lotimo se zdaj dokaza naše trditve. Dokazovali bomo z indukcijo po r; za začetek se prepričajmo, da trditev velja pri $r = 1$. Glede (c) je tako, da je v formuli $f_1(z) = \max_u F_1(z,u)$ tako ali tako mogoč en sam u, namreč $u = z$, saj moramo vseh z nadgradenj izvesti v edinem intervalu, ki ga imamo; najboljši in edini u je torej $u = z$, ravno tega pa nam tudi priporoči funkcija $U_1(z) = \max(0, z - z_0^{\max}) = \max(0, z-0) = z$. (To, da je $z_0^{\max} = 0$, sledi iz dejstva, da v 0 intervalih ne moremo izvesti več kot 0 nadgradenj.) Pri $r = 1$ imamo torej $f_1(z) = F_1(z,z) = f_0(0) + (t_1 - z)z = (t_1 - z)z$, funkcija pa je definirana od $\hat{z}_1^{\min} = z_0^{\min} = 0$ do $\hat{z}_1^{\max} = z_1^{\max} + t_1 = t_1$. Ta formula za $f_1(z)$ nam dá $\triangle f_1(z) = f_1(z-1) - f_1(z) = (t_1 - (z-1))(z-1) - (t_1 - z)z = 2z - 1 - t_1$. Pri $z = t_1 = \hat{z}_1^{\max}$ imamo torej $\triangle f_1(z) = z - 1$, torej (a) drži. Pri manjših z pa imamo $\triangle f_1(z+1) - \triangle f_1(z) = [2(z+1) - 1 - t_1] - [2z - 1 - t_1] = 2$, torej velja tudi (b).

Recimo zdaj, da smo našo trditev že dokazali do vključno $r-1$; preverili bi radi, da velja tudi za r. Oglejmo si najprej (c), ki govori o tem, pri katerem u je dosežen maksimum $\max_u F_r(z,u)$ v formuli za $f_r(z)$. Primerjajmo pri fiksnem z vrednosti $F_r(z,u)$ za dva zaporedna u: razlika med njima je $F_r(z,u+1) - F_r(u) = [f_{r-1}(z-u-1) + (t_r - u - 1)z] - [f_{r-1}(z-u) + (t_r - u)z] = \triangle f_{r-1}(z-u) - z = (\star)$. Seveda nas pri izračunu $f_r(z) = \max_u F_r(z,u)$ zanimajo le taki u, ki so $\geq \max(0, z - z_{r-1}^{\max})$, zato je $z - u \leq z_{r-1}^{\max} \leq \hat{z}_{r-1}^{\max}$, zato lahko za $z-u$ uporabimo posledico $(\dagger)$ naše trditve, kajti le-ta po induktivni predpostavki že velja za $r-1$; imamo torej $\triangle f_{r-1}(z-u) \leq 2(z-u) - \hat{z}_{r-1}^{\max} - 1$, zato pa $F_r(z,u+1) - F_r(u) = (\star) \leq 2(z-u) - \hat{z}_{r-1}^{\max} - 1 - z = (z - u - \hat{z}_{r-1}^{\max}) - (u+1)$; od teh dveh členov je prvi ≤ 0, ker je $z - u \leq \hat{z}_{r-1}^{\max}$, drugi pa je > 0, ker je $u \geq 0$; zato je razlika manjša od 0. Tako torej vidimo, da je $F_r(z,u+1) - F_r(u) < 0$, torej se $F_r(z,u)$ zmanjša vsakič, ko povečamo u za 1, tako da bo maksimum dosežen pri najmanjšem možnem u, to je pri $u = \max(0, z - z_{r-1}^{\max}) = U_r(z)$, torej (c) res velja.

S tem zdaj tudi vemo nekaj več o vrednostih funkcije $f_r(z)$. Na območju $z_{r-1}^{\max} \leq z \leq z_{r-1}^{\max} + t_r$ imamo $U_r(z) = z - z_{r-1}^{\max}$, na območju $z_{r-1}^{\min} \leq z \leq z_{r-1}^{\max}$ pa $U_r(z) = 0$. Če to vstavimo v $f_r(z) = F_r(z, U_r(z)) = f_{r-1}(z - U_r(z)) + (t_r - U_r(z))z$, dobimo:

$$f_r(z) = \begin{cases} f_{r-1}(z_{r-1}^{\max}) + (t_r - z + z_{r-1}^{\max})z & : z_{r-1}^{\max} \leq z \leq z_{r-1}^{\max} + t_r \\ f_{r-1}(z) + t_r z & : z_{r-1}^{\min} \leq z \leq z_{r-1}^{\max}. \end{cases}$$

Če potem izračunamo razliko dveh zaporednih $f_r(z)$, dobimo:

$$\triangle f_r(z) = \begin{cases} 2z - 1 - t_r - z_{r-1}^{\max} & : z_{r-1}^{\max} < z \leq z_{r-1}^{\max} + t_r \\ \triangle f_{r-1}(z) - t_r & : z_{r-1}^{\min} < z \leq z_{r-1}^{\max}. \end{cases}$$

Pri $z = \hat{z}_r^{\max} = z_{r-1}^{\max} + t_r$ nam ta formula pove, da je $\triangle f_r(z) = z_{r-1}^{\max} + t_r - 1 = z - 1$, torej velja (a). — Pri z z območja $z_{r-1}^{\max} < z < z_{r-1}^{\max} + t_r$ lahko izračunamo $\triangle f_r(z+1) - \triangle f_r(z) = \ldots = 2$, torej tu velja (b). — Pri $z = z_{r-1}^{\max}$ lahko izračunamo $\triangle f_r(z+1) - \triangle f_r(z) = \ldots = z_{r-1}^{\max} + 1 - \triangle f_{r-1}(z_{r-1}^{\max}) = (\star)$; ker po induktivni predpostavki naša trditev velja za $r-1$, nam $(\dagger)$ pove, da je $\triangle f_{r-1}(z_{r-1}^{\max}) \leq 2z_{r-1}^{\max} - \hat{z}_{r-1}^{\max} - 1$; zato je $(\star) \geq z_{r-1}^{\max} + 1 - [2z_{r-1}^{\max} - \hat{z}_{r-1}^{\max} - 1] = 2 + (\hat{z}_{r-1}^{\max} - z_{r-1}^{\max})$, kar je gotovo ≥ 2, saj je $z_{r-1}^{\max}$ po definiciji $\leq \hat{z}_{r-1}^{\max}$; tako torej vidimo, da (b) velja tudi pri $z = z_{r-1}^{\max}$. — In končno, pri z z območja $z_{r-1}^{\min} < z < z_{r-1}^{\max}$ lahko izračunamo $\triangle f_r(z+1) - \triangle f_r(z) = [f_{r-1}(z+1) - (z+1)] - [f_{r-1}(z) - z] = [f_{r-1}(z+1) - f_{r-1}(z)] + 1$; že prvi člen sam je po induktivni predpostavki (ker naša trditev velja za $r-1$) večji ali enak 2, torej velja (b) tudi tukaj. □

Ta razmislek je koristen še iz nekega drugega razloga poleg tega, da nam omogoča izogniti se notranji zanki po u pri izračunu $f_r(z)$: ob njem smo namreč potek funkcije f_r spoznali dovolj dobro, da lahko izpeljemo zgornjo mejo vrednosti $\hat{z}_r^{\max}$. Naj bo $z^\star$ tista vrednost z-ja, kjer doseže funkcija $f_r(z)$ svoj maksimum. Če je ta maksimum $\geq k$, se bo naš postopek tako ali tako nemudoma ustavil in preostanka funkcije niti ne bo računal; omejimo se zdaj na primer, ko je maksimum vendarle $< k$. Če zdaj z postopoma povečujemo od $z^\star$ navzgor, začne funkcija padati; in to padanje je vse hitrejše, saj smo videli, da je $\triangle f_r(z+1) \geq \triangle f_r(z) + 2$. Prvi padec je torej vsaj za 2, naslednji vsaj za 4, še naslednji vsaj za 6 in tako naprej. Po w korakih funkcija pade že vsaj za $2 + 4 + \ldots + 2w = w(w+1)$. Toda ker je bila vrednost funkcije že na maksimumu $< k$ in ker nikoli ne more pasti pod 0, morajo biti tudi vsi padci skupaj manjši od k; torej imamo $w(w+1) < k$, torej $w \leq \sqrt{k}$.

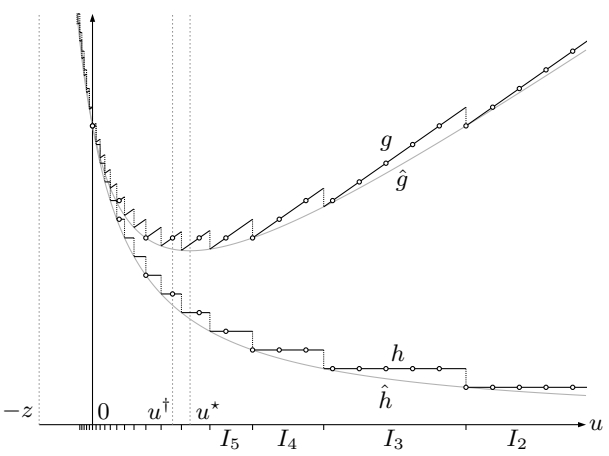
Podobno je tudi, če z postopoma zmanjšujemo od $z^\star$ navzdol; tudi zaradi $\triangle f_r(z-1) \le \triangle f_r(z) - 2$ funkcija pada vse hitreje, ko se z zmanjšuje. Enak razmislek kot prej nam tudi tokrat pokaže, da imamo lahko le $\sqrt{k}$ padcev.

Zaradi zveze $\triangle f_r(z) \le \triangle f_r(z+1) - 2$ tudi ni mogoče, da bi imela f_r pri treh zaporednih z-jih enako vrednost. Možen razpon z-jev, pri katerih je f_r sploh definirana, je torej širok le $2\sqrt{k} + O(1)$ elementov. Ker so poleg tega vsi z-ji pozitivni, lahko zaključimo, da je $\hat{z}_r^{\max} \le 2\sqrt{k} + O(1)$.

Najboljše število nadgradenj po koncu zadnje omejitve. Spomnimo se, da se moramo po tistem, ko dosežemo konec zadnjega intervala, odločiti še, koliko nadgradenj naj izvedemo po njem, da bomo potem čim hitreje dosegli želeno proizvodnjo k. Videli smo že, da pri tem iščemo minimum funkcije $g(u) = u + h(u)$ za $h(u) = \lceil (k-p)/(z+u) \rceil$, kjer je z dosedanja zmogljivost, $p = f_d(z)$ pa dosedanja proizvodnja; funkcija h torej pove, koliko dni proizvodnje potrebujemo, da dosežemo ali presežemo k. Doslej smo minimum g-ja iskali z zanko po u, v tem razdelku pa si bomo ogledali, kako lahko to počnemo učinkoviteje. Da bo manj pisanja, pišimo $\tilde{p} := k - p$; to je torej proizvodnja, ki nam še manjka do k.

Če ne bi bilo tistega $\lceil \cdot \rceil$, bi imeli funkcijo $\hat{g}(u) = u + \hat{h}(u)$ za $\hat{h}(u) = \tilde{p}/(z+u)$; in takšna $\hat{g}$ je odvedljiva: $\hat{g}'(u) = 1 - \tilde{p}/(z+u)^2$, kar je enako 0 pri $u^\star = \sqrt{\tilde{p}} - z$. Tam ima funkcija $\hat{g}$ svoj minimum; če se oddaljujemo levo ali desno od $u = u^\star$, vrednost funkcije $\hat{g}$ narašča.

Ker sta si funkciji $\hat{g}$ in g tako podobni, bi pričakovali, da bomo tudi minimum funkcije $g(u)$ našli nekje v bližini $u^\star$. Omejiti se moramo seveda na celoštevilske u, saj nam u pomeni število nadgradenj, to pa je vedno celo. Obetaven kandidat je torej $u^\dagger := \lfloor u^\star \rfloor = \lfloor \sqrt{\tilde{p}} \rfloor - z$. Prepričajmo se, da funkcija $g(u)$ res doseže svoj minimum ravno pri $u = u^\dagger$.



Primer za $z = 2$, $\tilde{p} = 32$. Graf kaže funkcije (od zgoraj navzdol) g, $\hat{g}$, h in $\hat{h}$ (g in h sta narisani z debelejšima črnima črtama, $\hat{g}$ in $\hat{h}$ pa s tanjšima sivima). Krožci kažejo vrednosti $g(u)$ in $h(u)$ za celoštevilske u. Navpične črtkane črte kažejo $u = -z$ (kjer imajo omenjene funkcije pol), $u = u^\dagger$ (kjer ima g svoj minimum) in $u = u^\star$ (kjer ima $\hat{g}$ svoj minimum). Oznake na vodoravni osi kažejo intervale I_h, na katerih je $h(u) = h$.

Razmislimo najprej, kaj se dogaja s funkcijo g pri majhnih u. Ko se u zmanjšuje, se $h(u)$ in $\hat{h}(u)$ povečujeta, in to še celo vse hitreje. Ker nas zanimajo le celoštevilski u, si oglejmo, kaj se zgodi, ko se premaknemo iz u v $u-1$. Tam se $\hat{h}$ poveča s $\hat{h}(u)$ na $\hat{h}(u-1)$ in če se tidve vrednosti razlikujeta za ≥ 1, potem gotovo nimata istega $\lceil \cdot \rceil$; takrat je torej $h(u-1) > h(u)$, torej se je v funkciji $g(u) = u + h(u)$ prvi seštevanec sicer zmanjšal za 1, drugi pa povečal za vsaj 1, torej se vrednost g pri tem ni zmanjšala: $g(u-1) \ge g(u)$. Takrat torej pri iskanju minimuma funkcije $g(u)$ nima smisla iti z u na $u-1$, kaj šele dlje v levo. Kdaj točno se to zgodi? Videli smo, da takrat, ko je $\hat{h}(u-1) - \hat{h}(u) \ge 1$; pišimo $w = z + u$, pa naš pogoj postane $\tilde{p}/(w-1) - \tilde{p}/w \ge 1$, torej $p \ge w(w-1)$, torej $w^2 - w - \tilde{p} \le 0$, kar je pri $w \le \frac{1}{2} + \sqrt{\tilde{p} + 1/4}$, torej $u \le \frac{1}{2} + \sqrt{\tilde{p} + 1/4} - z$. Naš $u^\dagger$ temu pogoju ustreza, kar pomeni, da manjših u-jev od njega ni treba gledati.

Kaj pa večji u-ji? Oglejmo si funkcijo $h(u)$ še malo pobliže. Zaradi $\lceil \cdot \rceil$ je ta funkcija odsekoma konstantna; pri katerih u je dosežena neka konkretna vrednost h? Z malo telovadbe se hitro vidi, da je to pri $u \in I_h := [\tilde{p}/h - z, \tilde{p}/(h-1) - z)$. Vodoravno os u lahko v mislih razdelimo na takšne intervale: $\ldots, I_{h+1}, I_h, I_{h-1}, \ldots, I_1$; vsak naslednji je širši od prejšnjega, zadnji od njih pa se celo razteza na desni v neskončnost. Na vsakem takem intervalu ima $h(u) = h$ konstantno vrednost, zato bo $g(u) = u + h(u)$

svojo najmanjšo vrednost (na tem intervalu) dosegla pri najmanjšem u s tega intervala. Toda spomnimo se, da nas zanimajo le celoštevilski u. Če je I_h širok vsaj 1 enoto, vsebuje gotovo vsaj eno celo število; naj bo u_h najmanjše celo število na tem intervalu; minimum funkcije g na tem intervalu je torej u_h+h. Na naslednjem intevalu, I_{h-1}, ki je še širši od I_h in torej gotovo tudi vsebuje vsaj eno celo število, je najmanjše celo število recimo u_{h-1}, ki je gotovo vsaj za 1 večje od u_h; najmanjša vrednost funkcije g na tem intervalu je potem $u_{h-1}+(h-1) \geq (u_h+1)+(h-1) = u_h+h$, torej minimum g-ja na I_{h-1} ni nič boljši kot na I_h. To pomeni, da čim dosežemo interval I_h, ki je širok vsaj 1 enoto, naslednjih intervalov (torej manjših h) ni več treba gledati. Pri katerih h pa nastopijo takšni intervali širine več kot 1? I_h je širok $\tilde{p}(1/(h-1)-1/h) = \tilde{p}/[h(h-1)]$, kar je >1 pri $h < h_D := \frac{1}{2}+\sqrt{\tilde{p}+1/4}$. Opazimo, da je $h_D > \sqrt{\tilde{p}}$, zato bomo namesto pogoja $h < h_D$ uporabljali strožji, a preprostejši pogoj $h \leq \sqrt{\tilde{p}}$.

Kmalu bomo videli, da pri $h = h(u^\dagger)$ ta pogoj še ni nujno izpolnjen; preden pridemo do dovolj majhnih h, moramo u še malo povečati. Toda videli bomo tudi, da pri tem povečevanju funkcija $g(u)$ nikoli ne doseže vrednosti, manjše od $g(u^\dagger)$. Pišimo $q := \lfloor\sqrt{\tilde{p}}\rfloor$; z drugimi besedami, q je celo število, za katero velja $q^2 \leq \tilde{p} < (q+1)^2$. Potem je $u^\dagger = q-z$ in $h(u^\dagger) = \lceil\tilde{p}/q\rceil$. V nadaljevanju razmisleka bomo območje $q^2 \leq \tilde{p} < (q+1)^2$ razdelili na tri dele in obravnavali vsakega posebej:

(a) Če je $\tilde{p} = q^2$, je $\tilde{p}/q = q$, torej celo število, zato $h(u^\dagger) = \lceil\tilde{p}/q\rceil = q$; takrat torej že $h = h(u^\dagger)$ ustreza pogoju $h \leq \sqrt{\tilde{p}}$, zato je I_h širok več kot 1 in pri večjih u ne bomo našli nobene manjše vrednosti funkcije g.

(b) Če je $q^2 < \tilde{p} \leq q(q+1)$, je $q < \tilde{p}/q \leq q+1$, torej je $h(u^\dagger) = \lceil\tilde{p}/q\rceil = q+1$. Ker je $(q-1)(q+1) = q^2-1 < q^2$, velja tudi $q-1 < q^2/(q+1) < \tilde{p}/(q+1) \leq q$, zato je $h(u^\dagger+1) = \lceil\tilde{p}/(q+1)\rceil = q$. Pri premiku iz $u^\dagger$ v $u^\dagger+1$ se je torej u za 1 povečal, h pa za 1 zmanjšal, zato je vrednost $g(u)$ ostala nespremenjena; pri $u = u^\dagger+1$ pa $h(u)$ že ustreza pogoju $h \leq \sqrt{p}$, torej je interval I_h širok več kot 1 in od tu naprej ne bomo našli nobene manjše vrednosti funkcije g.

(c) Ostane še primer, ko je $q(q+1) < \tilde{p} < (q+1)^2$. Ker sta $\tilde{p}$ in q cela, lahko desno neenakost zapišemo kot $\tilde{p} \leq (q+1)^2-1 = q(q+2)$. Velja torej $q+1 < \tilde{p}/q \leq q+2$, zato je $h(u^\dagger) = \lceil\tilde{p}/q\rceil = q+2$. Velja tudi $q < \tilde{p}/(q+1) < q+1$, zato je $h(u^\dagger+1) = \lceil\tilde{p}/(q+1)\rceil = q+1$. Velja pa tudi $(q-1)(q+2) = q^2+q-2 < q(q+1) < \tilde{p} \leq q(q+2)$, zato $q-1 < \tilde{p}/(q+2) \leq q$, tako da je $h(u^\dagger+2) = \lceil\tilde{p}/(q+2)\rceil = q$. Pri premiku iz $u^\dagger$ v $u^\dagger+1$ in nato iz $u^\dagger+1$ v $u^\dagger+2$ se torej u vsakič poveča za 1, h pa zmanjša za 1, zato $g(u)$ ostane nespremenjena; pri $u = u^\dagger+2$ pa imamo že $h(u) = q \leq \sqrt{\tilde{p}}$, torej smo na intervalu I_h širine več kot 1 in vemo, da od tu naprej ne bomo našli nobene manjše vrednosti funkcije g.

V vseh treh primerih torej vidimo, da pri $u > u^\dagger$ funkcija g nikoli ne pade pod vrednost $g(u^\dagger)$, torej smo pri iskanju minimuma funkcije g lahko zadovoljni že z $u^\dagger$. □

Zdaj torej vemo, da doseže $g(u)$ svoj minimum pri $u = u^\dagger = \lfloor\sqrt{k-p}\rfloor - z$; lahko pa se zgodi, da je to število negativno (če dosedanja zmogljivost z dovolj velika v primerjavi z manjkajočo proizvodnjo $k-p$), mi pa negativnega števila nadgradenj ne moremo izvesti, zato moramo takrat pač vzeti $u = 0$. Tako lahko v naši rešitvi zanko, v kateri smo proti koncu programa pregledovali različne u in iskali najmanjšo vrednost $g(u)$, zamenjamo z enim samim stavkom, ki preprosto izračuna $u = \max\{0, \lfloor\sqrt{k-p}\rfloor - z\}$.

Zgornja meja za število nadgradenj v najboljšem razporedu. Če ne bi imeli nobenih omejitev, bi lahko k odmerkov cepiva proizvedli tako, da bi najprej u dni nadgrajevali in nato $\lceil k/u\rceil$ dni proizvajali, torej v skupaj $g(u) = u+\lceil k/u\rceil$ dneh. V prejšnjem razdelku smo videli, da doseže ta funkcija minimum pri $u = \lfloor\sqrt{k}\rfloor$. Več kot toliko nadgradenj torej ne potrebujemo.

Ali bi se to kaj spremenilo, če bi imeli v nalogi tudi omejitve? Mislimo si nek nabor omejitev in zanj poiščimo najboljšo rešitev, torej tako, ki doseže proizvodnjo k v najmanj dneh; če je takih več, pa vzemimo med njimi tisto, ki ima najmanjše število nadgradenj, recimo z. Po zadnji nadgradnji nastopi v njej gotovo še nekaj proizvodnih dni, saj sicer od tiste zadnje nadgradnje ni bilo nobene koristi in bi se dalo rešitev za en dan skrajšati, če bi se tistemu zadnjemu dnevu z nadgradnjo popolnoma odpovedali.

Recimo torej, da po zadnji nadgradnji pride še t proizvodnih dni. Ali je mogoče, da je $t < z$? Potem bi lahko namesto z-te nadgradnje tisti dan proizvajali in takrat proizvedli $z-1$ odmerkov cepiva; v vsakem od naslednjih t dni pa bi po novem proizvedli en

odmerek manj kot prej. Skupna proizvodnja od začetka razporeda do vključno i-tega od tistih t proizvodnih dni bi se torej povečala za $z-1-i$, kar je ≥ 0, torej bi novi razpored še vedno ustrezal vsem omejitvam, ki jim je tudi prvotni. Zato je novi razpored še vedno optimalen (enako dolg kot prvotni), ima pa eno nadgradnjo manj; toda mi smo na začetku predpostavili, da je imel prvotni razpored najmanjše možno število nadgradenj (med vsemi optimalnimi razporedi). Tako smo v protislovju, torej mora biti $t \geq z$.

Za zadnjo, z-to nadgradnjo torej pride še vsaj z dni proizvodnje, v njih pa se proizvede vsaj z^2 odmerkov cepiva. Pišimo zdaj $q = \lfloor\sqrt{k}\rfloor$, tako da k leži na območju $q^2 \leq k < (q+1)^2$. Če je $z \leq q$, potem trditev, ki jo dokazujemo, velja tudi za naš trenutni nabor omejitev, torej je vse v redu. Ali se lahko zgodi, da bi bilo $z \geq q+2$? To bi pomenilo, da za zadnjo, $(q+2)$-go nadgradnjo pride še vsaj $q+2$ dni proizvodnje, takrat pa se proizvede $(q+2)^2$ enot cepiva; toda če bi se enemu od teh dni proizvodnje odpovedali, bi se takrat še vedno proizvedlo $(q+2)(q+1)$ enot, kar je že samo po sebi večje od k; torej prvotni razpored ni mogel biti optimalen, kar je protislovje.

Tako torej $z \geq q+2$ ni mogoče; kaj pa $z = q+1$? Tedaj pride po zadnji, $(q+1)$-vi nadgradnji še vsaj $q+1$ dni proizvodnje; toda več kot $q+1$ jih ne more biti, saj že v $q+1$ dneh proizvedemo $(q+1)^2$ odmerkov, kar je več kot k; če bi imeli $q+2$ ali več dni proizvodnje, bi lahko razpored skrajšali in še vedno ustregli vsem omejitvam, kar pa je nemogoče, saj smo začeli z optimalnim razporedom. Po zadnji nadgradnji imamo torej natanko $q+1$ dni proizvodnje.

Skupna proizvodnja v teh dneh je $(q+1)^2$, kar je $\geq k+1$. Če bi zdaj $(q+1)$-vo nadgradnjo zamenjali z dnevom proizvodnje, bi takrat proizvedli q odmerkov in za vsakega od naslednjih q proizvodnih dni bi veljalo, da je skupna proizvodnja od začetka razporeda do konca tistega dne vsaj tolikšna kot pred to spremembo. Po zadnjem, $(q+1)$-vem proizvodnem dnevu iz te skupine pa bi bila zdaj skupna proizvodnja za 1 manjša kot prej (v vsakem od $q+1$ starih proizvodnih dni smo izgubili eno enoto proizvodnje, pred tem pa smo pridobili q enot na tisti dan, ko smo nadgradnjo zamenjali s proizvodnjo). Toda to je tudi zadnji dan celotnega razporeda in omejitev takrat gotovo ni mogla biti tesna, saj bi v tem primeru ta omejitev znašala vsaj $(q+1)^2$, to pa je večje od k. Zato novi razpored gotovo še vedno ustreza vsem omejitvam, čeprav ima skupno proizvodnjo za 1 manjšo kot prej. Je enako dolg kot prvotni, torej je tudi novi razpored optimalen; ima pa tudi eno nadgradnjo manj kot prvotni, kar pa je v protislovju s predpostavko, da smo imeli že prej najmanjše možno število nadgradenj. Tako nas torej tudi $z = q+1$ pripelje v protislovje in lahko zaključimo, da je $z \leq q$. Z drugimi besedami, če ima naloga pri danem k in danem naboru omejitev sploh kakšno rešitev, potem med optimalnimi razporedi (takimi z najmanjšim številom dni) gotovo obstaja kak tak, ki ima kvečjemu $\lfloor\sqrt{k}\rfloor$ nadgradenj. □

To opažanje lahko uporabimo za drobno izboljšavo v naši rešitvi: doslej smo računali vrednosti funkcije $f_r(z)$ vse do $\hat{z}_r^{\max} = z_{r-1}^{\max} + t_r$, zdaj pa vidimo, da gotovo ne bomo spregledali nobene optimalne rešitve, če razporede z več kot $\lfloor\sqrt{k}\rfloor$ nadgradnjami ignoriramo. Dovolj je že, če tam, kjer imamo zdaj za nadaljevanje zanke po z pogoj `z <= zMax + tr`, dodamo še `&& z * z <= k`.

Prav veliko koristi od te izboljšave sicer ni, saj smo prej videli, da je $\hat{z}_r^{\max} \leq 2\sqrt{k} + O(1)$, torej smo zgornjo mejo naše zanke zdaj približno razpolovili (z $2\sqrt{k}$ na $\sqrt{k}$), red zahtevnosti pa je ostal enak.

Zgornja meja za dolžino najboljšega razporeda v odvisnosti od k in d. Pri danih k in d obstaja veliko primerov naše naloge; ti se razlikujejo med seboj po svojih omejitvah in imajo zato lahko tudi različne rešitve, torej porabijo različno veliko dni, da dosežejo proizvodnjo k ob upoštevanju vseh omejitev. Poskusimo najti neko zgornjo mejo za to; vprašajmo se torej, koliko dni porabimo za proizvodnjo k enot v najslabšem primeru, če imamo d omejitev, ki so razporejene na najbolj neugoden način.

Preden se lotimo tega vprašanja, zapišimo drobno opažanje, ki nam bo prišlo kasneje prav: recimo, da najboljši razpored vstopi v nek interval pri zmogljivosti z, interval pa je dolg t dni; če izvede razpored v tem intervalu še u nadgradenj, bo v preostanku intervala proizvedel $P(u) := (t-u)(z+u)$ enot cepiva. Odvod te funkcije je $P'(u) = t-z-2u$, kar je enako 0 pri $u = (t-z)/2$; ker pa nas negativni u tu ne zanimajo, lahko zaključimo, da je največja proizvodnja na tem intervalu dosežena pri $u = \max\{0, \lceil(t-z)/2\rceil\}$ nadgradnjah. Manj kot toliko nadgradenj se zagotovo ne splača narediti, ker bosta v tem primeru na

koncu intervala tako proizvodnja kot zmogljivost manjši ali enaki kot pri u nadgradnjah. Omejitve, bodisi tista na koncu trenutnega intervala bodisi kakšna kasnejša, nas lahko prisilijo v to, da izvedemo več kot u nadgradenj, ne pa manj kot toliko.

Razmislimo torej zdaj o zgornji meji za dolžino najboljšega razporeda. Omejimo se za začetek na naloge, pri katerih ni odvečnih omejitev, torej imamo $x_1 < x_2 < \ldots < x_d$ in $y_1 < y_2 < \ldots < y_d$, poleg tega pa predpostavimo še, da je $k = y_d$ in da te proizvodnje ni mogoče doseči prej kot na dan x_d.

Naših d omejitev nam razdeli časovno premico na d intervalov; v vsakem je pri optimalni rešitvi najprej nekaj (lahko tudi nič) nadgradenj in nato nekaj (lahko tudi nič) proizvodnih dni. Izberimo si neko število a; intervalom, pri katerih je zmogljivost ob koncu večja ali enaka a, bomo rekli visoko produktivni intervali, ostalim pa nizko produktivni. V visoko produktivnih intervalih se na vsak dan, ko poteka proizvodnja (in ne nadgradnja), izdela vsaj a izdelkov; zato je takih dni največ $\lceil k/a \rceil$. Poleg tega je v visoko produktivnih intervalih vsega skupaj tudi največ $\lfloor \sqrt{k} \rfloor$ dni nadgrajevanja, saj smo videli, da optimalna rešitev za določen k ne potrebuje več kot toliko nadgradenj.

Razmislimo zdaj še o nizko produktivnih intervalih; recimo, da jih je d_N (velja seveda $d_N \leq d$). V nekaterih od njih mogoče potekajo nadgradnje; takih intervalov je recimo q. Naj bo a_r število nadgradenj v r-tem od njih, naj bo $z_r = a_1 + \ldots + a_r$ skupno število nadgradenj v prvih r takih intervalih in naj bo t_r dolžina r-tega od njih. V r-tega torej vstopimo z zmogljivostjo z_{r-1}, zgoraj pa smo videli, da v takem primeru optimalna rešitev gotovo ne izvede manj kot $(t_r - z_{r-1})/2$ nadgradenj; torej je $a_r \geq (t_r - z_{r-1})/2$, torej $t_r \leq 2a_r + z_{r-1} = a_r + z_r$, torej je skupna dolžina teh intervalov $\sum_r t_r \leq \sum_r a_r + \sum_r z_r$; prva od teh vsot je enaka skupnemu številu nadgradenj v nizko produktivnih intervalih, kar je $< a$; druga vsota ima q členov, vsak od njih pa je $< a$; tako imamo $\sum_r t_r \leq a + q(a-1) = qa$.

Ostanejo še nizko produktivni intervali brez nadgradenj. Vsak od njih je lahko dolg kvečjemu a dni, saj bi sicer pri vstopni zmogljivosti $z < a$ in dolžini, večji od a in s tem tudi večji od z, v njem gotovo prišlo do vsaj ene nadgradnje.

Vsi nizko produktivni intervali skupaj so torej dolgi kvečjemu $qa + (d_N - q)a = d_N a \leq da$ dni; vsi visoko produktivni intervali skupaj pa so dolgi kvečjemu $k/a + \sqrt{k} + 2$ dni (prišteli smo 2, da se nam ni treba ukvarjati z zaokrožanjem navzgor pri k/a in $\sqrt{k}$). Skupna dolžina razporeda, recimo ji T, je torej $T \leq da + k/a + \sqrt{k} + 2$. To velja za poljuben a; predstavljajmo si desno stran te neenačbe kot funkcijo a-ja in jo odvajajmo: dobimo $f'(a) = d - k/a^2$, kar je enako 0 pri $a = \sqrt{k/d}$, takrat pa dobimo $T \leq 2\sqrt{kd} + \sqrt{k} + 2$, torej $T = O(\sqrt{kd})$. □

Da je ta meja vsaj v asimptotičnem smislu tesna, se lahko prepričamo tako, da sestavimo primeren nabor omejitev, pri katerem bo najboljša rešitev res porabila $O(\sqrt{kd})$ dni. Pri naših poskusih, da bi za različne k in d našli tak nabor omejitev, pri katerem bi optimalna rešitev zahtevala čim več časa (več o tem v naslednjem razdelku), je največ časa vedno zahteval nabor omejitev naslednje oblike: naj bo $b := \sqrt{k/(d+3)}$; imejmo najprej en interval dolžine $2b$ z zahtevano proizvodnjo b^2; nato imejmo $d-2$ intervala dolžine b z zahtevano proizvodnjo b^2 v vsakem intervalu; in končno imejmo še en interval dolžine $3b$ z zahtevano proizvodnjo $4b^2$. Tako imamo d intervalov s skupno proizvodnjo $(d+3)b^2$, kar je ravno k; rešitev izvede b nadgradenj v prvem in še b v zadnjem intervalu, drugače pa ves čas le proizvaja; in trajanje tega razporeda je $(d+3)b = \sqrt{k(d+3)}$ dni.

Na začetku tega razdelka smo se omejili na naloge brez odvečnih omejitev, s $k = y_d$ in pri katerih tega k ni mogoče doseči prej kot na dan x_d. Če dodamo odvečne omejitve, s tem seveda najboljši razpored ni nič daljši, kot če teh omejitev ne bi bilo, torej zanj velja meja za d' namesto d, če je d' število ne-odvečnih omejitev; in ker je naša meja $2\sqrt{kd}$ naraščajoča funkcija d-ja, ne moremo ničesar pridobiti, če namesto d vzamemo manjšo vrednost d'. Podobno je, če imamo nabor omejitev, v katerem je mogoče doseči k prej kot na dan x_d; bodisi lahko rok te zadnje omejitve skrajšamo ali pa, če lahko k dosežemo celo na dan x_{d-1} ali prej, lahko zadnjo omejitev pobrišemo, saj je odvečna; tako smo spet pri manjšem številu omejitev, s tem pa, kot smo videli, ne moremo ničesar pridobiti (pri iskanju nabora omejitev, ki bi kot optimalno rešitev zahteval čim daljši razpored). In končno, če imamo $k > y_d$ in je mogoče proizvodnjo k doseči šele po roku zadnje omejitve, torej po x_d, je učinek tak, kot da bi imeli $d+1$ omejitev namesto d, pri čemer bi nova omejitev zahtevala proizvodnjo k z najzgodnejšim rokom,

na katerega je to proizvodnjo mogoče doseči. Našo zgornjo mejo za trajanje razporeda pri najbolj neugodnem razporedu lahko še vedno uporabimo, le da moramo vzeti $d+1$ namesto d; dobimo $2\sqrt{k(d+1)}$. Med to mejo in $2\sqrt{kd}$ pa tako ali tako ni velike razlike: $2\sqrt{k(d+1)} = 2\sqrt{kd}\sqrt{1+1/d}$, tale zadnji faktor pa je blizu 1 in to tem bliže, čim večji je d.

Iskanje čim daljšega optimalnega razporeda pri danih k in d z dinamičnim programiranjem. V prejšnjem razdelku smo izpeljali zgornjo mejo za dolžino optimalnega razporeda v odvisnosti k in d v najslabšem primeru, torej po vseh možnih naborih omejitev za ta k in d. Zdaj pa si oglejmo še, kako lahko za dana k in d z dinamičnim programiranjem poiščemo konkreten primer čim daljšega optimalnega razporeda oz. nabora omejitev, pri katerem ta razpored nastane.

Omejili se bomo na primere, ko ni odvečnih omejitev (in imamo $x_1 < x_2 < \ldots < x_d$ in $y_1 < y_2 < \ldots < y_d$) in so vse omejitve tesne (optimalni razpored ravno doseže zahtevano proizvodnjo, preseže pa je ne) in je zadnja omejitev ravno enaka $y_d = k$, optimalni razpored pa jo doseže na dan x_d. (Če bi hoteli pokriti tudi primere, ko je $k > y_d$, bi lahko preprosto vzeli za 1 večji d in si mislili, da je tam na koncu še ena omejitev več.)

Naj bo $h(z, \ell, d)$ najmanjši p, pri katerem je mogoče sestaviti tak nabor d omejitev (pod pogoji iz prejšnjega odstavka), ki bo imel $y_d = p$, njegova optimalna rešitev bo dolga $\ell = x_d$ dni, od tega pa bo porabila z dni za nadgradnje, proizvedla pa bo natanko p enot cepiva. Potem pravzaprav iščemo (pri danih k in d) največji tak ℓ, pri katerem za neki z velja $h(z, \ell, d) \le k$.

Teh reči ni težko računati po naraščajočih d. Pri $d = 0$ je problem rešljiv le za $z = \ell = 0$, ko imamo $h(0, 0, 0) = 0$; drugod pa si mislimo $h(z, \ell, 0) = \infty$. Nato pa, ko poznamo rešitve za d, lahko razmišljamo takole: nabor omejitev, ki nas je pripeljal do $h(z, \ell, d)$, lahko podaljšamo s še enim intervalom dolžine t; na njem se bo izvedlo še $u := \max\{0, \lceil (t-z)/2 \rceil\}$ nadgradenj[3] in proizvedlo še $q := (t-u)(z+u)$ enot cepiva; tako torej vemo, da je vrednost $h(z, \ell, d) + q$ kandidat za vrednost $(z+u, \ell+t, d+1)$. To moramo ponoviti za vse z in ℓ, pri katerih je stanje (z, ℓ, d) dosegljivo, torej pri katerih je $h(z, \ell, d) < \infty$; pri vsakem pa moramo preizkusiti t-je do 1 naprej tako daleč, dokler $h(z, \ell, d) + q$ ne preseže k, kajti nabori omejitev, ki zahtevajo več kot k enot proizvodnje, nas ne bodo zanimalo. Če se nam pri istem stanju $(z', \ell', d+1)$ nabere več kandidatov, obdržimo med njimi seveda najmanjšega (tistega z najmanjšo skupno proizvodnjo); koristno si je zraven tudi zapisati, pri kateri vrednosti z in t smo ga dobili — tako bomo lahko na koncu tudi rekonstruirali nabor omejitev, ki pripelje do tega stanja.

Kot smo omenili že v prejšnjem razdelku, so najbolj neugodni nabori omejitev, ki smo jih s tem postopkom našli, zahtevali rešitve dolžine približno $\sqrt{k(d+3)}$ dni. Pri $k = 10^6$ in $d = 101$ (kar ustreza omejitvam iz naše naloge, pri čemer smo vzeli 101 namesto 100) nastane tako nabor omejitev, pri katerem je najboljša rešitev dolga 10198 dni.

Časovna zahtevnost naše rešitve. Razmislimo za konec še o časovni zahtevnosti naše rešitve. V prvotni različici moramo najprej za vsako omejitev r izračunati $f_r(z)$ vse do $z = \hat{z}_r^{\max}$, pri vsakem z pa imamo zanko po u, ki izvede v najslabšem primeru $O(t_r)$ iteracij. Spomnimo se, da je $\hat{z}_r^{\max} = O(\sqrt{k})$; če namesto tega uporabimo izboljšavo, da gledamo le z-je do $\sqrt{k}$, smo še vedno pri $O(\sqrt{k})$, zato o tej izboljšavi ne bomo govorili posebej. Pri vsakem r torej porabimo $O(\sqrt{k} \cdot t_r)$ časa, kar bo skupaj po vseh intervalih dalo $O(\sqrt{k} \sum_r t_r)$.

V tej zadnji fomuli nastopa $\sum_r t_r$, vsota dolžin vseh intervalov; ta je potencialno lahko zelo velika, toda spomnimo se, da po največ $O(\sqrt{kd})$ dneh gotovo že lahko dosežemo končno proizvodnjo k, takrat pa se glavna zanka našega programa ustavi; poleg tega pa, če bi bil posamezni t_r večji od $2\sqrt{k}$, bi se tam ustavili, ne da bi sploh poskusili

[3]Spomnimo se, da z omejitvami ne moremo prisiliti optimalnega razporeda, da bi na tem intervalu izvedel manj kot toliko nadgradenj; lahko pa bi ga prisilili, da jih izvede več, vendar smo se tej možnosti odpovedali. Pri sestavljanju takega nabora omejitev, ki zahteva čim daljši razpored, nam tako ali tako ni v interesu, da bi razpored prehitro nadgrajeval zmogljivost, kajti prej ko doseže visoko zmogljivost, manj proizvodnih dni se bo dalo dodati v razpored; ključno pri doseganju dolgega razporeda pa je ravno to, da imamo veliko dni proizvodnje (pri nizki zmogljivosti).

računati f_r. Tako torej vidimo, da od vsote $\sum_r t_r$ na našo časovno zahtevnost v resnici vpliva le $O(\sqrt{kd}) + O(\sqrt{k})$ dni; časovna zahtevnost izračuna vseh potrebnih f_r je torej $O(\sqrt{k} \cdot \sqrt{kd}) = O(k\sqrt{d})$.

Po tistem, ko izračuna vse f_r, mora naša rešitev pri vsakem z še razmisliti, koliko nadgradenj bi bilo najbolje izvesti po koncu zadnje omejitve. V prvotni različici gremo tu do $z = z_d^{\max}$ in pri vsakem z imamo zanko po u. Ta zanka po največ $\sqrt{k}$ iteracijah opazi rešitev, ki $\sqrt{k}$ dni nadgrajuje in $\sqrt{k}$ dni proizvaja; po največ $2\sqrt{k}$ iteracijah pa opazi, da zdaj še samo za nadgradnje porabi več časa kot pri najboljši rešitvi za nadgradnje in proizvodnjo skupaj, zato se ustavi. Ta zanka torej porabi $O(\sqrt{k})$ časa pri vsakem z; po vseh z skupaj (do $z_d^{\max}$, kar je $\leq \hat{z}_d^{\max} = O(\sqrt{k})$) je to $O(\sqrt{k} \cdot \sqrt{k}) = O(k)$. Ta del postopka je torej poceni v primerjavi z glavnim delom, ki je računal funkcije $f_1, \ldots, f_d$.

Razmislimo zdaj o glavnih dveh izboljšavah, ki smo si ju ogledali. Ena je, da pri izračunu $f_r(z) = \max_u F_r(z, u)$ ne uporabimo zanke po u, pač pa upoštevamo, da je maksimum vedno dosežen pri najmanjšem u, to je $u = \max\{0, z - \hat{z}_{r-1}^{\max}\}$. Tako imamo pri vsakem z le $O(1)$ dela, skupaj $O(\hat{z}_r^{\max}) = O(\sqrt{k})$ pri vsakem r, kar nanese $O(d\sqrt{k})$ pri vseh r skupaj.

Druga izboljšava je, da na koncu pri vsakem z ne iščemo števila nadgradenj po koncu zadnjega intervala z zanko po u, pač pa ga izračunamo po formuli $\max\{0, \lfloor k - f_d(z) \rfloor - z\}$. To pomeni, da imamo pri vsakem z le $O(1)$ dela, pri vseh skupaj pa zato $O(\sqrt{k})$.

Zaključimo torej lahko, da ima osnovna različica naše rešitve — tista, katere izvorno kodo smo si ogledali malo poprej — časovno zahtevnost $O(k\sqrt{d})$; z obema tu omenjenima izboljšavama pa se to zmanjša na $O(d\sqrt{k})$. V obeh primerih se spodobi prišteti še $O(d \log d)$ za urejanje omejitev na začetku postopka.

4. Virus v Timaniji

V opisu naloge vidimo, da bomo imeli opravka z dvema precej različnima skupinama testnih primerov: pri nekaterih bo ljudi veliko, vendar bo vsak okužil samo enega drugega, nato pa takoj umrl ($\ell = 1$, $n \leq 10^5$); pri drugih pa bo ljudi malo, vendar lahko vsak okuži po več drugih ($\ell \leq 15$, $n \leq 1000$). Ti dve različni vrsti problemov bomo tudi reševali na dva različna načina. Naj bo $S(u, d)$ človek, s katerim se sreča oseba u na dan d — to je tisto, kar dobimo kot vhodne podatke.

Pri $\ell = 1$ okuži vsakdo samo enega drugega človeka; vendar pa je to, koga točno okuži, odvisno od tega, kdaj se je sam okužil: če se u okuži na dan d, bo nato sam na dan $d' := (d + k) \bmod 7$ okužil človeka $S(u, d')$. Te povezave si lahko predstavljamo kot graf, v katerem so točke vsi možni pari oblike (u, d), torej človek ter dan v tednu, ko se je okužil; povezave v tem grafu pa naj bodo $(u, d) \to (S(u, d'), d')$, kot smo videli zgoraj. Iz vsake točke torej kaže ena sama povezava; hitro pa vidimo, da tudi v vsako točko kaže le ena povezava: človek u' se lahko na dan d' okuži le od tistega, s komer se ta dan sreča, torej od $S(u', d')$; in ker je ta človek kužen le en dan (nato pa umre), se je moral sam okužiti na dan $(d' - k) \bmod 7$. V točko (u', d') torej kaže povezava le iz točke $(S(u', d'), (d' - k) \bmod 7)$.

Začnimo v poljubni točki grafa in sledimo izhodnim povezavam: $u_0, u_1, u_2, \ldots$, toda ker ima graf le končno mnogo točk, moramo prej ali slej priti v neko točko, ki smo jo že obiskali; recimo, da se to prvič zgodi tako, da je $u_t = u_i$ za $i < t$. Če bi bila $u_i > 0$, bi to pomenilo, da smo v u_i prišli tako iz u_t kot iz u_{i-1}, kar je nemogoče, ker v vsako točko kaže samo ena povezava. Ostane le možnost $u_i = 0$, torej smo prišli nazaj v točko, kjer smo začeli; točke, ki smo jih prehodili, tvorijo cikel. Ves graf je sestavljen iz takšnih ciklov, ki so drug od drugega seveda ločeni (saj točke nimajo drugih izhodnih povezav kot naprej po svojem ciklu, tako da ni povezav med cikli).

Epidemija se torej ne bo mogla širiti iz enega cikla v drugega; cikle lahko obdelujemo vsakega posebej v zanki, dokler ne obiščemo vseh točk grafa. Recimo torej zdaj, da imamo pred seboj cikel $q_0, q_1, \ldots, q_{t-1}, q_t = q_0$. Če se epidemija začne v točki q_i na tem ciklu, se od tam razširi v q_{i+1}, pa v q_{i+2} in tako naprej. Edino, zaradi česar se to širjenje ustavi, je, če doseže nekega človeka, ki se je okužil že prej: če imamo recimo $q_i, q_{i+1}, \ldots, q_j, q_{j+1}, \ldots, q_{k-1}, q_k$ in se točki q_j in q_k nanašata na istega človeka (vendar na dva različna dneva v tednu), to pomeni, da se je ta človek nalezel bolezni v stanju q_j, nato jo je v stanju q_{j+1} prenesel na nekoga drugega in takoj zatem umrl (ker je $\ell = 1$);

ko torej pride čas za prenos $q_{k-1} \to q_k$, do njega ne more priti, ker je tisti, ki bi se moral v stanju u_k okužiti, že mrtev.

Na ciklu moramo torej poiskati čim daljši strnjen podniz točk, ki se nanašajo na same različne ljudi. To lahko naredimo tako, da gremo v zanki naraščajoče po i, nato pa pri vsakem i pogledamo, kako daleč pride okužba, če se začne pri q_i. Tega ni treba računati pri vsakem i od začetka; če se okužba začne pri q_{i+1} namesto pri q_i, se bo razširila vsaj tako daleč, kot se je prej, ko se je še začela pri q_i; preveriti moramo le, če se zdaj mogoče razširi kaj dlje. Zato je koristno v neki tabeli hraniti podatke o tem, kateri ljudje so prisotni v trenutno opazovanem podnizu; na desnem koncu lahko podniz podaljšujemo tako dolgo, dokler se naslednja točka nanaša na človeka, ki ga še ni v podnizu.

Oglejmo si zdaj še drugi del naloge, pri katerem je ℓ lahko večji od 1, vendar pa je ljudi malo. Takrat si lahko privoščimo preizkusiti vseh $7n$ možnih scenarijev (torej vseh možnih kombinacij tega, kdo se je prvi okužil in na kateri dan v tednu) in za vsakega od njih preprosto odsimulirati dogajanje, dokler se epidemija ne neha širiti.

Vzdrževali bomo tabelo s podatki o tem, kdo je še zdrav (tisti, ki niso, so ali že umrli ali pa še bodo); poleg tega potrebujemo še nekakšno čakalno vrsto okužb, do katerih bo prišlo v prihodnosti. Ko na primer med simulacijo pridemo do dneva d in vidimo, da se tisti dan okuži človek u, s tem tudi vemo, da bo ta človek v dnevih od $d+k$ do $d+k+\ell-1$ okužil tiste, s katerimi bo takrat v stiku, tako da moramo te okužbe dodati v čakalno vrsto pod ustreznimi dnevi.

V spodnji rešitvi je ta čakalna vrsta implementirana tako, da imamo po en seznam za vsakega od prihodnjih $k+\ell$ dni. Ker nam ne bo treba hraniti več kot $k+\ell$ takih seznamov naenkrat, jih bomo hranili kar v krožni tabeli (*ring buffer*), kjer seznam za dan d hranimo na indeksu $d \bmod (k+\ell)$. Vsak dan se sprehodimo po seznamu za tisti dan in ustrezno popravimo stanje ljudi na njem.

Simulacija se konča, ko so vsi seznami za prihodnjih $k+\ell$ dni prazni — ali, z drugimi besedami, takrat, ko ni več okuženih ali kužnih ljudi, ampak samo še zdravi in mrtvi.

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

int n, k, L; vector<array<int, 7>> T;                  // vhodni podatki
int uNaj = -1, dNaj = -1, pNaj = -1, nNaj = 0;  // rezultati

void Cikli()
{
  vector<bool> obiskan(n * 7, false), vNizu(n, false);
  vector<int> cikel;
  for (int u0 = 0; u0 < n; ++u0) for (int d0 = 0; d0 < 7; ++d0) if (! obiskan[u0 * 7 + d0])
  {
    // Začnimo okužbo z človekom u0 na dan d0.
    int u = u0, d = d0; cikel.clear();
    do {
      // u se je okužil na dan d; kdaj in koga bo okužil on?
      d = (d + k) % 7; u = T[u][d];
      cikel.push_back(u * 7 + d); obiskan[u * 7 + d] = true;
    } while (u != u0 || d != d0);

    // Poiščimo v ciklu najdaljši podniz, ki ne vsebuje po večkrat istega človeka.
    int m = cikel.size(), r = 0;
    for (int i = 0; i < m; ++i, --r)
    {
      // Če se podniz začne pri i in je dolg r členov, še ne vsebuje nobenega
      // človeka po večkrat. vNiz[u] nam pove, ali ta podniz vsebuje človeka u.
      // Podaljšajmo r, kolikor je le mogoče.
      while (! vNizu[cikel[(i + r) % m] / 7])
        vNizu[cikel[(i + r++) % m] / 7] = true;

      // Zapomnimo si najboljšo rešitev.
      int preziveli = n - r;
      if (pNaj < 0 || preziveli < pNaj)
```

```
      pNaj = preziveli, uNaj = cikel[i] / 7, dNaj = cikel[i] % 7, nNaj = 1;
    else if (preziveli == pNaj) ++nNaj;
    // Ko bomo premaknili i, torej začetek podniza, za en znak naprej,
    // bo s tem en človek izpadel iz podniza.
    vNizu[cikel[i % m] / 7] = false;
  }
  while (r > 0) vNizu[cikel[--r] / 7] = false; // pospravimo za sabo
 }
}

void Simulacija()
{
 vector<bool> zdrav(n, true); vector<int> mrtvi, kuzni;
 // vrste[d % M] je seznam ljudi, ki se bodo na dan d okužili.
 const int M = k + L; vector<vector<int>> vrste(M);
 for (int u0 = 0; u0 < n; ++u0) for (int d0 = 0; d0 < 7; ++d0)
 {
  // Začnimo okužbo z človekom u0 na dan d0.
  mrtvi.clear(); vrste[d0 % M].push_back(u0);
  // Simulirajmo, dokler se ima še kaj spremeniti.
  for (int dan = d0, toDo = 1; toDo > 0; ++dan)
  {
   // Kdo danes se danes okuži?
   auto &vrsta = vrste[dan % M];
   for (int u : vrsta) if (zdrav[u]) {
    zdrav[u] = false; mrtvi.push_back(u);
    // Koga vse bo u okužil in kdaj?
    for (int d = dan + k; d < dan + k + L; ++d)
     vrste[d % M].push_back(T[u][d % 7]), ++toDo; }
   toDo -= vrsta.size(); vrsta.clear();
  }
  // Zapomnimo si najboljšo rešitev.
  int preziveli = n - mrtvi.size();
  if (pNaj < 0 || preziveli < pNaj) pNaj = preziveli, uNaj = u0, dNaj = d0, nNaj = 1;
  else if (preziveli == pNaj) ++nNaj;
  for (int u : mrtvi) zdrav[u] = true; // pospravimo za sabo
 }
}

int main()
{
 // Preberimo vhodne podatke.
 scanf("%d %d %d", &n, &k, &L);
 T.resize(n); for (int u = 0; u < n; ++u) for (int d = 0; d < 7; ++d) scanf("%d", &T[u][d]);
 // Rešimo nalogo s primerno izbranim algoritmom.
 if (L == 1) Cikli(); else Simulacija();
 // Izpišimo rezultate.
 printf("%d %d %d %d\n", uNaj, dNaj, pNaj, nNaj); return 0;
}
```

5. Tja in spet nazaj

Hobitova pot je sestavljena iz dveh delov: tja (od zahoda proti vzhodu) in spet nazaj (od vzhoda proti zahodu). Če se uspemo nekako odločiti, katere točke bi obiskali v prvem delu, je s tem določena že cela pot: prvi del mora obiskati te točke v naraščajočem vrstnem redu njihovih x-koordinat, drugi del pa mora obiskati vse ostale točke v padajočem vrstnem redu x-koordinat.

Ker je torej vrstni red točk po x-koordinatah pomemben pri reševanju te naloge, je koristno, če si vhodne podatke za začetek uredimo po x-koordinati; v nadaljevanju bomo torej predpostavili, da velja $x_1 < x_2 < \ldots < x_n$.

Recimo, da bi želeli prvi del poti sestavljati postopoma po korakih in vanj dodajati točke eno po eno od zahoda proti vzhodu; in recimo, da je točka i zadnja, ki smo jo

doslej dodali v prvi del poti. To pomeni, da kasneje nobene od prvih točk $\{1, 2, \ldots, i\}$ ne bomo več dodali v prvi del poti; tiste, ki jih doslej še nismo dodali v prvi del, bomo morali torej obiskati v drugem delu. Zato bi se dalo že zdaj izračunati, kako dolgo pot bo drugi del opravil, da bo obiskal tiste točke; tako bomo pravzaprav sestavljali oba dela poti naenkrat, pri čemer bomo drugi del gradili v nasprotni smeri od tiste, v kateri ga bo hobit kasneje zares prehodil.

Vidimo lahko tudi, da pri tem razmisleku ni zares pomembno, kateri del poti je prvi in kateri drugi, kajti ko imamo enkrat pripravljen cel obhod (tja in spet nazaj), ga lahko hobit prehodi tudi v nasprotni smeri, pa bo rezultat enako dober. Rečemo lahko torej, da preprosto sestavljamo dve poti od zahoda proti vzhodu hkrati, pri čemer bo šla vsaka nova točka v natanko eno od teh dveh poti.

To je torej tisto, glede česar se bomo morali pri vsaki točki odločiti: ali bi z njo podaljšali eno ali drugo od naših dveh nastajajočih poti. Da pa bomo lahko izračunali, za koliko se pri tem poveča dolžina tiste poti, moramo vedeti, katera točka je bila doslej zadnja na tisti poti.

Naj bo torej $f(i, j)$ skupna dolžina obeh poti za najboljšo tako rešitev, pri kateri smo na obe poti že razporedili prvih i točk (ostalih pa še ne), pri čemer je zadnja točka na eni poti bila točka j (za $j < i$), zadnja točka na drugi poti pa je bila točka i. To slednje pomeni, da tista pot, ki vsebuje točko i, gotovo pred tem vsebuje tudi točke $j+1, j+2, \ldots, i-1$; brez točke i imamo torej pred seboj rešitev problema za prvih $i-1$ točk, pri čemer ima druga pot še vedno j kot zadnjo točko; najboljša rešitev tega pa je $f(i-1, j)$. Tako torej vidimo, da pri $j < i-1$ velja $f(i, j) = f(i-1, j) + d(i-1, i)$, pri čemer $d(\cdot, \cdot)$ pomeni razdaljo med točkama v oklepajih.

Malo drugače pa moramo obravnavati primer, ko je $j = i - 1$. Pot, ki bo zdaj vsebovala točko i, torej ne vsebuje točke $i - 1$ (kajti z njo se konča druga pot); zadnja točka pred i na njej je torej neka točka $k < i - 1$. Brez točke i imamo torej pred sabo poti, ki pokrijeta prvih $i - 1$ točk, pri čemer se ena konča pri k (druga pa seveda pri $i - 1$); najboljša rešitev tega podproblema pa je $f(i - 1, k)$. Ko potem našo pot do k podaljšamo s korakom od k do i, naraste skupna dolžina na $f(i - 1, k) + d(k, i)$. Ker ne moremo vnaprej vedeti, pri katerem k bo ta skupna dolžina najmanjša, moramo preizkusiti vse. Tako smo dobili:

$$f(i, i-1) = \min\{f(i-1, k) + d(k, i) : 1 \leq k < i - 1\}.$$

Poseben primer je še $i = 2$, ko imamo le dve točki in sploh nimamo nobene izbire: $f(1, 0) = d(0, 1)$.

Funkcijo f je koristno računati po naraščajočih i in shranjevati njene vrednosti v tabelo, da jih bomo imeli pri roki, ko jih bomo kasneje spet potrebovali. Ko računamo vrednosti funkcije za neki i, potrebujemo rezultate za $i - 1$, ne pa več tistih za $i - 2$, $i - 3$ itd., zato lahko tiste sproti pozabljamo. Tako je prostorska zahtevnost naše rešitve le $O(n)$, časovna pa $O(n^2)$.

Na koncu nas zanima dolžina celotnega obhoda. Recimo, da smo vse točke že razdelili med obe poti in da se je ena torej končala s točko n, druga pa s točko j za neki $j < n$. Najkrajša možna dolžina takih dveh poti je $f(n, j)$; da pa dobimo dolžino obhoda, moramo pot do j še podaljšati s korakom do n. Med tako dobljenimi obhodi moramo vrniti najkrajšega, to je $\max\{f(n, j) + d(j, n) : 1 \leq j < n\}$.

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

int main()
{
  // Preberimo vhodne podatke.
  int n; scanf("%d", &n);
  struct Tocka { int x, y; };
  vector<Tocka> T(n);
```

```
  for (auto &t : T) scanf("%d %d", &t.x, &t.y);

  // Uredimo točke naraščajoče po x.
  sort(T.begin(), T.end(), [] (const auto &t, const auto &u) { return t.x < u.x; });

  // Funkcija, ki vrne razdaljo med dvema točkama.
  auto D = [&T] (int i, int j) { double dx = T[i].x - T[j].x, dy = T[i].y - T[j].y;
                                 return sqrt(dx * dx + dy * dy); };

  // Rešimo nalogo.
  vector<double> f(n - 1), ff(n - 1);
  f[0] = D(0, 1); // Rešitev za prvi dve točki.

  for (int i = 2; i < n; i++)
  {
    // f[j] = najboljša rešitev za točke 0, ..., i - 1, pri čemer se ena pot konča v točki j,
    // druga pa v točki i - 1. Izračunajmo zdaj rešitve za točke 0, ..., i in jih shranimo v ff.
    ff[i - 1] = numeric_limits<double>::infinity();
    for (int j = 0; j < i - 1; ++j) {

      // V rešitvi, pri kateri se ena pot konča v i - 1, druga pa v j,
      // lahko podaljšamo prvo pot s korakom iz i - 1 v i.
      ff[j] = f[j] + D(i - 1, i);

      // Lahko pa podaljšamo drugo pot s korakom iz j v i.
      ff[i - 1] = min(ff[i - 1], f[j] + D(j, i)); }

    swap(f, ff);
  }

  // Poiščimo dolžino najkrajšega obhoda.
  double r = numeric_limits<double>::infinity();
  for (int j = 0; j < n - 1; ++j)

    // Rešitev, pri kateri se konča ena pot v n - 1 in druga v j,
    // lahko podaljšamo s korakom iz j v n - 1 in dobimo obhod.
    r = min(r, f[j] + D(j, n - 1));

  // Izpišimo rezultat.
  printf("%.6f\n", r); return 0;
}
```

Naloge so sestavili: gesla — Nino Bašić; povprečnež, kapniki, tja in spet nazaj — Tomaž Hočevar; kako dobri so virusni testi? — Boris Horvat; socialno omrežje, proizvodnja cepiva — Vid Kocijan; zlaganje loncev, virus v Timaniji — Jurij Kodre; svetilka — Mark Martinec; marsovci — Polona Novak; pletenje puloverja — Jasna Urbančič; rekonstrukcija poti — Anže Žagar; tetris — Borut in Peter Žnidar; pangramski podniz — Janez Brank.

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