18. tekmovanje ACM v znanju računalništva

25. marca 2023

NALOGE ZA PRVO SKUPINO OŠ

1. Trikotniki

Učitelj matematike je v šoli za domačo nalogo naročil učencem, naj doma narišejo nekaj trikotnikov in zapišejo dolžine njihovih stranic. Zdaj pregleduje njihove domače naloge in ugotavlja, da so si nekateri učenci oddana števila kar izmislili.

Napiši program, ki prebere tri pozitivna cela števila, urejena po velikosti od najmanjšega do največjega, in izpiše „`trikotnik`“ (brez narekovajev), če je mogoče narisati trikotnik s takšnimi dolžinami stranic, sicer pa naj izpiše „`ponaredek`“.

Trije primeri vhoda:	Pripadajoči izhodi:
`4 4 4`	`trikotnik`
`3 4 5`	`trikotnik`
`1 4 6`	`ponaredek`

2. Daljnovod čez podeželje

Napetost v daljnovodni vrvi (žici) daljnovoda je tako visoka, da jih ni praktično izolirati z gumo tako kot žice, ki jih srečamo vsak dan. Namesto tega jih obesimo zelo visoko in se zanašamo na to, da ni ničesar blizu njih, kamor bi lahko tok stekel; že previsoka veja drevesa lahko povzroča težave.

Imamo niz, dolg n znakov, ki predstavlja neki daljnovod. Znak „`T`“ predstavlja stolp daljnovoda, znak „`_`“ (podčrtaj) prazno polje, ki ne povzroča težav, znak „`o`“ pa krošnjo previsokega drevesa. Daljnovod se začne pri prvem `T`-ju niza in konča pri zadnjem. Znakom med dvema zaporednima stolpoma rečemo *razpetina*.

Napiši program, ki za dani n in niz daljnovoda pove, na koliko razpetinah obstaja kakšna previsoka krošnja.

Primer vhoda:	Pripadajoči izhod:
`20` `__o__T__TT_o_o_ToT__`	`2`

Komentar: v primeru je pred prvim stolpom krošnja, ki nas ne zanima, saj se daljnovod začne šele s prvim stolpom. Razpetini, ki nas motita, sta med tretjim in četrtim stolpom ter med četrtim in petim stolpom.

3. Predor

Malokdo ve, da se policija za lovljenje prehitrih voznikov poleg uporabe tako imenovanih radarjev poslužuje še enega premetenega načina. Na začetku in koncu predora

postavijo čitalec registrskih številk, nato pa merijo, kdaj je vozilo prišlo v predor in kdaj je zapeljalo iz njega. Če je izmerjen potovalni čas premajhen, vedo, da je vozilo šlo prehitro, in vozniku izstavijo kazen. Za delovanje sistema poleg čitalcev seveda potrebujejo tudi program, ki je sposoben iz dobljenih podatkov izračunati, katera vozila so šla prehitro. Za pisanje tega programa so se obrnili nate.

Na vhodu se v prvi vrstici nahajata števili n in t, ki predstavljata število zaznanih vozil in minimalni čas, ki ga mora vozilo porabiti za vožnjo skozi predor. Sledi n vrstic, ki opisujejo vozila. V vsaki od njih se nahaja registrska številka vozila, ki je zaporedje šestih malih črk angleške abecede, ter števili z in k, ki povesta, kdaj je vozilo prišlo v predor in kdaj ga je zapustilo. Vsi časi so podani od začetka merjenja. **Napiši program**, ki izpiše registrske številke tistih vozil, ki so skozi predor šla prehitro in jim je sedaj treba napisati kazen.

Primer vhoda:

```
2 3
abcdef 0 4
ghijkl 3 5
```

Pripadajoči izhod:

```
ghijkl
```

4. Besede

Sara bi rada iz kupa modelčkov črk sestavila besede in jih nato nanizala na vrvico. Ker pa ima na voljo le omejeno število modelčkov, bi rada besede nizala tako, da bi zadnjo črko ene besede in prvo črko naslednje besede združila in s tem prihranila en modelček. Za besedi `banana` in `avtomehanik` bi tako na vrvico nanizala `bananavtomehanik`. Sara je že izbrala besede, ki jih želi nanizati, zdaj pa jo zanima, ali so postavljene v pravilen vrstni red, da bo ob vsakem stikanju lahko eno črko na ta način izpustila. Ker je Sarinih besed kar precej, te je prosila, da zanjo **napišeš program**, ki bo ugotovil, ali besede ustrezajo njenim zahtevam.

Na vhodu je v prvi vrstici podan n — število besed, ki jih želi Sara nanizati na vrvico. Sledi n vrstic, v vsaki po ena beseda. Besede bodo vsebovale le male črke angleške abecede, dolge pa bodo kvečjemu 15 znakov.

Tvoj program naj izpiše `ustreza`, če je besede mogoče nanizati v danem zaporedju po Sarinem postopku, sicer pa izpiši `ne ustreza`.

Razmisli in **opiši** še, kako bi ugotovil, ali je mogoče dane besede preurediti, da jih bo Sara lahko nanizala.

Primer vhoda:

```
3
banana
avtomehanik
kenguru
```

Pripadajoči izhod:

```
ustreza
```

Še en primer vhoda:

```
2
kolo
mleko
```

Pripadajoči izhod:

```
ne ustreza
```

NALOGE ZA DRUGO SKUPINO OŠ

1. Napačna imena

Učitelj Uroš uči veliko učencev, zato ima težave pri pomnenju imen vseh učencev. Pogosto se mu zgodi, da koga pokliče po napačnem imenu. Prosil te je, da mu napišeš program, s katerim lahko preveri, če je ime pravilno in kje je morebitna napaka.

Napiši program, ki prebere dva niza; prvi je pravilno ime, drugi pa ime, ki se ga je spomnil Uroš. Izpiši „`pravilno ime`“ (brez narekovajev), če je Uroševo ime pravilno, drugače pa zaporedni položaj prve črke, ki se ne ujema med Uroševim in pravilnim imenom.

Vhodni podatki: v prvi vrstici je nahaja pravilno ime, v drugi pa ime, ki se ga Uroš spomni. Obe imeni imata največ 20 znakov, sta gotovo enake dolžine in sta sestavljeni samo iz črk angleške abecede.

Izhodni podatki: izpiši zaporedni položaj prve črke, v kateri se imeni razlikujeta (prva črka ima položaj 1); če sta enaki, pa izpiši „`pravilno ime`“.

Primer vhoda:	Pripadajoči izhod:
`Janez` `Janez`	`pravilno ime`

Še en primer vhoda:	Pripadajoči izhod:
`Timon` `Timor`	`5`

2. Ceneno potovanje

Letos imamo zaradi povišanih stroškov bivanja manj denarja za dopust. Ker si še vedno želimo iti na potovanje okoli sveta, smo se odločili, da za izbiro lokacij uporabimo posebno strategijo. Začeli bomo na ljubljanskem letališču, kjer bomo kupili najcenejšo možno karto. Ko se bomo nagledali te destinacije, bomo tudi tam kupili najcenejšo letalsko karto in odleteli naprej. Postopek bomo ponavljali, dokler se ne bomo nagledali dovolj mest, pristali nazaj v Ljubljani ali pa na letališču, od koder ne moramo nadaljevati.

Napiši program, ki bo iz podatkov o cenah letov izračunal, kje bomo pristali, če k-krat kupimo najcenejšo letalsko karto in se odpeljemo na to destinacijo. Če pred k-tim letom pristanemo nazaj na začetku, naj se program tam konča.

V nalogi so mesta predstavljena z zaporednimi številkami med vključno 1 in 1000. Ljubljansko letališče, kjer potovanje začnemo, ima vedno številko 1.

Vhodni podatki. V prvi vrstici bosta dani števili n in k. V i-ti od naslednjih n vrstic so podana števila a_i, b_i in c_i, ločena s presledkom; ta zapis pomeni, da enosmerna letalska karta iz mesta a_i v mesto b_i stane c_i evrov.

Zagotovljeno bo, da so vse cene letov izven nekega letališča med seboj različne (lahko pa imata dva leta iz različnih letališč enako ceno).

V 50 % testnih primerov bo na vsakem letališču mogoče kupiti največ eno letalsko karto (torej ne bomo imeli izbire, kam gremo v naslednjem koraku).

Omejitve: $2 \leq n \leq 10^5$; $1 \leq k \leq 10^5$; za vsak $i = 1, 2, \ldots, n$ bo veljalo tudi $1 \leq a_i \leq n$, $1 \leq b_i \leq n$ in $1 \leq c_i \leq 10^9$.

Izhodni podatki. Program naj izpiše dve števili, vsako v svojo vrstico: na katerem letališču s potovanjem končamo ter kolikokrat smo na celotnem potovanju leteli z letalom. Če smo postopek uspešno opravili, mora program torej kot drugo število izpisati k, če se je naša pot končala predčasno, pa manjše število.

Štirje primeri vhodov in pripadajočih izhodov:

Vhod:	Izhod:
8 3 1 2 500 2 3 174 3 1 200 12 7 100 1 12 120 1 3 350 7 3 500 7 2 300	2 3

Vhod:	Izhod:
3 2 1 2 120 2 3 130 3 1 140	3 2

Vhod:	Izhod:
3 500 1 2 2 2 1 2 2 3 3	1 2

Vhod:	Izhod:
5 10 4 3 20 5 4 2 1 5 3 1 5 10 5 4 200	3 3

3. Barvne packe

Na belo platno mečemo balončke, polne različnih barv. Preberi, koliko, kako velike balončke in kam smo jih vrgli, ter nariši, kako izgleda končna risba.

Prazno platno, na primer dolžine 10 in višine 5, si predstavljamo kot mrežo, ki jo lahko predstavimo s poljem samih pik:

```
..........
..........
..........
..........
..........
```

Če na tretji znak tretje vrstice vržemo balon neke barve, ki jo bomo označili z `#`, in velikosti 1, se bo razpočil in naredil packo v obliki kvadrata v razdalji 1 v vsako smer:

```
..........
.###......
.###......
.###......
..........
```

Če nato vržemo še en balon, npr. velikosti 0 in neke druge barve `*`, v četrto vrstico na četrto mesto, dobimo naslednjo risbo:

```
..........
.###......
.###......
.##*......
..........
```

Če vržemo nato še enega velikosti 2 in barve `?` v drugo vrsto, na zadnji znak, dobimo:

```
.......???
.###...???
.###...???
.##*...???
..........
```

Pri zadnjem primeru gre nekaj packe tudi mimo platna, ampak ta del nas ne zanima, saj ga ne bo na končni sliki.

Napiši program, ki prebere podatke o metih balonov in izriše končno stanje platna.

Vhodni podatki. V prvi vrstici so tri s presledkom ločena števila, d, v in p, ki predstavljajo dolžino ter višino platna in število metov. V vsaki izmed naslednjih p vrstic je opis enega meta; i-ta od teh vrstic vsebuje tri cela števila in en znak, x_i, y_i, s_i in b_i. Število y_i pove, v katero vrstico platna smo vrgli balon, število x_i pa, na katero mesto v tej vrstici. Število s_i predstavlja velikost packe, ki jo naredi ta balon, znak b_i pa njeno barvo. Mete izvajamo v takem zaporedju, kot so podani.

Omejitve:

- $1 \leq d \leq 1000$; $1 \leq v \leq 1000$; $0 \leq p \leq 100$;
- za vsak $i = 1, 2, \ldots, p$ bo veljalo $0 \leq s_i \leq 100$, $1 \leq x_i \leq d$, $1 \leq y_i \leq v$, znak b_i pa bo eden izmed znakov „`+`“, „`-`“, „`?`“, „`#`“, „`*`“, „`%`“ in „`$`“.

Dodatne omejitve:

- V prvih 30 % primerov bo višina platna enaka 1 (to pomeni $v = 1$), velikost packe s_i bo vedno 0, barva b_i bo vedno `#` in packa ne bo nikoli gledala prek roba platna.
- V naslednjih 50 % (skupaj 80 %) testnih primerov bo višina platna enaka 1, velikosti, barve in položaji pack pa so lahko poljubni.
- V zadnjih 20 % testnih primerov ni dodatnih omejitev.

Izhodni podatki: izpiši celotno platno po tem, ko smo nanj vrgli vse balone.

Primer vhoda:

```
10 1 3
5 1 0 #
8 1 0 #
1 1 0 #
```

Pripadajoči izhod:

```
#...#..#..
```

Še en primer vhoda:

```
20 1 3
2 1 3 *
20 1 0 $
6 1 1 #
```

Pripadajoči izhod:

```
*******............$
```

Še en primer vhoda:

```
10 5 3
3 3 1 #
4 4 0 *
10 2 2 ?
```

Pripadajoči izhod:

```
.......???
.###...???
.###...???
.##*...???
..........
```

Komentar: prvi primer ustreza dodatnim omejitvam za 30 %, drugi primer omejitvam za 80 %, tretji primer pa je splošen.

4. Dolge skladbe

Razvijamo aplikacijo za predvajanje glasbe z inovativno idejo; uporabili bomo namreč „tok simfonije“, kar pomeni, da se bodo skladbe ena za drugo zlile v eno skupno skladbo, brez očitnih meja med posamičnimi sestavnimi skladbami. Ko uporabnik pritisne tipko Igraj, se bo njegov seznam predvajanja pretvoril v to eno skupno skladbo in začel igrati.

Da prihranimo s procesorsko močjo, pa aplikacija ne bo naenkrat pretvorila celotnega seznama, temveč bo združevala skladbo po skladbo. Pri tem pa se pojavi težava: če uporabnik preskoči naprej (ali nazaj) v času predvajanja, moramo hitro izračunati, katera skladba naj bi se takrat predvajala, da jo lahko pretvorimo.

Napiši program, ki bo sprejel podatke o dolžinah skladb in odgovarjal na vprašanja tipa „Katera skladba se predvaja ob času t_j?“

Vhodni podatki. V prvi vrstici se nahajata števili n in q. V i-ti od naslednjih n vrstic je dano število ℓ_i, ki predstavlja dolžino i-te skladbe (te so na vhodu urejene tako, kakor so urejene v seznamu predvajanja). Sledi q vrstic, od katerih j-ta vsebuje število t_j, ki zaznamuje čas od začetka predvajanja, za katerega moraš izračunati, katera skladba se takrat vrti. Vsi časi so podani v sekundah.

Omejitve: $1 \le n \le 10^5$; $1 \le q \le 10^5$; za vsak $i = 1, 2, \ldots, n$ velja $1 \le \ell_i \le 10^9$; za vsak $j = 1, 2, \ldots, q$ velja $0 \le t_j \le \ell_1 + \ell_2 + \ldots + \ell_n$. V 20 % testnih primerov bo dodatno veljalo $q = 1$.

Izhodni podatki. Za vsako od q vprašanj izpiši zaporedno številko skladbe, ki se vrti ob času t_j. Odgovore piši vsakega v svojo vrstico. Skladbe so oštevilčene od 1 do n, kakor so podane na vhodu. Če je čas ravno na meji med dvema skladbama, izpiši tisto, ki se je ravno nehala predvajati.

Primer vhoda:

```
4 4
200
300
100
400
34
600
561
601
```

Pripadajoči izhod:

```
1
3
3
4
```

REŠITVE NALOG ZA PRVO SKUPINO OŠ

1. Trikotniki

Označimo stranice našega domnevnega trikotnika z a, b in c, pri čemer naj bo c najdaljša. Spomnimo se, da v trikotniku velja trikotniška neenakost: posamezna stranica je krajša kot ostali dve skupaj. To pomeni $a + b > c$, pa tudi $a + c > b$ in $b + c > a$. Zadnji dve neenakosti za naše a, b in c gotovo veljata že zaradi tega, ker je $c \geq a$ in $c \geq b$ (in ker so števila a, b in c pozitivna); prva neenakost, $a + b > c$, pa ne velja nujno in jo moramo preveriti. Če ne velja, lahko zaključimo, da trikotnik s stranicami a, b in c ne obstaja.

Če pa neenakost $a + b > c$ velja, trikotnik s stranicami a, b in c gotovo obstaja. O tem se lahko prepričamo na primer takole: vpeljimo v ravnini koordinatni sistem in to tako, da bo eno oglišče najdaljše stranice trikotnika v točki $(0, 0)$, drugo pa v $(c, 0)$. Tretje oglišče, recimo mu (x, y), mora potem ležati na oddaljenosti a od $(0, 0)$ in oddaljenosti b od $(c, 0)$. Tako imamo pogoja $x^2 + y^2 = a^2$ in $(x-c)^2 + y^2 = b^2$. Iz prve enačbe izrazimo $y^2 = a^2 - x^2$, kar lahko vstavimo v drugo in dobimo $(x-c)^2 + a^2 - x^2 = b^2$, iz tega pa sčasoma $x = (c^2 - b^2 + a^2)/(2c)$. Ker je $a+b > c$, je $a^2 + b^2 + 2ab > c^2$; ker sta a in b pozitivna, je $2ab > 0$, tako da dobimo $a^2 > c^2 - b^2$; zato pa je $x < (2a^2)/(2c) = a \cdot (a/c) \leq a$. In ker je $b \leq c$, je $c^2 - b^2 \geq 0$; in ker sta a in c pozitivna, je potem $x = (c^2 - b^2 + a^2)/(2c) \geq a^2/2c > 0$. Tako torej vidimo, da je $x \in (0, a)$; zato je v prej omenjeni enačbi $y^2 = a^2 - x^2$ desna stran večja od 0 in lahko y izračunamo kot $y = \sqrt{a^2 - x^2}$. Tako smo torej našli primerno točko (x, y), ki jo lahko vzamemo za tretje oglišče trikotnika in zanj potem vemo, da bo imel stranice dolžine a, b in c.

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

int main()
{
  // Preberimo stranice trikotnika.
  int a, b, c; cin >> a >> b >> c;
  // Izpišimo rezultat.
  cout << (a + b > c ? "trikotnik" : "ponaredek") << endl; return 0;
}
```

Naloga ne pove natančno, kaj naj naredimo v primerih, ko je $a + b = c$. Takrat je načeloma mogoč „trikotnik“ s temi stranicami, ki pa je v resnici izrojen v daljico. Zgornja rešitev take primere razglasi za ponaredke.

2. Daljnovod čez podeželje

Niz z opisom daljnovoda bomo pregledovali znak za znakom od leve proti desni. Pri tem bomo v spremenljivkah vzdrževali dva podatka: ali ima trenutna razpetina kakšno previsoko krošnjo (trenutnaPrevisoka) in koliko razpetin s previsoko krošnjo smo doslej že videli (stPrevisokih).

Spomnimo se, da se prva razpetina ne začne na začetku niza, pač pa šele pri prvem znaku T; zato ima spremenljivka trenutnaPrevisoka v spodnji rešitvi lahko tri

možne vrednosti: -1 pomeni, da sploh še nismo dosegli prve razpetine; 0 pomeni, da v trenutni razpetini še nismo videli nobene previsoke krošnje; 1 pa, da smo jo že.

Ko vidimo znak `T`, lahko pogledamo vrednost te spremenljivke in če je bila 1, to pomeni, da se tu končuje razpetina s previsoko krošnjo in moramo povečati števec takšnih razpetin. V vsakem primeru nato postavimo to spremenljivko na 0, saj se zdaj začenja nova razpetina (ali pa del niza za zadnjo razpetino, če je trenutni `T` zadnji v nizu).

Ko pa vidimo znak `o`, smo pri drevesu s previsoko krošnjo; če ima `trenutnaPrevisoka` takrat vrednost -1, je ne smemo spreminjati, saj še nismo pri prvi razpetini in nima trenutno drevo nobenega učinka. Če pa smo že pri neki razpetini, zanjo zdaj vemo, da vsebuje neko previsoko krošnjo, torej moramo postaviti `trenutnaPrevisoka` na 1.

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

int main()
{
  // Preberimo vhodne podatke.
  int n; string s; cin >> n >> s;

  // Preglejmo vhodni niz.
  int stPrevisokih = 0, trenutnaPrevisoka = -1;
  for (char c : s)
    if (c == 'T') {
      // Trenutna razpetina se končuje; poglejmo, če je imela kakšno previsoko krošnjo.
      if (trenutnaPrevisoka == 1) ++stPrevisokih;

      // Pripravimo se na naslednjo razpetino.
      trenutnaPrevisoka = 0; }

    else if (c == 'o' && trenutnaPrevisoka == 0)
      // Pravkar smo ugotovili, da je v trenutni razpetini previsoka krošnja.
      trenutnaPrevisoka = 1, ++stPrevisokih;

  cout << stPrevisokih << endl; return 0; // Izpišimo rezultat.
}
```

Pomembna podrobnost pri tej nalogi je, da moramo števec razpetin s previsoko krošnjo povečati šele, ko pridemo do `T`-ja na koncu razpetine, ne pa že takrat, ko prvič zagledamo kakšen `o` na njej, saj takrat še ne moremo vedeti, ali smo sploh še na razpetini ali pa morda že na območju za zadnjim `T`-jem v nizu (ki ne šteje za razpetino).

3. Predor

Podatke o vozilih berimo v zanki; pri vsakem sproti izračunajmo čas vožnje skozi predor, torej $k - z$, in preverimo, če je morda manjši od najmanjšega dovoljenega časa, torej ali je $k - z < t$. Če to drži, izpišimo registrsko število trenutnega vozila.

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

int main()
```

```
{
  // Preberimo število vozil in minimalni čas vožnje.
  int n, t; cin >> n >> t;

  // Obdelajmo vsa vozila.
  while (n-- > 0)
  {
    // Preberimo podatke o naslednjem vozilu.
    string regSt; int z, k; cin >> regSt >> z >> k;

    // Če je čas vožnje prekratek, izpišimo registrsko številko.
    if (k - z < t) cout << regSt << endl;
  }
  return 0;
}
```

4. Besede

Besede vhodnega zaporedja berimo v zanki; pri vsaki besedi moramo preveriti, če je njen prvi znak enak zadnjemu znaku prejšnje besede, zato je koristno, če si slednjega nekje zapomnimo — v spodnji rešitvi je to spremenljivka prejsnji. Če sta prvi znak trenutne besede in zadnji znak prejšnje različna, lahko takoj zaključimo, da vhodno zaporedje ne ustreza pogojem naloge, ne glede na to, kaj se bo v nadaljevanju zaporedja še dogajalo; za hranjenje tega podatka bomo imeli zato še eno spremenljivko (ustreza), ki nam bo na koncu tudi povedala, kaj moramo izpisati.

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

int main()
{
  bool ustreza = true; // Ali dosedanje zaporedje ustreza pogojem?
  char prejsnji = ' '; // Zadnji znak prejšnje besede.

  int n; cin >> n;      // Preberimo število vhodnih besed.

  // Preglejmo vhodne besede.
  while (n-- > 0)
  {
    // Preberimo naslednjo besedo.
    string beseda; cin >> beseda;

    // Preverimo, ali se njen prvi znak ujema z zadnjim zankom prejšnje.
    if (prejsnji != ' ' && beseda[0] != prejsnji) ustreza = false;

    // Zapomnimo si njen zadnji znak.
    prejsnji = beseda.back();
  }

  // Izpišimo rezultat.
  cout << (ustreza ? "" : "ne ") << "ustreza" << endl; return 0;
}
```

Mimogrede, ko opazimo neujemanje in postavimo ustreza na **false**, bi lahko glavno zanko tudi takoj prekinili, saj takrat že vemo, da je vhodno zaporedje neustrezno in bo takšno tudi ostalo.

Naloga sprašuje še, kako ugotoviti, ali lahko vhodno zaporedje postane ustrezno, če spremenimo vrstni red besed v njem. Za potrebe tega razmišljanja je pri posamezni besedi pomembno le, s katero črko se začne in s katero konča. Predstavimo zato naš seznam besed z grafom, ki ima po eno točko za vsako črko abecede (v abecedo bomo šteli le tiste črke, na katere se začenja ali končuje vsaj ena beseda našega seznama) in po eno povezavo za vsako besedo z našega seznama; če se beseda začne na črko u in konča na črko v, jo v grafu predstavlja usmerjena povezava $u \to v$. (Natančneje rečeno je to multigraf in ne navaden graf, ker je lahko v njem več povezav z enakim začetnim in enakim končnim krajiščem.)

Vsak sprehod v tem grafu zdaj ustreza nekemu zaporedju besed, v katerem se vsaka naslednja začne na tisto črko, na katero se prejšnja konča. Ker bi radi sestavili tako zaporedje iz vseh n besed, nas torej pravzaprav zanima, ali v grafu obstaja sprehod, ki uporabi vseh n povezav (vsako natanko enkrat) — to pa je ravno Eulerjev sprehod.

Za vsako črko u označimo z d_u razliko med vhodno in izhodno stopnjo točke u (ali, z drugimi besedami, med številom besed, ki se končajo na u, in besed, ki se začnejo na u). Prepričali se bomo, da obstaja Eulerjev sprehod natanko tedaj, ko je naš graf šibko povezan in ko bodisi (1) za vse črke u velja $d_u = 0$ bodisi (2) za eno črko velja $d_u = 1$, za eno $d_u = -1$ in za vse ostale $d_u = 0$.

($\Rightarrow$) Recimo, da obstaja Eulerjev sprehod in da se začne v točki v in konča v točki w. Na začetku torej točko v enkrat zapusti; nato v vsakem koraku vstopi v neko točko in jo v naslednjem koraku spet zapusti; in na koncu še enkrat vstopi v točko w. Pri tem tudi porabi vse povezave, vsako natanko enkrat, torej je razlika med številom vstopov v točko u in izstopov iz nje ravno enaka d_u. Tako torej vidimo, da je $d_v = -1$, $d_w = 1$, za ostale točke pa je $d_u = 0$ — to je lastnost (2). Poseben primer nastopi, če je $v = w$, torej če se sprehod začne in konča v isti točki; tedaj je število vstopov in izstopov tudi pri njej enako in imamo lastnost (1). — Ker Eulerejev sprehod uporabi vse povezave, obišče s tem tudi vse točke, saj smo v naš graf vzeli le take točke, ki imajo tudi kakšno povezavo (= le take črke, na katere se začne ali konča kakšna beseda), torej je graf res šibko povezan.

($\Leftarrow$) Ogledali si bomo Hierholzerjev algoritem za iskanje Eulerjevega sprehoda. Če ima graf lastnost (2), naj bo v tista črka, ki ima $d_v = -1$, in naj bo w tista črka, ki ima $d_w = 1$; če ima graf lastnost (1), vzemimo za v poljubno črko in naj bo $w = v$. Začnimo sprehod v točki v in pojdimo na vsakem koraku naprej po poljubni taki povezavi, ki je doslej še nismo uporabili. Ustavimo se, ko iz trenutne točke ne kaže nobena še neuporabljena povezava. Hitro se vidi, da se to zagotovo zgodi v točki w in ne kje drugje.[1] Če smo s tem porabili že vse povezave, imamo Eulerjev obhod in lahko končamo.

Sicer pa zdaj za vsako točko grafa velja, da je število še neuporabljenih vhodnih povezav vanjo enako številu še neuporabljenih izhodnih povezav. Začnimo v poljubni točki y, ki leži na dosedanjem sprehodu in ima kakšno neuporabljeno iz-

[1] Recimo, da se ustavimo v neki točki u. (a) Če je $u \neq v$, je število naših vstopov vanjo za 1 večje od števila izstopov; ker poti iz u ne moremo nadaljevati, to pomeni, da so vse izhodne povezave iz u že uporabljene; zato mora biti vhodnih povezav vsaj za 1 več kot izhodnih, torej $d_u \geq 1$, kar pa je mogoče le tako, da je $d_u = 1$ in $u = w$. (b) Če pa je $u = v$, je število naših vstopov vanjo enako številu izstopov in ker so vse izhodne povezave iz u že uporabljene, mora biti vhodnih vsaj toliko kot izhodnih, torej $d_v \geq 0$, kar pa je mogoče le, če ima graf lastnost (1) (in je $d_v = 0$), tedaj pa je $v = w$, torej spet vidimo, da smo se ustavili ravno v točki w.

hodno povezavo.[2] Iz nje spet nadaljujmo po neuporabljenih povezavah, dokler je to še mogoče. Podoben razmislek kot v prejšnjem odstavku nam pokaže, da se bomo ustavili spet v točki y; tako bomo dobili neki obhod (z začetkom in koncem v y), ki ga lahko vrinemo v naš dosedanji nastajajoči sprehod in slednji tako pokrije nekaj več povezav kot prej.

Postopek iz prejšnjega odstavka lahko zdaj v zanki ponavljamo, dokler ne uporabimo vseh povezav. □

Na vprašanje, ali je dane nize mogoče prerazporediti v ustrezen vrstni red, lahko torej odgovorimo tako, da pripravimo zgoraj opisani graf; nato izračunamo stopnje točk in preverimo, če ima eno od lastnosti (1) in (2); in končno z iskanjem v širino preverimo še, ali je šibko povezan.

[2] Prepričajmo se, da taka točka gotovo obstaja. Vemo, da obstajajo v grafu še neuporabljene povezave; naj bo x neko krajišče ene od njih. Ker je naš graf šibko povezan, obstaja gotovo pot (če zanemarimo smeri povezav) od v (začetne točke našega sprehoda) do x. Naj bo y zadnja točka na tej poti, ki leži na našem sprehodu, in naj bo y' njena naslednica na tej poti. Povezava med y in y' torej še ni bila uporabljena, saj bi sicer tudi y' ležala na našem sprehodu. Če kaže ta povezava iz y v y', ima torej y (ki leži na sprehodu) neuporabljeno izhodno povezavo, kar smo tudi iskali; če pa kaže povezava iz y' v y, ima y neuporabljeno vhodno povezavo, zato pa mora imeti tudi neko neuporabljeno izhodno povezavo, saj že vemo, da ima zdaj vsaka točka enako število obojih.

REŠITVE NALOG ZA DRUGO SKUPINO OŠ

1. Napačna imena

V zanki primerjajmo istoležne znake obeh vhodnih nizov, dokler bodisi ne opazimo neujemanja bodisi ne pridemo do konca nizov. Če smo opazili neujemanje, izpišimo indeks, kjer je do njega prišlo (pri tem pazimo, da se v C/C++ indeksi znakov štejejo od 0 naprej, mi pa ga moramo izpisati od 1 naprej); če pa smo prišli do konca nizov, izpišimo „`pravilno ime`“.

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

int main()
{
  // Preberimo vhodna niza.
  string s, t; cin >> s >> t;

  // Poiščimo prvo neujemanje.
  int i = 0; while (i < s.length() && s[i] == t[i]) ++i;

  // Izpišimo rezultat.
  if (i < s.length()) cout << (i + 1) << endl;
  else cout << "pravilno ime" << endl;
  return 0;
}
```

2. Ceneno potovanje

Ker z vsakega letališča vedno letimo z najcenejšim letom, je dovolj, če si od vseh letov s tistega letališča zapomnimo le najcenejšega. Ob branju vhodnih podatkov si torej pripravimo dve tabeli oz. vektorja: `kam[a]` pove, kam leti najcenejši let iz mesta a, `cena[a]` pa je cena tega leta. Nato lahko s pomočjo tabele `kam` enostavno simuliramo potek našega potovanja: na vsakem koraku gremo s trenutnega letališča v tisto, ki ga zanj določa tabela `kam`. Ustavimo se, če s trenutnega letališča sploh ni nobenega leta (kar je v spodnji rešitvi predstavljeno tako, da je v tabeli `kam` tam vrednost -1), če pridemo spet nazaj v začetno letališče 1 ali pa če smo že naredili k korakov.

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

int main()
{
  // Preberimo vhodne podatke. Za vsako letališče si zapomnimo le
  // ceno najcenejšega leta z njega ter to, kam ta let vodi.
  int n, k; cin >> n >> k;
  const int m = 1000; // število mest oz. letališč
  vector<int> kam(m + 1, -1), cena(m + 1, -1);
  for (int i = 0; i < n; ++i) {
    // Preberimo naslednji let.
    int ai, bi, ci; cin >> ai >> bi >> ci;

    // Če je to najcenejši let z „ai“ doslej, si ga zapomnimo.
```

```
    if (kam[ai] < 0 || ci < cena[ai]) kam[ai] = bi, cena[ai] = ci; }
  // Odsimulirajmo potovanje.
  int kje = 1, stKorakov = 0;
  do {
    // Morda s trenutnega letališča sploh ni nobenega leta.
    if (kam[kje] < 0) break;
    // Naredimo naslednji korak.
    kje = kam[kje]; ++stKorakov;
  // Ustavimo se, ko pridemo nazaj v 1 ali naredimo k korakov.
  } while (kje != 1 && stKorakov < k);
  // Izpišimo rezultat.
  cout << kje << endl << stKorakov << endl; return 0;
}
```

3. Barvne packe

Stanje platna lahko predstavimo z dvodimenzionalno tabelo oz. (kot v spodnji rešitvi) z vektorjem v nizov, ki so dolgi po d znakov in predstavljajo vsak po eno vrstico platna. Na začetku naj bodo vsi znaki pike, nato pa v zanki beremo podatke o packah s standardnega vhoda in jih rišemo na platno. Naloga sicer šteje koordinate od 1 naprej, mi pa jih bomo šteli od 0 naprej, da jih bomo lahko uporabljali kot indekse v vektor in nize; pri branju vhodnih podatkov zato pazimo, da koordinatam x_i in y_i odštejemo 1.

Packa s središčem (x_i, y_i) in velikostjo s_i je načeloma prisotna pri x-koordinatah od $x_i - s_i$ do $x_i + s_i$, razen če ležijo zunaj platna; v resnici moramo torej iti po x od $\max\{x_i - s_i, 0\}$ do $\min\{x_i + s_i, d - 1\}$. Podoben razmislek velja tudi za y-koordinate. Ko tako določimo koordinate pravokotnika, ki ga ta packa na platnu res pokrije, se lahko z dvema gnezdenima zankama sprehodimo po vseh celicah tega pravokotnika in vpisujemo barvo trenutne packe na ustrezna mesta v tabeli oz. vektorju, ki predstavlja platno.

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

int main()
{
  // Preberimo velikost in pripravimo prazno platno.
  int d, v, p; cin >> d >> v >> p;
  vector platno(v, string(d, '.'));
  while (p-- > 0) //Obdelajmo vse packe.
  {
    // Preberimo podatke o naslednji packi.
    int xi, yi, si; char bi; cin >> xi >> yi >> si >> bi; --xi; --yi;
    // Izračunajmo koordinate, ki jih ta packa pokriva.
    int xOd = max(xi - si, 0), xDo = min(xi + si, d - 1);
    int yOd = max(yi - si, 0), yDo = min(yi + si, v - 1);
    // Narišimo packo.
    for (int y = yOd; y <= yDo; ++y) {
```

```
      auto &vrstica = platno[y];
      for (int x = xOd; x <= xDo; ++x) vrstica[x] = bi; }
  }
  // Izpišimo končno stanje platna.
  for (const auto &vrstica : platno) cout << vrstica << endl;
  return 0;
}
```

4. Dolge skladbe

Naj bo $L_i := \ell_1 + \ell_2 + \ldots + \ell_i$ čas, ko se konča i-ta skladba. Ko nas zanima, katera skladba se vrti ob času t_j, torej pravzaprav iščemo tak i, za katerega velja $L_{i-1} < t_j \leq L_i$. Ali še drugače: iščemo najmanjši i, za katerega je $t_j \leq L_i$. (Pri kasnejših skladbah, recimo $k > i$, je pogoj $t_j \leq L_k$ tudi izpolnjen, vendar pa ni več izpolnjen pogoj $L_{k-1} < t_j$.) Ker je zaporedje $L_1, L_2, \ldots, L_n$ naraščajoče, lahko v njem najmanjši element, ki je večji ali enak t_j, poiščemo z bisekcijo; v C++ lahko uporabimo kar funkcijo lower_bound iz standardne knjižnice.

Ker imamo lahko do 10^5 skladb dolžine do 10^9, gredo lahko vrednosti L_i do 10^{14}, zato moramo zanje uporabiti kak 64-bitni podatkovni tip (v spodnji rešitvi je to **long long**). Pri izpisu rezultatov pazimo še na to, da moramo izpisovati številke skladb od 1 do n in ne od 0 do $n-1$.

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

int main()
{
  int n, q; cin >> n >> q;
  // Preberimo dolžine skladb in izračunajmo njihove kumulativne vsote.
  // Tako bo vsote[i] čas, ko se konča skladba i.
  vector<long long> vsote(n);
  for (int i = 0; i < n; ++i) {
    cin >> vsote[i]; if (i > 0) vsote[i] += vsote[i - 1]; }
  while (q-- > 0) // Obdelajmo poizvedbe.
  {
    long long tj; cin >> tj;
    // Z bisekcijo poiščimo prvo skladbo, ki se konča ob času t_j ali kasneje.
    int skladba = lower_bound(vsote.begin(), vsote.end(), tj) - vsote.begin();
    cout << (skladba + 1) << endl; // Izpišimo rezultat.
  }
  return 0;
}
```

Časovna zahtevnost te rešitve je $O(n + q \log n)$, namreč $O(n)$ za izračun vsot L_i in nato pri vsaki od q poizvedb po $O(\log n)$ časa za bisekcijo.

Nalogo lahko rešimo tudi tako, da poizvedbe uredimo naraščajoče po t_j in jih obravnavamo v tem vrstnem redu. Ker zdaj časi t_j ves čas le naraščajo, bodo tudi odgovori na poizvedbe ves čas le naraščali, zato lahko pregledujemo vrednosti L_i po vrsti in pri vsaki poizvedbi nadaljujemo pri tistem i, pri katerem smo se

pri prejšnji poizvedbi ustavili. (To si lahko predstavljamo tudi kot zlivanje dveh naraščajočih zaporedij, namreč $L_1, \ldots, L_n$ in $t_1, \ldots, t_q$.) Vendar pa moramo na koncu izpisati odgovore na poizvedbe v takem vrstnem redu, v kakršnem so bile v vhodnih podatkih, zato moramo pri urejanju poizvedb ob vsakem t_j hraniti še njegov prvotni indeks j.

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

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

  // Preberimo dolžine skladb in izračunajmo njihove kumulativne vsote.
  // Tako bo vsote[i] čas, ko se konča skladba i.
  vector<long long> vsote(n);
  for (int i = 0; i < n; ++i) {
    cin >> vsote[i]; if (i > 0) vsote[i] += vsote[i - 1]; }

  // Preberimo poizvedbe in jih uredimo naraščajoče po t_j.
  vector<pair<long long, int>> poizvedbe(q);
  for (int j = 0; j < q; ++j) {
    auto &P = poizvedbe[j]; cin >> P.first; P.second = j; }
  sort(poizvedbe.begin(), poizvedbe.end());

  // Izračunajmo odgovore na vse poizvedbe.
  vector<int> odgovori(q);
  int i = 0; for (auto [tj, j] : poizvedbe) {
    // Premaknimo se z i do prve skladbe, ki se konča ob času t_j ali kasneje.
    while (tj > vsote[i]) ++i;
    odgovori[j] = i; }

  // Izpišimo rezultate.
  for (int odgovor : odgovori) cout << (odgovor + 1) << endl;
  return 0;
}
```

Časovna zahtevnost te rešitve je $O(n + q \log q)$, namreč $O(n)$ za računanje vsot L_i, nato $O(q \log q)$ za urejanje poizvedb po t_j in potem $O(n+q)$ za zlivanje. To, katera od obeh tu opisanih rešitev je boljša, je torej odvisno od tega, ali je število poizvedb veliko v primerjavi s številom skladb ali obratno.

Naloge so sestavili: besede — Bor Grošelj Simić; trikotniki — Tomaž Hočevar; predor — Matija Likar; barvne packe — Jure Slak; daljnovod, napačna imena — Jaka Velkaverh; ceneno potovanje, dolge skladbe — Patrik Žnidaršič.